University of Bergen
Department of Informatics

Online learning through Reinforcement learning in a high-fidelity physics simulator

Author: Erik Hystad
Supervisor: Rodica G. Mihai

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November, 2022

Abstract

The research aim of this thesis is to explore model-based reinforcement learning in a high-fidelity physics simulator. This exploration implies answering the following questions:

- Can the world model of the DreamerV2 agent model future states of the high-fidelity physics simulator OpenLab?
- Can a reinforcement learning agent control the OpenLab drill string flow rate to reach down hole pressure goals?
- Given a change in the environment after finished training, does the DreamerV2 agent benefit from continuous updates to its weight parameters, from a short-term perspective?

To answer these questions a set of experiments was designed to use OpenLab as a reinforcement learning environment together with the model-based reinforcement learning agent DreamerV2. From the results of these experiments it was observed that the DreamerV2 agent was able to control and model selected parts of the OpenLab simulator. However the DreamerV2 was not able to show improvement from continuous updates to the weight parameters in a short-term perspective, in the analysed episodes. The results in this thesis indicates also that model-based reinforcement learning has a potential of solving more advanced problems in dynamic physics related environments, but the world model might show a lack of adaptability when faced with changes in the environment.

Acknowledgements

I would thank the people who has supported me throughout this thesis. First thanks to Rodica G. Mihai for great support and weekly guidance.

Fellow students who have provided a good humored working environment, especially Bendik for help with discussion and feedback.

Finally thanks to my family for the continued support.

Erik Hystad
Monday 21st November, 2022

Contents

1 Introduction **1**

2 Background materials **5**

2.1 Machine learning . . . 5
2.1.1 What is machine learning? . . . 6
2.1.2 Supervised and unsupervised learning . . . 7
2.1.3 Neural Networks . . . 8
2.1.4 Recurrent neural networks . . . 12
2.2 Reinforcement learning . . . 14
2.2.1 Introduction . . . 14
2.2.2 Agent-environment interaction . . . 15
2.2.3 Exploration and exploitation . . . 16
2.2.4 Rewards and Return . . . 16
2.2.5 Bellman equations . . . 17
2.2.6 Markov Decision Process . . . 17
2.2.7 On-policy and Off-policy . . . 18
2.2.8 Model-based learning . . . 19
2.2.9 Policy Gradients . . . 21
2.2.10 World Models . . . 23
2.3 Online and offline learning . . . 25
2.4 OpenLab . . . 29

3 Methodology and Experiment **31**

3.1 DreamerV2 . . . 31
3.1.1 Computational analysis . . . 32
3.1.2 Components of DreamerV2 . . . 33
3.1.3 Training . . . 37
3.1.4 Results . . . 40
3.1.5 PyDreamer . . . 42

3.2 Data analysis 43
3.2.1 Analysis of collected simulations 44
3.2.2 Analysis of isolated parameters 51
3.2.3 Insights from the analysis 57
3.3 Experiment setup and training 58
3.3.1 Problem 58
3.3.2 Environment 59
3.3.3 Training description 62
3.3.4 Evaluation 63

4 Results and Evaluations **65**
4.1 Results obtained during training 65
4.2 Evaluation 70
4.2.1 Modelling results 71
4.2.2 Control and online results 74
4.3 Further research 77

5 Conclusion **78**

Glossary **80**

Bibliography **82**

List of Figures

2.1 Illustration of machine learning compared to traditional programming . . 6
2.2 Linear regression as neurons 8
2.3 Simple neural network with two hidden layers 9
2.4 ReLU activation function 10
2.5 ELU activation function 11
2.6 Simple recurrent neural network 13
2.7 Agent-environment interaction 15
2.8 Model-free learning and model-based learning 20
2.9 Learning from both experience and model 20
2.10 Vision, memory and controller example from world models paper[11] . . . 24
2.11 Example of online learning process 26
2.12 Example of offline learning process 27
2.13 OpenLab simulator GUI. 30

3.1 Gaussian and categorical distributions in latent space, image from DreamerV2 blog[15] 33
3.2 Training the world model 35
3.3 Training the DreamerV2 agent 37
3.4 Training the Actor Critic with imgaination Markov Decision Process . . . 39
3.5 Figure a and b is the results reported from the "Deep Reinforcement Learning at the Edge of the Statistical Precipice"[1] paper, and figure c is the performance reported from "Mastering Atari with Discrete World models"[14]. 41
3.6 Simulation(1) with random set points parameters. 44
3.7 Correlation matrix of simulation 3.6 45
3.8 Simulation(2) data with random values for the set point parameters, with intervals between change. 46
3.9 Correlation matrix of simulation 3.8 47
3.10 Simulation(3) with random set points for flow rate, with intervals between change 48

3.11 Correlation matrix of simulation 3.10 49
3.12 Simulation(2) with normalized values 50
3.13 Linear regression of down hole pressure given bit depth 51
3.14 Simulation with only flow rate adjusted 52
3.15 Simulation with only flow rate adjusted, shortened 53
3.16 Simulation with only top of string velocity adjusted 54
3.17 Simulation with only top of string velocity adjusted, shortened 54
3.18 Simulation with only surface RPM adjusted 55
3.19 Simulation with only surface RPM adjusted, shortened 56

4.1 Example of episode 1 during training 66
4.2 Episode example after around 35 episodes of training 67
4.3 Episode example after around 85 episodes of training 67
4.4 Episode example after around 275 episodes of training 68
4.5 Episode example after around 300 episodes of training 69
4.6 Loss during training 69
4.7 Return during training 70
4.8 Predicted future pressure given first 5 states and actions, short episode. . 72
4.9 Predicted future pressure given first 5 states and actions, long episode. . 72
4.10 Log loss of the state reconstruction 73
4.11 Comparison of evaluation scenarios. 75
4.12 Episode from the end of evaluation with weight parameter updates. . . . 76

List of Tables

3.1 Computation comparison of model-based reinforcement learning agents, values from DreamerV2 paper [14]. 32
3.2 Minimum and maximum values for normalising future data 51
3.3 Initial set point values 60
3.4 DreamerV2 configuration 62
3.5 Hardware 62

Listings

2.1 Pseudocode of episode rollout from world models paper[11] 24
3.1 Straight-through gradients[2], pseudocode from DreamerV2 paper[14] . . 36
3.2 Episode following OpenAI gym[3] template. 60
3.3 Pseudocode of OpenLab environment 61

Chapter 1

Introduction

Reinforcement learning is a category within artificial intelligence and machine learning consisting of experience based learning to perform decision making. Recent progress of reinforcement learning has mainly been done with environments based on games, like chess and Atari. While there are exceptions, such as DeepMind's "Magnetic control of tokamak plasmas through deep reinforcement learning"[6], the application of model-based reinforcement learning in a high-fidelity physics simulator has not been explored extensively. The aim of this thesis is to explore and evaluate the decision making and Online learning capabilities of a model-based reinforcement learning agent in the high fidelity physics simulator OpenLab[28](https://openlab.app/).

The background for the thesis will be introduced in this chapter, followed by an introduction of the research problem and research aims. Next, the motivation of this exploration of model-based reinforcement learning will be discussed. Furthermore the limitations of the research and experiment is presented, before finally the structure of the thesis is laid out.

Thesis background and related work

Model-based reinforcement learning utilises a model of the environment in addition to the agent to improve sample efficiency. Model based reinforcement learning is often seen as a more computationally efficient and sample efficient option, dependent on the level of quality for the model of the environment.

Online learning can be described as learning from a continuous stream of data with the goal of managing to adapt with the environment when necessary. The ability to adapt with the environment is useful for a reinforcement learning agent to avoid acting on outdated experience if the environment evolves.

In this thesis the model-based reinforcement learning agent DreamerV2[14] will be used in conjunction with the well drilling simulator OpenLab. The DreamerV2 utilises a world model[11] to perform decision making and to learn a model of the environment. OpenLab is a well drilling simulator developed by NORCE[28], which will be utilised in this thesis to investigate how capable the DreamerV2 is in this environment. Unlike previous evaluations of the DreamerV2 agent, the OpenLab simulator is represented by a vector of values rather than an image. This is a point of uncertainty and interest to see if the categorical latent state representation of the DreamerV2's world model manages to represent the state with a vector based input rather than a high dimensional image input.

Model-based reinforcement learning with the use of world modes was first proposed by Ha et al[11], this concept introduced planning with latent state representations. Where the reinforcement leaning agent utilises the compact representation from the world model to make decisions in the environment. This was improved upon by Hafner et al[12] where they included a combination of a deterministic latent state and stochastic latent state to improve future state prediction and included model-predictive control[26] to plan the best action by looking at future states at each time step. Then Hafner et al[13] introduced the Dreamer agent which utilised *Latent Imagination* to train the behaviour of the agent, this was performed by imagining latent trajectories with the use of a world model.

One of the most notable examples of utilising reinforcement learning in a high-fidelity physics was performed by Degrave et al[6] to sculpt plasma into desired shapes. The agent was trained in a simulator designed to simulate the dynamics of a Tokamak reactor and validated using the real Tokamak reactor.

Research problem and motivation

Historically it has been difficult to learn a model for dynamic environments without the model knowing the rules of the environment. However recently there have been improvements in learned models that manage to learn the dynamics of the environment, and then perform better than their model-free alternatives, for example DeepMind's MuZero[29], DreamerV2 [14] and IRIS[20].

The research problem of the thesis is defined as: To explore model-based reinforcement learning in a dynamic high-fidelity physics simulator. We apply a model-based reinforcement learning agent, the DreamerV2, to the high-fidelity OpenLab simulator.

This is performed to see how capable the agent is in this environment. Central questions in this thesis would be:

- Can the world model of the DreamerV2 agent model future states of the OpenLab simulator given the actions preformed during the simulation?
- Can the DreamerV2 agent control the flow rate in the drill string in the OpenLab simulator to reach down hole pressure with respect to given pressure goals?
- Given a change in the environment after finished training, does the DreamerV2 agent benefit from continuous updates from a short-term perspective?

The outcome of the thesis could contribute to the extent of knowledge within model-based reinforcement learning in dynamic environments with a learned model and reinforcement learning in a physics and real world application related scenario. The use of a reinforcement learning agent with a learned model in a physics based simulation, while not extensively explored, has some interesting advantages. Both the resulting learned model of the environment, but also the ability to train with less interaction with the environment. This is especially an advantage if the simulator or environment can not be significantly sped up.

Limitations

The choice of agent in the thesis was limited by the available computation and data, both in regards to how much computation for the training of the agent and in regards to how much computation that was available for drilling simulation using OpenLab. Furthermore the action space of the agent was limited to only controlling the flow rate, instead of also controlling other parameters such as top of string velocity and surface RPM to simplify this initial exploration of OpenLab.

Thesis structure

This introduction chapter presented the thesis topic and research problem, also the model-based reinforcement learning agent, DreamerV2, was introduced together with the high-fidelity physics simulator OpenLab. Then finishing this chapter with outlining the thesis limitations and motivation for research. Chapter 2 describes the background materials and theory that the thesis is built upon, and the relevant concepts from machine learning, reinforcement learning, online and offline learning is reviewed. Afterwards there will be a short introduction of the drilling process and the OpenLab simulator. The Methodology and Experiment chapter explains the DreamerV2 agent, its components and training, an analysis of data from the OpenLab simulator, and finally the set up of the experiment to explore the DreamerV2 in the OpenLab simulator. The Results and Evaluation chapter presents the outcome from the experiment, discussing insights and suggesting further research. In chapter 5 the conclusion drawn from the experiment results is presented.

Chapter 2

Background materials

The background materials for the thesis are introduced and discussed in this chapter. To begin the introduction the overarching themes and concepts are presented. First, central concepts from machine learning is introduced in chapter 2.1, where we are looking at the building blocks from machine learning, that will be applied in chapter 3. Followed by chapter 2.2 where reinforcement learning is introduced and briefly touch upon the concept of world models and their strengths and weaknesses. Then online and offline learning 2.3, their differences and strengths and weaknesses. Finally a short introduction of the OpenLab simulator and its domain in chapter 2.4.

2.1 Machine learning

In the following chapter an introduction to the subject of machine learning as well as of the main concepts that forms the basis of this thesis. To introduce machine learning, we will explore the fundamental aspects of machine learning in the following chapter 2.1.1. After the introduction we will familiarize the machine learning subcategories, supervised and unsupervised learning in chapter 2.1.2. There will be an exploration into how a neural network and a recurrent neural network is built, including how we plan to utilize them in chapter 2.1.3 and 2.1.4.

2.1.1 What is machine learning?

Machine learning is a subcategory of Artificial Intelligence (AI) which is focusing on algorithms that can learn tasks without being explicitly programmed to do said task. This is done by learning from data where one usually trains on a data set to try to generalize from it. Instead of explicitly programming rules that take input x in a program to get result y, we train an algorithm by giving it a data set with input features and output results, which it can then train on to learn to predict y.

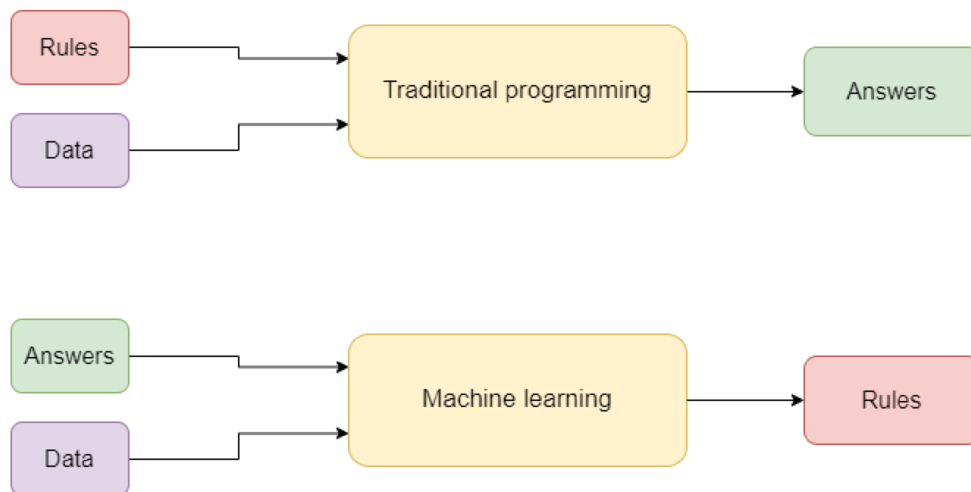


Figure 2.1: Illustration of machine learning compared to traditional programming

This makes a powerful tool to create programs that can be difficult to explicitly write the rules of, such as object detection. These algorithms or programs are often called *models* in the field of machine learning, since reinforcement learning also uses the term *model* for a part of reinforcement learning, models in machine learning context will be referred to as machine learning models.

From figure 2.1 it can be seen that machine learning receives an input x and an output y and returns a program to use. Compared to a traditional computer program where one would explicitly define the rules to create a program that receives an input x and gives us the answer y.

The definition "A computer is said to learn from experience E with respect to some class of tasks T and performance P, if its performance at tasks in T, as measured by P, improves with experience E."[21](Mitchell, 1997). Illustrates an algorithm that learns. The computer program learns from experience E, the data set, to solve a class of tasks

T, such as object detection or machine translation, and is measured with a performance metric P, often accuracy. If P improves with experience E the computer program is said to learn.

2.1.2 Supervised and unsupervised learning

It is common to divide Machine Learning into 3 different categories, supervised learning, unsupervised learning and reinforcement learning. In the following chapter we will focus on supervised and unsupervised learning and acquaint ourselves with reinforcement learning in chapter 2.2.

Supervised learning is essentially to train machine learning algorithms on labeled data. An example of this is a data set with input x and the the wanted result labeled y. The algorithm can be trained to learn how to approximate the function $f(x)$ by calculating the loss between the predicted $\hat{y}$ and the labeled y. Finally we update the the algorithm using some form of gradient descent multiplied by the loss. This simple approach of calculating how wrong predictions are and then updating the algorithm with gradient descent is straight forward to implement and has historically yielded good results.

The main drawback of this method is the expense to manually label data to be used. For each sample in a data set there needs to be a clarified wanted result, which can often be time consuming and expensive. An example of supervised learning is in object classification such as the MNIST[7] data set where each image of a handwritten number has a label of the number written, and is used to train machine learning models to learn how to recognize handwritten numbers.

Unsupervised learning is to train a machine learning algorithm without labeled data. Contrary to having the true y value to calculate the loss from, we need to find another approach to accurately calculate the loss. The machine learning model is learning patterns in the data to build a internal representation of the data. Clustering is a popular unsupervised task where the goal is to divide the data into groups. Another task is models that learn latent variables such as autoencoders and variational autoencoders, this can be used as a dense low dimensional representation of the information, similar to classical statistics methods such as Principal Component Analysis (PCA).

Autoencoders aim at recreating the input as accurately as possible while it is reduced in dimensionality by the encoder and then recreated by the decoder, resulting in an efficient representation of the data in a lower dimensionality.

The variational autoencoder(VAE), works similarly to the autoencoder. Instead of passing the latent variable directly to the decoder, the latent variable is utilized as the prior to a given distribution. We sample from said distribution, and pass our sample to our decoder to recreate the original input.

One of the main advantages of unsupervised learning is that the data is cheap. Since no human expert is needed to go through the entire data set to label all the samples, allowing for usage of a lot of collected data.

The DreamerV2 used in this thesis uses elements from the VAE in its world model, and the the original world model paper [11] uses a VAE to to encode and decode its latent state representation. While the DreamerV2 uses a vector of several categorical variables, exactly how the world model in DreamerV2 works will explained in chapter 3.1.

2.1.3 Neural Networks

Artificial neural networks, often called neural networks, is a machine learning algorithm that is central in deep learning and is what is used in the DreamerV2 agent. Neural networks imitate real life neurons that signal to one another. The simplest neural network is just one input and one output as in the figure 2.2 this is also the same as linear regression. A neural network is often described having several layers, with the terminology input layer, hidden layer and output layer. The input layer accepts the input data and passes it to the first hidden layer, see example figure 2.3. The hidden layer(s) can be one or more layers in the middle of the network extracting and creating features to represent the input data in a meaningful way. The final layer is the output layer which takes the output from the last hidden layer and transforms it to the wanted output, either it is a single regression value or probabilities for a classification task.

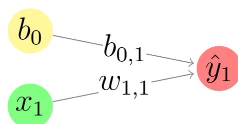


Figure 2.2: Linear regression as neurons

x_1 represents input, b the bias term, ans $\hat{y}$ the output, and w the weight. This illustrates linear regression as a simple neural network $\hat{y} = x_1 * w_1 + b$

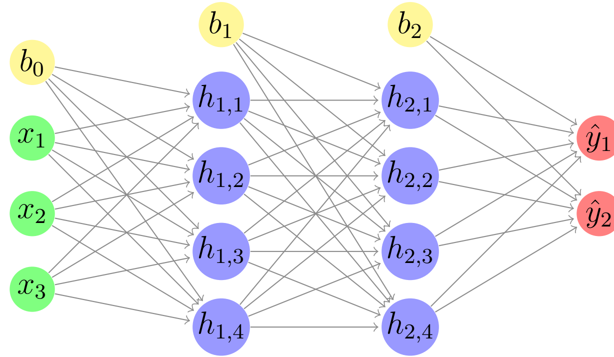


Figure 2.3: Simple neural network with two hidden layers

Here we have included two hidden layers to illustrate the structure of a artificial neural network with three input values and two output values.

Linear layer and convolutional layer

A linear layer in a neural network, sometimes called a fully connected layer, is a layer consisting of a simple matrix multiplication of the input from the previous layer and the weights and the addition of a bias term. It can be expressed as:

$$h = xw + b \tag{2.1}$$

Where x is the input vector, w is the weight vector and b is the bias term. This can also be viewed as one multiple input and multiple output linear regression, or as a single layer of neurons, see figure 2.3.

The main advantage of a linear layer is that it is very versatile since all inputs can affect all outputs, therefore also called a fully connected layer. This also makes it more computational inefficient since a lot of connections are redundant or not useful and the network needs to learn to ignore these and they add unnecessary computation. This is used in several part of the DreamerV2 model which is used in this thesis, the encoder and decoder, and the actor critic to name a few.

On larger input like images it is then a convolutional network that is often used. Here we will give a short introduction to convolutional networks, and we base this introduction on chapter 9 in "the deep learning book"[9]. Goodfellow describes convolution as "Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. "[9]

The DreamerV2[14] which is used in this thesis uses convolutional networks in the encoder and decoder, since it has been used with visual input. However in this thesis

we will use linear layers instead since the state representation is not visual. The main strength of convolutional layers is sparse interactions parameter sharing and equivarian representations.

Activation function

The activation function is a non-linear transformation that is applied to each unit between two layers. This is is done to introduce non-linear transformations to the network to be able to solve non-trivial problems, since without activation functions neural networks would only be a set of linear transformations. One of the most common activation functions is ReLU which is short for rectified linear activation unit, it is simply:

$$f(x) = max(0, x) \tag{2.2}$$

One of ReLU's main strengths compared to other commonly used activation functions such as sigmoid and tanh is the derivative of ReLU does not decrease when x increases which can be a factor to create vanishing gradients which can be a large problem in neural networks.

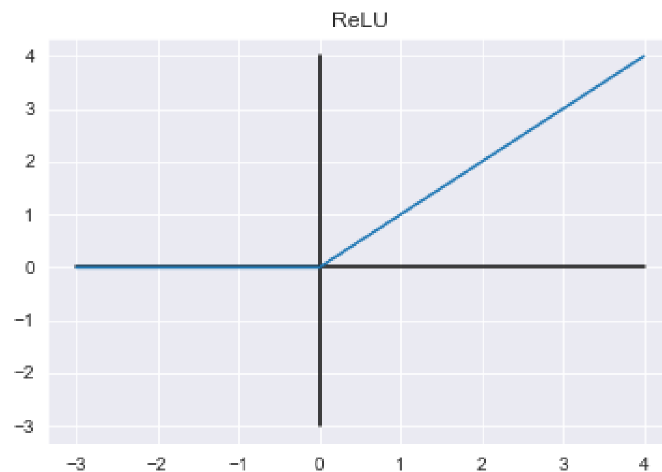


Figure 2.4: ReLU activation function

In the DreamerV2 both the world model and the actor critic uses ELU[14], the exponential linear unit. The formula for ELU is:

$$\begin{aligned} f(x) = & x <= 0 : \alpha(e^x - 1), \\ & x > 0 : x \end{aligned} \tag{2.3}$$

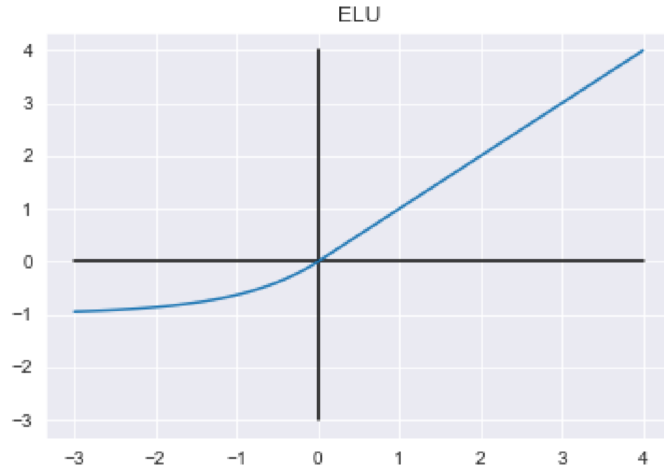


Figure 2.5: ELU activation function

ELU with $\alpha = 1$, where the function produces negative values down to -1.

The main difference of ELU compared to ReLU is ELU can produce negative results down to a value of α, see figure 2.5.

Loss function

The loss function is how we evaluate how well a machine learning model is preforming, where the goal is to minimize the error in the predictions from the machine learning model. Two common loss functions are mean squared error and cross entropy, also called log loss. Where mean squared error is a common loss function for regression, calculated as:

$$mse = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.4}$$

Where n is the number of data points. Cross entropy is commonly used loss function for classification of multiple classes, calculated as:

$$ce = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) \tag{2.5}$$

Where n is the number of classes that can be classified.

In our case we are doing reinforcement learning our goal is to maximize the sum of rewards, see chapter 2.2.4. The world model in DreamerV2 minimizes the loss with a custom loss function which is a combination of several well known loss functions such as Kullback-Leibler loss and negative loss-likelihood, see chapter 3.1.3.

Backpropagation and Gradient Descent

The backpropagation algorithm[27] is used to calculate the gradients of the weights with respect to the loss function, then it is simple to use gradient descent, or one of its variants, to minimize the loss function. The chain rule of calculus is used to calculate the derivatives. In case of several layers in a neural network, or another set of functions, such as $y = outputlayer(z_1)$, $z_1 = hiddenlayer(z_0)$, and $z_0 = intputlayer(x)$ then the derivatives can be calculated by applying the chain rule as follows:

$$\frac{dy}{dx} = \frac{dy}{dz_1}\frac{dz_1}{dz_0}\frac{dz_0}{dx} \tag{2.6}$$

Using gradient descent the weights of the network can be updated, to be expressed as:

$$w = w - \alpha \nabla L(w) \tag{2.7}$$

Where ∇L is the gradient of the loss with respect to the weights, α is the learning rate which dictates how large steps in the direction of the gradient should the network take. A lower learning rate will be more accurate, but also slower to train. Using gradient descent to minimize the loss function can be interpreted as the gradient gives information on what affected the output the most, if the result was advantageous, do more of what was just done, if the result was negative do less of that.

In the DreamerV2 the Adam[18] optimization method is used, which is an extension to stochastic gradient descent. The Adam update step is defined as:

$$\theta_t = \theta_{t-1} - \alpha * \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{2.8}$$

The main difference between Adam and stochastic gradient descent is m_t the exponential moving average of the gradient and v_t the element wise squared gradient. The hyper parameters for Adam is the learning rate α, and the decay rate of m_t which is controlled with β_1 and decay rate of v_t which is β_2.

2.1.4 Recurrent neural networks

The information in chapter 2.1.4 is primarily based on chapter 10 in the Deep Learning book[9]. Recurrent neural networks are neural networks specialized for processing a

sequence of input values, such as words in a sentences or a time series. The calculation of several steps in a sequence with a recurrent neural network can be expressed as:

$$h^t = f(h^{t-1}, x^t; \theta) \tag{2.9}$$

Where h^t is the hidden state for index t, x^t is the input for index t, θ is the weights. To visualize the concept, see figure 2.6, we can unroll the function to understand how the predicted value of time step t is calculated from the hidden state from the previous time step, and the input from the current time step. A recurrent neural network evaluates the input from left to right to take into account the history of the previous states. This is a benefit since the data from the OpenLab simulator is conditioned on the previous time steps of the well trajectory which will enable a more accurate prediction of the simulation. A demonstration of relation between current and past time steps is if the down hole pressure is $3.5e7\ Pa$, it has different consequences if the pressure is rising or if it is decreasing, i.e. if the pressure is decreasing it the probability that the pressure is lower than $3.5e7\ Pa$ in the next time step is higher than the probability that the pressure is higher than $3.5e7\ Pa$. Recurrent neural networks can be bidirectional, being able to evaluate input from left to right and right to left, but this ability is not applicable in our setting, since previous time steps is not influenced by future time steps.

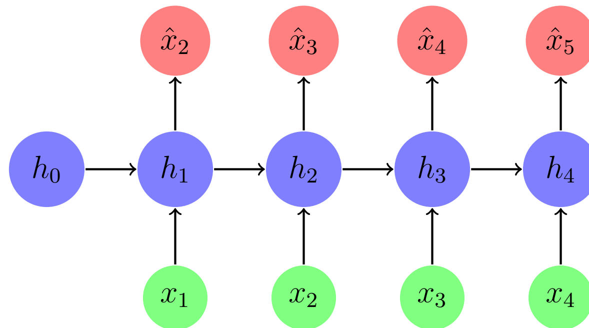


Figure 2.6: Simple recurrent neural network

Example of unrolled many to many recurrent neural network. For each time step t the recurrent neural network produces $\hat{x}_{t+1} h_t = f(h_{t-1}, x_t; \theta)$, where x_t is the input of the current time step, h_t is the hidden state and $\hat{x}_{t+1}$ is the predicted value for the next time step.

One of the inherent problems of a standard recurrent neural network is vanishing gradients, originating from backpropagating through the network the same weights were used to calculate the hidden states backwards. If the weights related to the hidden state is a small value the repeated multiplication results in a vanishing gradient, or if the weights have a large value an exploding gradient. One solution to the vanishing gradient problem is the Gated Recurrent Unit(GRU), which is a version of a recurrent neural

network that applies a reset gate and update gate to decide how much of the new input should be included in the new hidden state, inversely how much of the old hidden state should be forgotten. In other words how important is the new hidden state compared to the previous input for calculating future hidden states and output. DreamerV2 applies a GRU for its world model, it works as a memory mechanism where the hidden state represents the memory of the model.

2.2 Reinforcement learning

This chapter introduces the different concepts related to reinforcement learning. In the introduction chapter 2.2.1 reinforcement learning as a concept is to be defined, afterwards the agent-environment interaction is presented in chapter 2.2.2. A central challenge in reinforcement learning is the balance between exploration and exploitation of the environment which is discussed in chapter 2.2.3. Further on in chapter 2.2.4 we discuss *Rewards* and the sum of *Rewards*, *Return*, which is the value to be maximized in reinforcement learning. The Bellman equation being one of the central elements of many reinforcement learning algorithms is elaborated on in 2.2.5. The Bellman equation decomposes the value function into two parts, the immediate reward in addition to the discounted future rewards. It is central to all value based reinforcement learning algorithms, where the policy makes choices based on the expected return, the value function, that is the immediate reward plus discounted future rewards. Chapter 2.2.6 describes the mathematical framework for decision making in reinforcement learning, the Markov Decision Process. On-policy and off-policy is two categories which reinforcement learning algorithms can be divided into, they have both strengths and weaknesses which we will look into in 2.2.7. How models of the environment in model-based reinforcement learning work and how these models can improve the reinforcement learning agent in chapter 2.2.8. The part of the DreamerV2 agent responsible for decision making is an actor critic, this is a policy gradient method which will be explained in chapter 2.2.9. Finally a look in to world models as a central concept of the DreamerV2 agent in chapter 2.2.10.

2.2.1 Introduction

Reinforcement learning is a goal-directed learning approach, with the goal to maximize a numerical reward signal. In this case the agent is not told what decisions to make, but has to act in a way that maximize the reward. This *knowledge* is learned from the trial

and error experience. The agent is interacting with its environment to experience what actions receives a positive feedback and what actions receives a negative feedback. The delayed reward is a central concept in reinforcement learning where previous decisions giving a negative reward at that specific moment in time might position the agent for a larger positive reward later.

2.2.2 Agent-environment interaction

It is common to consider reinforcement learning to be the third subcategory of machine learning with the other two categories being supervised and unsupervised learning. Where the latter two categories are based on data with and without labels, reinforcement learning is based on an agent-environment setting. In this case an agent chooses an Action to interact with its environment, then this environment the rewards the agent accordingly. The agent then traverses the environment to learn from the experience the it receives to map situations, often called states, to actions. The notation for an individual State in the set of all states S, is s, while the notation for actions it is a for a single action in the set of all possible actions A. A policy is applied to choose action a from all possible actions A, and a specific policy is often noted π and it represents the rules for how to choose an action.

An illustration, a greedy policy chooses the best action it can at each time step, while a random policy takes random actions for each time step. One commonly used policy is the ϵ-greedy policy which chooses the best possible action with the probability of 1 - ϵ and a random action otherwise. The reason to take a random action is to encourage exploration.

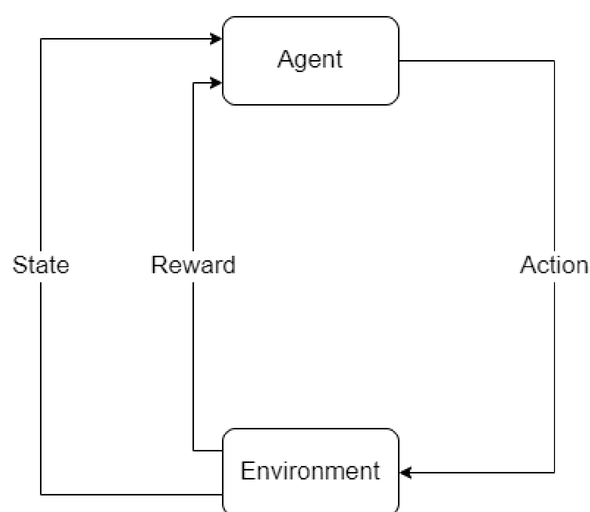


Figure 2.7: Agent-environment interaction

2.2.3 Exploration and exploitation

One of the fundamental problems of reinforcement learning is the exploration and exploitation of the environment. This dilemma origins from the trade off between looking for more solutions, by exploring, hence future return might be better or to keep exploiting the current best solution, but at the cost of no improvement in the future.

In some occasions it is better to choose a lower exploration rate when the learning time is long to try to find the best combinations and not influence your information by bad, exploring, decisions. In case of a shorter learning period more exploration will return an acceptable solution faster. To combine both exploration and exploitation by applying a decreasing exploration rate during learning, the agent will then explore more of the environment in earlier episodes and when the agent gets smarter the amount of bad decisions from exploring declines. If the environment can change dynamically, and the agent has already learned a good policy, this policy is no longer a good policy. While having low exploration rate slow learning takes place, because the agent will do bad decisions based on an outdated policy. The agent does not explore new decisions, thus learning a good policy will take longer than with a higher exploration rate.

2.2.4 Rewards and Return

Reward is the notation for the positive and negative feedback given from the environment to the agent. Noted r for a reward and R_t for a reward at time t, describing the immediate feedback from the environment to the agent. As explained in the introduction chapter 2.2.1 in Reinforcement learning, the immediate reward is not of great interest, rather the sum of rewards, defined as Return. An example of this fact could be an agent preforming stock trading where earned money is the reward and it needs to spend money, negative reward, to earn more money in the future, i.e. a larger total sum of rewards.

Another example of why return is more important than immediate reward could be a game of chess where the agent decides to sacrifice a piece on purpose to earn a better position, in such a way that the probability of victory is higher. The agent's goal is thus to maximize the sum of all the rewards in an episode, not just acquiring the immediate reward. The expected sum of rewards from a given state, can be expressed by the term defined as return. Return noted with G, and is defined as $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}...$ where t is the time step and γ is a discount factor for future rewards. The discount factor γ is used to take into account uncertainty of the environment and in some reward models the discount factor is making sure the return is not a infinite sum.

2.2.5 Bellman equations

The value function in reinforcement learning is used to evaluate how much return is the agent expected to receive from the remaining episode given a state. Many reinforcement learning algorithms build upon making decisions based using the value function. The ϵ-greedy policy makes the decision of which action to take with the highest value from the value function given the possible next states, $v(s_{t+1})$, with a probability of $1-\epsilon$, and otherwise a random action to explore new trajectories. The value function is defined as:

$$v(s) = E(G_t|S_t = s) \tag{2.10}$$

Where $v(s)$ is the notation for the value function given state s and given the state s the value function returns the expected *return*, G_t for the time step t.

In the introduction of the chapter 2.2, it is stated that the Bellman equation decomposes the value function into two parts, the immediate reward plus the discounted future rewards, as can expressed here in formula 2.11. The Bellman equation then simplifies the future rewards by defining the value function as a recursive function that makes the computation of the value function easier since *return* is defined as $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}...$ the Bellman equation can be extended to:

$$\begin{aligned}
v(s) &= E(G_t|S_t = s) \\
v(s) &= E(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3}...|S_t = s) \\
v(s) &= E(R_{t+1} + \gamma G_{t+1}|S_t = s) \\
v(s) &= E(R_{t+1} + \gamma v(S_{t+1})|S_t = s)
\end{aligned} \tag{2.11}$$

By this extension the value function for a state S_t can be calculated as the reward from that transition in addition to the value function for the next state S_{t+1}.

The Bellman equation is a powerful tool by which the agent can evaluate how much return to be expected from a given state. In this thesis the Bellman equation is applied in the DreamerV2 agent to update the critic's value function in the actor critic part of the DreamerV2.

2.2.6 Markov Decision Process

Markov Decision Process(MDP) is a mathematical framework for modeling decision making, which can be used to provide a formal description of an reinforcement learning environment. A central element in reinforcement learning is describing a problem as a Markov

decision process. In the previous chapters there has been some use of the framework already with the description of return, reward, states and actions, and these concepts can be expressed as functions for rewards and transitions.

- $R_a(s', s)$ represents the reward from state s to new state s' given action a.
- $P_a(s', s)$ represents the transition probability from state s to new state s' give action a.

The Markov decision process is an extension of the well known Markov chain, where the Markov chain describes the transition probabilities between states. The Markov decision process also includes actions that affect these transition probabilities.

A central concept in Markov decision processes is the Markov property which is:

$$p(S_{t+1}|S_t) = p(S_{t+1}|S_t, S_{t-1}, S_{t-2}...S_0) \tag{2.12}$$

"The future is independent of the past given the present"[31]. The state s needs to contain the information needed such that the future is independent of the past. As the example mentioned in chapter 2.1.4 if a reinforcement learning algorithm is only given the down hole pressure of the current time step, e.g. $3.5e7\ Pa$, the probability of the next time step's pressure is dependent on the past, because the probabilities of the next time step would be different if the pressure was increasing or decreasing at this time step.

This leads into an important issue in our case, physics does not inherently satisfy the Markov property. The authors of DreamerV2 claims that with the GRU utilised in the world model the latent state that the Actor Critic evaluates should satisfy the Markov Property[14].

2.2.7 On-policy and Off-policy

When the value function is learning the value of the behaviour policy, it is defined as an on-policy method. Off-policy learning takes place when the value function is learning a different policy from the behaviour policy. The value function learns the value of the target policy and the behaviour policy is utilised to make decisions. One advantage for using off-policy is learning an optimal policy while acting according to an non-optimal policy. In many reinforcement learning problems, the agent can explore more extensively while learning a solution that is optimal. When applying on-policy would either limit the

exploration while learning, or limit the exploitation of the return for the agent. Applying a policy like ϵ-greedy this limitation could be mitigated with a decreasing ϵ value as discussed in chapter 2.2.3, but with the shortcomings when dealing with a dynamically changing environment as discussed in chapter 2.2.3. Off-policy would however, ensure larger exploration throughout the learning since it could continue to sample all possible actions while the target policy is not suffering from these decisions, since the behaviour policy is deciding the actions and updating the value function with decisions from the target policy. A well known example of an off-policy method is Q-learning where the target policy is a greedy policy, always choosing the decision to exploit the highest value of return. The behaviour policy, often ϵ-greedy, but it could also be a policy following human decisions or random decisions. The value function thus learned with a greedy policy and would have higher return values than a non greedy policy, but the agent is acting with another policy to ensure exploration and finding new possible trajectories.

A weakness of off-policy learning methods are that they can be require more gradient updates compared to on-policy methods. On-policy methods learns faster by narrowing down the states it searches by not exploring to the same degree and focusing on better return compared to off-policy algorithms may explore more states that does not yield better return, depending on the behaviour policy. Off-policy algorithms can however be more sample efficient by reusing data. Either data is collected by the agent or from an outside source such as a human expert or another conventional controller. It can be an advantage especially in environments that requires a lot of compute or longer time between time steps, since the reuse of data reduces the amount of samples needed to train the agent, even though the amount of gradient updates might be more than with on-policy.

2.2.8 Model-based learning

Model-based algorithms rely on planning rather than primarily relying on learning from experience. In reinforcement learning a model is an approximation of an Markov Decision Process, where the model approximates the state transitions $R_a(s', s)$ and $P_a(s', s)$ to model the environment. The term planning in reinforcement learning is used for a process that with the use of a model can either improve a policy or produce a policy for interaction with the environment the model is based upon. Some examples of this is dynamic programming and heuristic search.

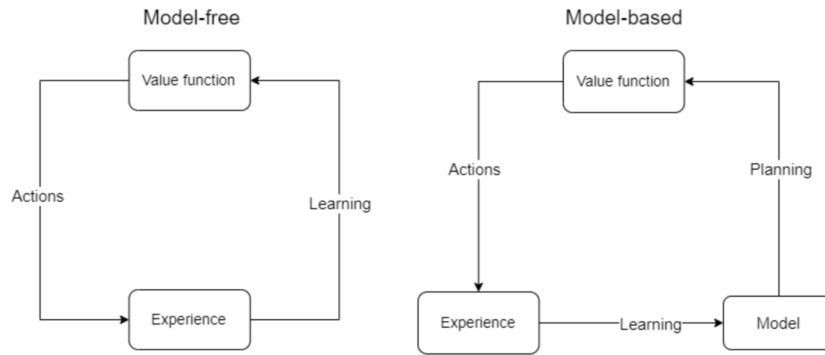


Figure 2.8: Model-free learning and model-based learning

Inspired by DeepMind x UCL RL Lecture Series - Planning & models

The value function can be learned with experience both from the model and experience directly from the environment. Experience in this context imply the interaction with the environment and the consequences from it. This method is used very effectively in the class of Dyna-Q algorithms, see figure 2.9. By using both experience from the model and the environment less biases is experienced in the value function, avoiding biases from the model.

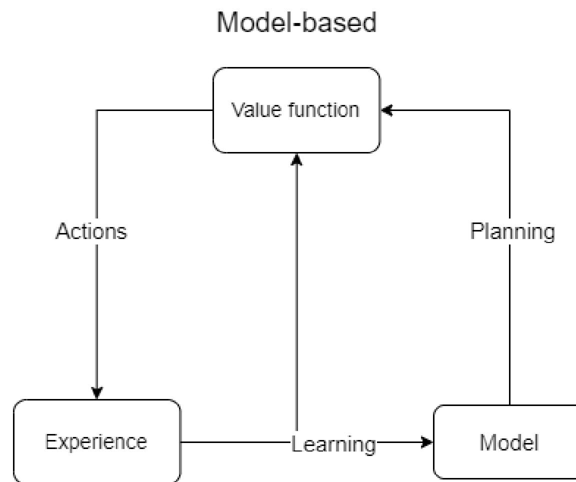


Figure 2.9: Learning from both experience and model

Inspired by DeepMind x UCL RL Lecture Series - Planning & models

The main strengths of model-based learning is the efficient methods of learning a model, usually traditional supervised or unsupervised machine learning, which does not suffer from reinforcement learning's problem of sparse signals and sample inefficiency. Using the learned model to produce or improve a policy is cheaper than model-free learning reduces the the amount of reward signals directly from the environment and can

learn more efficiently from the model. The main drawback of model-based reinforcement learning is that there is now two sources of approximation, the value function and the model. A previous obstacle in model-based reinforcement learning is that these methods has not preformed well on environments where the rules were unknown to the model or the environment was too complex to model easily[29]. In contrast environments such as chess and go saw success with AlphaGo[32] and AlphaZero[33], both developed by DeepMind, which used Monte Carlo Tree Search to plan ahead in combination with Q-learning to out preform humans in both chess and go, but the model knew the rules of the game. MuZero[29], also from DeepMind, managed to preform at a similar or better level than previous reinforcement learning algorithms without knowing the rules of the environment with a learned model.

The DreamerV2 used in this thesis has a world model trained from online and offline data retrieved from the simulator, and the uses a latent space representation of the world model to create internal simulations for training the Actor Critic for decision making, similar to the model-based diagram 2.8.

2.2.9 Policy Gradients

Policy gradient methods the process revolves around mapping directly state to an action, $f(s) = a$, instead of mapping state to a value as explained previously in this thesis, with the value function $v(s) = E(G_t|S_t = s)$. This mapping is done by learning a policy π that is differentiable with respect to its parameters θ when producing a probability for an action given the parameters and a state s. Policy gradient methods is expressed as:

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t, \theta_t = \theta\}[34] \tag{2.13}$$

For the probability of action a taken at time t with state s at time t and parameter θ.

One advantage of policy gradients is that the optimal policy for a given environment may be a stochastic policy with arbitrary probabilities. Such as environments where the optimal policy is not deterministic, but stochastic. In this case most value function based reinforcement learning method would never learn the optimal policy, whereas a policy gradient method could learn an arbitrary probability. If the agent samples from a distribution of the possible actions it can learn stochastic optimal policies. An example of this could be bluffing in poker, where different states has different probabilities of how good the different actions are, and doing a deterministic strategy would be easy for an opponent to exploit.

Another advantage of policy gradient methods is that the rate of exploration is tied to the probability distribution for the actions produced by the learned policy π. This advantage lets the agent explore when the possible actions is evaluated as equally good, and when it learns that the best action is a given state it will decrease the exploration. Since the agent is continuously updating the parameters, θ, and if the environment changes the distribution will change and encourage more exploration again. This dynamic exploration rate gives an elegant way of managing the exploration and exploitation problem where the exploration will decrease when the agent finds trajectories with higher *return*, but if the environment were to change and the previously good trajectories turned bad the exploration rate would not be locked at a low rate such as with an $\epsilon - greedy$ policy with a decreasing ϵ.

A final advantage of policy gradients is stronger convergence guarantees due to the smooth changes of the policy's probability distribution. A slight change in an $\epsilon - greedy$ policy could lead to a complete shift in action probabilities since it is greedy with a probability of $1 - \epsilon$, on the other hand a policy gradient gives smooth slight changes therefore policy gradients have a stronger convergence guarantee[34](chapter 13.2).

The DreamerV2 utilises an actor critic to preform decision making. This is a policy gradient method which uses an actor to produce a probability distribution for the action given the state, and a critic which predicts the value of a given state. The actor and critic is used in conjunction to train an agent. One update of the actor can be formulated as such:

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w))\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \tag{2.14}$$

Where $\hat{v}$ is the approximate value function produced by the critic, and π is the parameterized policy produced by the actor. $\nabla\pi$ is the gradient of π with respect to the weight parameters θ. α is the step size of the gradient ascent and A_t and S_t is the action and state at time t. Two advantages of using a critic in conjunction with the actor are:

- Being able to solve continuous problems with only the next time step needed for an update instead of an entire episode.
- Significantly reducing variance with the critic update, and thus speeding up training.[34] Chapter 13.4

Actor critic is a policy gradient method that combines the best of approximating a policy with the actor, using it's advantages of learning stochastic policies and stronger convergence guarantees due to the smooth changes of the policy's probability distribution and estimating the value function with the critic to significantly reduce both the variance and training time.

2.2.10 World Models

The World Models as described in "World Models"[11] takes inspiration from the human mind to develop models for reinforcement learning where the human mind creates a mental model based on abstractions to simplify the world we experience around us. "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." (Forrester, 1971)[8] To mimic the human mind the David Ha and Jürgen Schmidhuber created an agent that uses internal abstract representations for decision making.

The world model works differently from standard model-based reinforcement learning whereby the state representation that the agent receives is not the state representation from the environment, but a latent state representation from the world model. This representation can be viewed as an abstraction of the input state combined with the memory of previous states as it is the encoded state from the encoder, input, and the hidden state, memory, from the recurrent neural network.

The agent model is composed of three parts: The Vision model, the Memory model and the Controller model. The Vision model which is a variational autoencoder that encodes the current observation from the environment to a latent vector z and after that decodes it back to the same image. The Memory model is a Mixture Density Network - Recurrent Neural Network and its role is to predict probability distributions of future latent z vectors as $P(z_{t+1}|a_t, h_t, z_t)$. The Controller model is a single linear layer that maps the latent state representation of the latent vector z_t from the Vision model and the hidden state h_t from the Memory model to action a_t, see figure 2.10. The reason for using a simple controller is the fact that the reward signal in reinforcement learning can be sparse, and many agent may need large amounts of training, thus with a smaller model this effort will be faster and require less resources. The goal is the to create a representation of the environment in the latent state representation of sufficient quality, enabling the simple single layer controller to be able to control the agent to a acceptable degree.

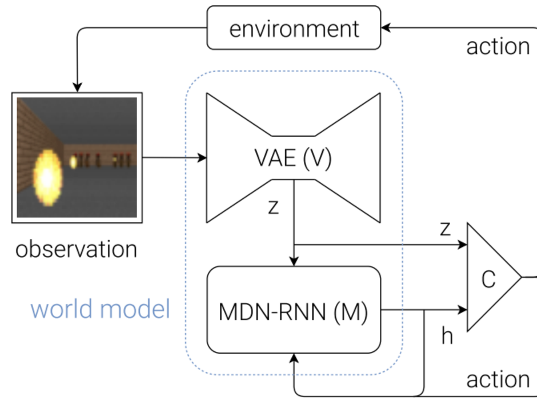


Figure 2.10: Vision, memory and controller example from world models paper[11]

Picture directly from world models paper.

Listing 2.1: Pseudocode of episode rollout from world models paper[11]

```
def rollout(controller):
        """
        env, rnn, vae are
        global variables
        """
        obs = env.reset()
        h = rnn.initial_state()
        done = False
        cumulative_reward = 0
        while not done:
                z = vae.encode(obs)
                a = controller.action([z, h])
                obs, reward, done = env.step(a)
                cumulative_reward += reward
                h = rnn.forward([a, z, h])
        return cumulative_reward
```

The world model is trained by rolling out 10 000 episodes with random actions of the CarRacing-v0[19] environment to initially train the world model. Since the controller is limited in terms of number of parameters. Ha and Schmidhuber trained it by using a evolutionary algorithm, CMA-ES, to optimize it according to the reward. The training can then be viewed as these four sequential steps:

1. First collect 10 000 rollouts
2. Train Vision model to represent latent vector z
3. Train Memory model to predict $z_t + 1$
4. Train Controller model to maximise *return*

The authors claim new state of the art results in this environment with this method. One problem with this approach is that the agent might find a way to cheat the simulation in the world model for an optimal strategy that does not translate into the real environment.

World models offers two major strengths compared to more traditional reinforcement learning. Faster training and higher quality results by utilising the better state representation with the latent state representation. In the "World Models" paper they achieve new state of the art results with just a simple single layer mapping in the CarRacing environment. High efficiency is achieved by the more compact and information rich state representation with the encoding and memory from the recurrent neural network. The other advantage is using dreams to learn, in this paper the authors try to learn only inside dreams of the agent. Where the world model simulates a dream environment and the controller learns a policy inside this environment. This is how the DreamerV2 actor critic is trained and has an advantage since the agent can train in parallel on the GPU. This is faster than the interaction with the environment and decreases training time significantly.

The DreamerV2 agent utilizes a similar world model as described in this paper, but trains its controller, an actor critic, entirely internally simulated in the world model. This agent has instead of the Gaussian distribution often used in variational autoencoder, a world model where the recurrent neural network is a GRU and the latent state representation is 32 categorical vectors.

Recently the University of Geneva released IRIS[20] a world model based reinforcement learning agent that uses transformers[35] to model the environment to great extent. With this model they managed state of the art performance beating MuZero[29] with at the Atari 100k benchmark. Exhibiting that world models is sample efficient and able to learn dynamic environments without prior knowledge of the rules. A use case for this can be real life environments or simulated environment which can not be sped up significantly to produce the needed hundreds of millions of time steps for training a reinforcement learning agent.

2.3 Online and offline learning

In this chapter we aim to describe important characteristics of Online learning as well as those of Offline learning by elaborating on their differences and how they are applied

in the field of machine learning. Further we will introduce the methods used later in this thesis, such as a combination of offline and online data. Exploring the training of the world model initially with offline data, and afterwards batching the data from the environment to facilitate easier asynchronous training of the world model. Finally, we will compare the use of online learning during deployment versus not learning during deployment.

Online learning is learning from a continuous stream of data, at the moment the data arrives, and in the case of online learning, the weight parameters is changing continuously with the arrival of new data, where a shift in the distribution of data will lead to a shift in the learned parameters or even the algorithm "forgets" older information.

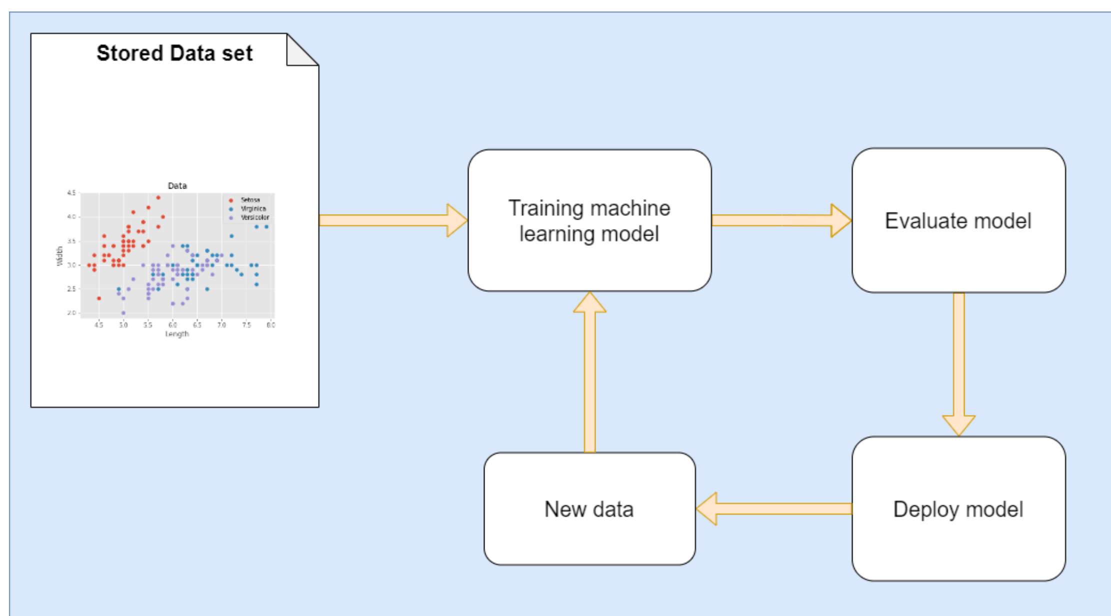


Figure 2.11: Example of online learning process

During offline learning precollected data is utilised to learn an algorithm, and it is considered to be the most commonly used method of training in machine learning today.

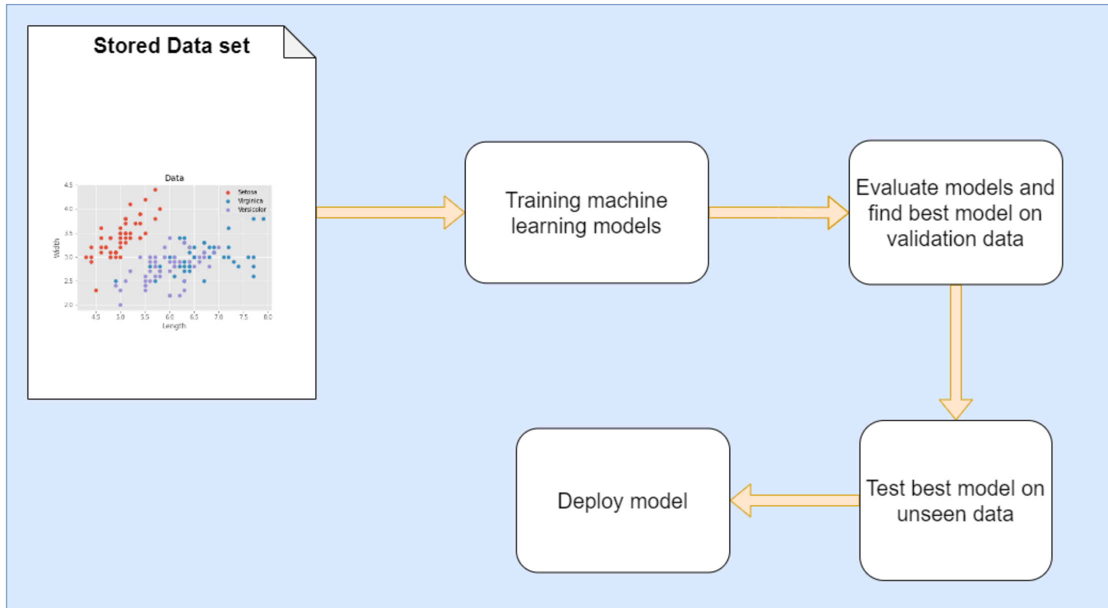


Figure 2.12: Example of offline learning process

Both offline and online learning methods are applied in reinforcement learning while online learning historically has been used the most frequently. The traditional way to train an agent in reinforcement learning can be defined in three steps:

1. The agent interacts with the environment.
2. Receives a reward, and updates its value function.
3. Repeat until termination.

This process creates a continuous stream of data and updates to the agent. In this case the improvement of the agent results in better episode trajectories, which facilitates faster learning,

However, the use of offline learning has been explored in reinforcement learning, where agents utilise pre-stored trajectories to update the parameters or learn a model of the environment. There are a number of reasons to use either learning method, and one of the strengths related to offline learning is computational efficiency. The ability to learn without waiting for the arrival of new data, and executing training in batches significantly improves the speed to learn an offline machine learning model compared to learning an online machine learning model.

To develop a a simulation to interact with an agent can be both difficult and time consuming and therefore expensive, since it will need large efforts in developing an accurate simulation of the task that is to be learned. No need for simulation of the training environment is an additional strength of offline learning in context of reinforcement learning. Errors or mistakes in the execution of the simulation can lead to learning the wrong parameters thus causing faulty results when using the trained machine learning model, or reinforcement learning agent, in a real world setting. When only training from a data set collected from the real world, some of these issues are avoided, even though the data might be misrepresenting the reality similarly to a simulation.

One solution to avoid such misrepresentations is to train a machine learning model from real world interaction, where a machine learns from real world input and consequences of the machine's results and decisions directly. A considerable drawback of this method is to develop a machine, at a large cost, and running the risk of damaging the machine during operations and finally having additional repair costs. Developing a machine often ends up being more expensive and time-consuming, and therefore less feasible than a simulation or a stored data set. A valuable feature of offline data is that it can be reused both by the machine learning model to improve its performance, and also by other machine learning models to enable a reduced cost of developing new machine learning models.

The possibility to collect data of wanted behaviour, similar to human behaviour, serves as a good starting point of both inexpensive data and acceptable quality results to facilitate faster learning. Rather than the agent learning good behaviour by trial and error.

The main argument for choosing online learning, even though it is more expensive, harder to implement, more complex and computationally more inefficient, is the ability to adapt and evolve with the environment that can help future proof a system. The spam filter in an email system, can adapt without manual updates to counteract new spam emails, and similarly a stock trading robot can adapt to new patterns in the stock market.

By collecting data in batches and perform regular updates to the system, it is still possible to benefit from an evolving system that adapts to changes, but still maintaining a lot of the benefits of having a batched of data to learn from. When combining online and offline learning some benefits from both methods of learning are obtained, without all the detriments of both offline and online learning.

Finally, when comparing offline and online learning we find that they are not mutually exclusive. It is possible to create a system that trains agent, or a machine learning model, on pre-collected data and afterwards deploy it as an online system that learns from streamed data as it arrives.

In reinforcement learning often the method is to train in an online fashion, but when evaluating a reinforcement learning agent it is done without doing updates to the agent. In this thesis the difference between updating the algorithm's weight parameters and not updating it to see its ability to adapt, including a comparison of online learning during evaluation, and without updating the weight parameters during evaluation is explored.

2.4 OpenLab

The information about drilling in this chapter is mainly gathered from the *Store Norske Leksikon* article on petroleum drilling[10]. Modern deep well drilling operations utilize advanced techniques to improve operational efficiency, such as new drill bit variants to improve the rate of penetration (ROP) and new logging techniques to maneuver the drill string to the desired position. Data logging is applied to measure the properties of the rock while drilling, a technique that is called Logging While Drilling (LWD).

To drill a deep well for CO_2 or water injection purposes, or oil and natural gas extraction a rotating drill bit is mounted on the top end of the drill string, and it consists of a number of of steel pipes that is joined together. The drill string is rotated by the top drive electric motor, located on the drilling rig. To reduce the drill bit's friction against the walls of the well and at the same time balance the down hole pressure, drilling mud is injected into the bore hole.

This pressure balance is important to control in order to avoid that the down hole pressure is too low and resulting in a "drill kick" which is happening when formation fluids leak out into the well, and in worst case scenario leading to a blowout. If the down hole pressure is too high, the drill mud is pressed from the bore hole and into the rock formation and is eventually lost. In the case of oil drilling this can destroy the production capabilities of the reservoir. The down hole pressure is controlled by continuously monitoring of the volume of the fluids that is in circulation. If there is a high level in the mud tank on-board the drilling rig, this indicates that formation fluids has come into the drilling mud. A low level in the mud tank indicates leakage of drilling mud from the bore hole to the formations.

OpenLab[28] is a high fidelity physics based simulator for web enabled deep well drilling simulation developed by Drilling & Well Modeling group at NORCE in collaboration with the University of Stavanger. The purpose is to provide a simulation environment within drilling and well technology for research, education and innovation. OpenLab's core technology is WeMod that is widely recognized as a thoroughly tested well model as it has widely been used by both industry and by universities to simulate drilling operation based on the well model they developed.

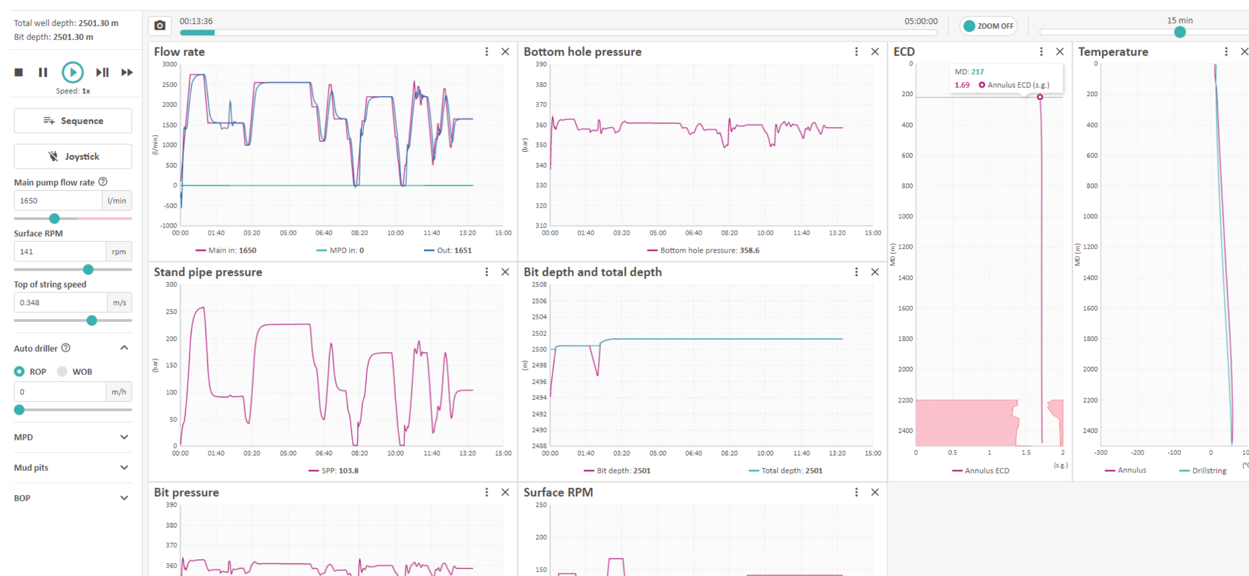


Figure 2.13: OpenLab simulator GUI.

Historically model-based reinforcement learning has struggled with dynamic environments where the rules of the environment were unknown to the model. In this thesis a modern approach using the complex modeling capabilities of the DreamerV2 agent's world model to attempt to model the environment of OpenLab, where the DreamerV2 will be both evaluated on its modeling capabilities and its control capabilities.

Chapter 3

Methodology and Experiment

In this chapter the experiment and its setup is presented, beginning with a detailed description of the DreamerV2, which is the reinforcement learning agent used in the thesis. To determine if there are correlations between different parameters and the adjustable set points parameters in the OpenLab simulator, and most importantly the relation with the down hole pressure, a data analysis of different simulated well trajectories is described. Next, both the implementation of OpenLab as a reinforcement learning environment, and the configuration of both OpenLab and the DreamerV2 agent is specified. Afterwards an explanation of how the DreamerV2 agent was trained is given. Finally the design of the evaluation of the world model's ability to create a representative model of the environment, the DreamerV2 agent's ability to control the flow rate in OpenLab, including an online evaluation of continuous updates, to explore if online learning with the DeamerV2 can be an appropriate method to learn new dynamics in the case of new data of previously unknown events.

3.1 DreamerV2

In this chapter The DreamerV2 agent proposed in "Mastering Atari with Discrete World models"[14] is introduced and discussed. A short analysis of the computation needed to train the DreamerV2 agent compared to other modern agents. The different components of the agent such as the world model and the actor critic as well as a description of how the training of these components are preformed. Furthermore we are looking at some of the results from the original DreamerV2 paper and what benchmarks the DreamerV2 agent

preformed well on together with a discussion of the reasons behind these results. Finally an overview of PyDreamer[24], an implementation of the DreamerV2 in PyTorch[25] developed by Jurgis Pasukonis with focus on the slight differences between DreamerV2 and PyDreamer.

DreamerV2 is an agent that utilises a world model to generalize past experience and imagine new trajectories. It was developed by Hafner, Lillicrap, Norouzi and Ba[14]. The agent's world model encodes a state representation to a discrete latent space. The encoding of the state representation to a latent space enables simulation of 2500 latent trajectories in parallel[14]. This efficiency is essential to achieve state of the art performance on single GPU agents, and it is the first single GPU agent with world modelling that achieves human-level performance [14] on the 200M Atari benchmark.

3.1.1 Computational analysis

The DreamerV2 model's efficiency derives from its ability to simulate imagination Markov Decision Process to train quickly. The original implementation of the DreamerV2[14] used less than 10 days on a single nvidia V100 GPU, to train 200 million frames on the Atari benchmark. The PyDreamer implementation achieved similar results on an nvidia T4 in 10 days. This is because the DreamerV2's ability to simulate up to 2500 trajectories in parallel, which enables the DreamerV2 to train quickly without needing to compute the environment. This enables the DreamerV2 to train at a similar speed as IQN, another single GPU state of the art model-free reinforcement learning agent. In this thesis the ability to predict trajectories to train the actor critic is a major advantage, because the OpenLab simulator only produces 5 time steps per second, resulting in a long training time with a model-free agent. The actual parameter size of the implementation is 22 million parameters which is about half of MuZero's 40 million, however the simpler model of DreamerV2, training in an imagined Markov Decision Process and not utilising as complex planning as MuZero makes it possible to utilise a consumer grade GPU. For comparison in the "Mastering Atari with Discrete World models" paper the authors claim that it will take 80 days to train MuZero on a single GPU for an Atari agent[14].

Algorithm	Trainable Parameters	Atari Frames	Accelerator Days
DreamerV2	22M	200M	10
SimPLe	74M	4M	40
MuZero	40M	20B	80
MuZero Reanalyze	40M	200M	80

Table 3.1: Computation comparison of model-based reinforcement learning agents, values from DreamerV2 paper [14].

3.1.2 Components of DreamerV2

World model

The World model is composed of several different components that work together to create a model of the environment. These components are the Recurrent model, Representation model, Transition predictor, Image predictor, Reward predictor and Discount predictor. In the following chapter each part of this definition 3.1 will be discussed to give insight in how the DreamerV2 world model functions and how it differs from the original World Models paper[11].

$$\begin{aligned}
&Recurrent\ model: && h_t = f_\phi(h_{t-1}, z_{t-1}, a_{t-1}) \\
&Representation\ model: && z_t \sim q_\phi(z_t | h_t, x_t) \\
&Transition\ predictor: && \hat{z}_t \sim p_\phi(\hat{z}_t | h_t) \\
&Image\ predictor: && \hat{x}_t \sim p_\phi(\hat{x}_t | h_t, z_t) \\
&Reward\ predictor: && \hat{r}_t \sim p_\phi(\hat{r}_t | h_t, z_t) \\
&Discount\ predictor: && \hat{\gamma}_t \sim p_\phi(\hat{\gamma}_t | h_t, z_t)
\end{aligned} \tag{3.1}$$

The Representation model and Image predictor is what would correspond to the encoder decoder component from the variational autoencoder in the original World models paper[11] as discussed in chapter 2.2.10. The Representation model, which can be viewed as the encoder in a variational autoencoder, samples a representation z_t from a probability conditioned on the input state x_t and the hidden state h_t. The posterior state z_t which is used to make decisions with the actor critic. This latent state is comprised of a vector with several categorical variables to give a state representation better than a Gaussian distribution could as can be seen in figure 3.1.

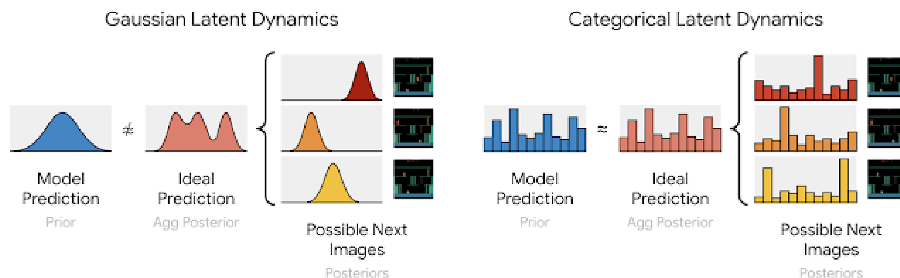


Figure 3.1: Gaussian and categorical distributions in latent space, image from DreamerV2 blog[15]

DreamerV2 Blog

In the DreamerV2 paper the categorical latent variables outperformed traditional Gaussian latent variables in 42 Atari games, tied in 5 and were worse in 8. The authors propose four hypotheses that may be the cause of this improvement[14]:

- The categorical prior can better fit the aggregate posterior than a Gaussian prior could match a mixture of Gaussion posteriors. Figure 3.1 for illustration from Hafner's blog about the DreamerV2[15].
- Since the categorical variables are sparse this could help with generalization of the latent state space.[14]
- The use of straight-through gradients in combination with the categorical variables might be easier, and avoid vanishing or exploding gradients. Since "the straight-through gradient estimator ignores a term that would otherwise scale the gradient."(Hafner, Lillicrap, Norouzi, Ba)[14]
- The fourth hypothesis considers that the categorical variables might have better inductive bias in the Atari games. This is not relevant in this thesis since it is neither image input or the same mechanics as a video game.

The Image predictor, which can be viewed as the decoder in a variational autoencoder, predicts the original input state from the hidden state h_t and posterior state z_t with the mean of a diagonal Gaussian likelihood with unit variance as the output distribution. This is a vital part of the loss function to learn high quality representations of the input state x_t. It can also be used to visualize the states from a imagined trajectory.

In the case of simulated well drilling the input state is not an image, but a vector describing the current state values of both the drill system and the physical parameters at the bottom of the well. In this case linear layers will be used instead of convolutions.

The Recurrent model calculates the hidden state h_t from the previous hidden state h_{t-1}, action a_{t-1} and posterior state z_{t-1}. The hidden state representation h_t is vital to every other prediction of the world model as can be seen in the definition 3.1. The hidden state is calculated from the previous hidden and latent state, and the action from that time step a_t, this is done with the use of a Gated Recurrent Unit[4], which uses a reset gate and update gate avoid the problem of vanishing and exploding gradients. The extra calculation of the forget gate is defined in equation 3.2 in a general setting.

$$\begin{aligned} z_t &= \sigma(W_z\, x_t + U_z\, h_{(t-1)} + b_z) \\ r_t &= \sigma(W_r\, x_t + U_r\, h_{(t-1)} + b_r) \\ \hat{h}_t &= tanh(W_h\, x_t + r_t * U_h\, h_{(t-1)} + b_z) \\ h_t &= z_t * h_{(t-1)} + (1 - z_t) * \hat{h}_t \end{aligned} \tag{3.2}$$

Where r is the reset gate which allows the hidden state to drop any irrelevant information and create a more compact hidden state[4]. The update gate z controls how much of the previous hidden state h_{t-1} is included in the new hidden state h_t[4]. The candidate hidden state $\hat{h}_t$ is dependent on the activation of the reset gate, which allows the possibility of ignoring new irrelevant information. W and U is weight matrices and b is the bias term, and finally $*$ is the element-wise product, σ is the sigmoid function and $tanh$ is the hyperbolic tangent function.

The transition predictor tries to predict z_t only given the hidden state h_t, and then predicts the prior state $\hat{z}_t$, this is used to imagine trajectories without getting input for each state, the use of these imagined trajectories for training the actor critic will be elaborated on in 3.1.3.

The Reward and Discount predictor predicts the reward and discount from a given hidden state and latent state. The reward prediction $\hat{r}_t$ is sampled from the output of the Reward predictor which is a univariate Gaussian distribution with unit variance. The Discount predictor outputs a Bernoulli likelihood to sample the discount prediction for each time step. The discount $\hat{\gamma}$ used in the DreamerV2 is predicted the likelihood of an episode ending when learning behaviours from model predictions.

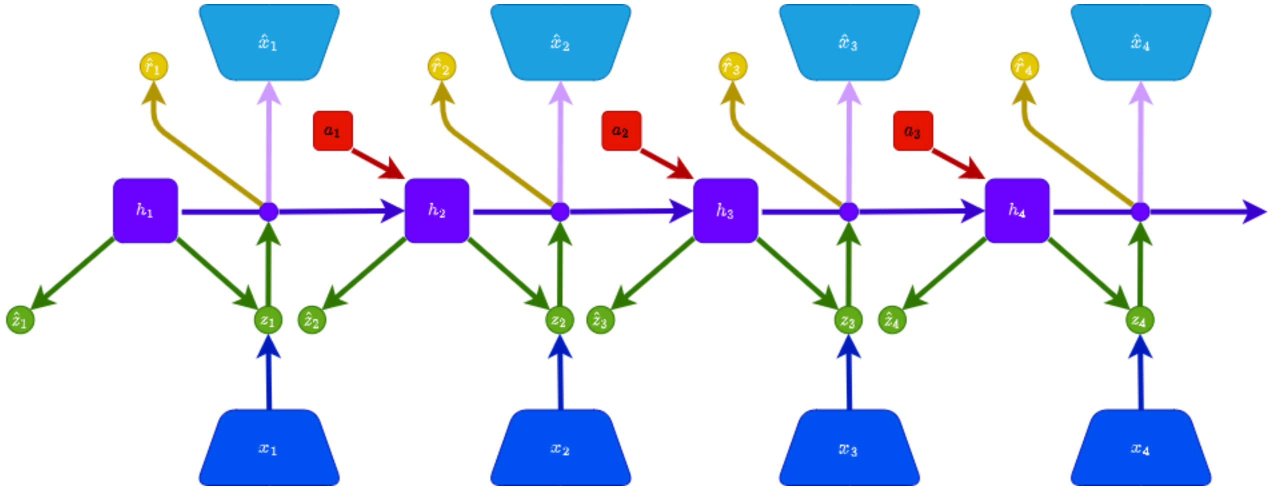


Figure 3.2: Training the world model

Inspired by DreamerV2 illustrations[14]

The figure 3.2 is an illustration of how the world model is trained. Where given an input x_t, the world model uses the representation model to sample z_t with the history h_t from the recurrent model and the input x_t, illustrated with dark blue symbols for the representation model, purple symbols for recurrent model and green symbols for the

posterior z_t. Similarly the transition model predicts the prior $\hat{z}_t$ given h_t, also noted in green. The recurrent model, in purple symbols, calculates h_{t-1} from the action a_{t-1}, represented with red symbols and the previous hidden state h_{t-1}. Finally the image predictor and reward predictor is on the top row in light blue and yellow predicting the input $\hat{x}_t$ and reward $\hat{r}_t$. The discount predictor is not included in the illustration.

Actor Critic

As previously mentioned the DreamerV2 utilises an actor critic as the decision making component in the agent. How an actor critic works in general was introduced in chapter 2.2.9. The next paragraphs details the actor critic utilised in the DreamerV2 agent.

The actor is a artificial neural network composed of linear layers with 1 million weight parameters that gives a stochastic action a_t that aims to maximize future rewards, since receiving its input from the world model's recurrent neural network the input state is Markovian[14]. Therefore both the actor and the critic does not need to condition previous states. The loss function of the actor utilises the Reinforce gradients [37], which has high variance, leveraged with straight-through gradients [2].

Listing 3.1: Straight-through gradients[2], pseudocode from DreamerV2 paper[14]

```
1 # Draw sample for latent state, with no gradient because gradients are removed with
     sampling
2 sample = one_hot(draw(logits))
3 # Store the wanted gradient
4 probabilities = softmax(logits)
5 # Add the wanted gradient to the sample, but remove the probabilities
6 sample = sample + probabilites - stop_grad(probabilites)
```

The critic is seeking to predict expected return, given a state, and it is evaluating the value of being in this state by estimating the value functions, i.e. how much *return* is expected in a given state s. The DreamerV2 critic uses TD(λ) to learn to predict the expected return from a given state. The actor critic is then defined as:

$$\begin{aligned} &\text{Actor: } \hat{a}_t \sim p_\psi(\hat{a}_t|\hat{z}_t) \\ &\text{Critic: } v_\mathcal{E}(\hat{z}_t) \approx E_{p_\phi p_\psi}\Big[\sum_{\tau \geq t} \hat{\gamma}^{\tau - t} - \hat{r}_\tau\Big] \end{aligned} \tag{3.3}$$

3.1.3 Training

The training of the world model is executed on a data set of past experiences. This data set contains the previous trajectories of the DreamerV2 and for each data point in these trajectories there is the state observation x_t, the action a_t, rewards r_t and discount factors γ_t. The discount factors are the fixed hyper parameter given in the configuration of the model, and in this thesis $\gamma = 0.995$ is chosen, but if γ_t is the terminal step the discount factor is equal to 0 to indicate the end of and episode. From these stored trajectories it is created batches of 50 truncated trajectories with a length 50 time steps. Each batch used for updating the world model has the dimensions of $(50, 50, (x, a, r, \gamma))$. These truncated trajectories are uniformly randomly sampled with the start index in the interval $[0, episodelength - 50]$ to never exceed the episode length in the truncated trajectory.

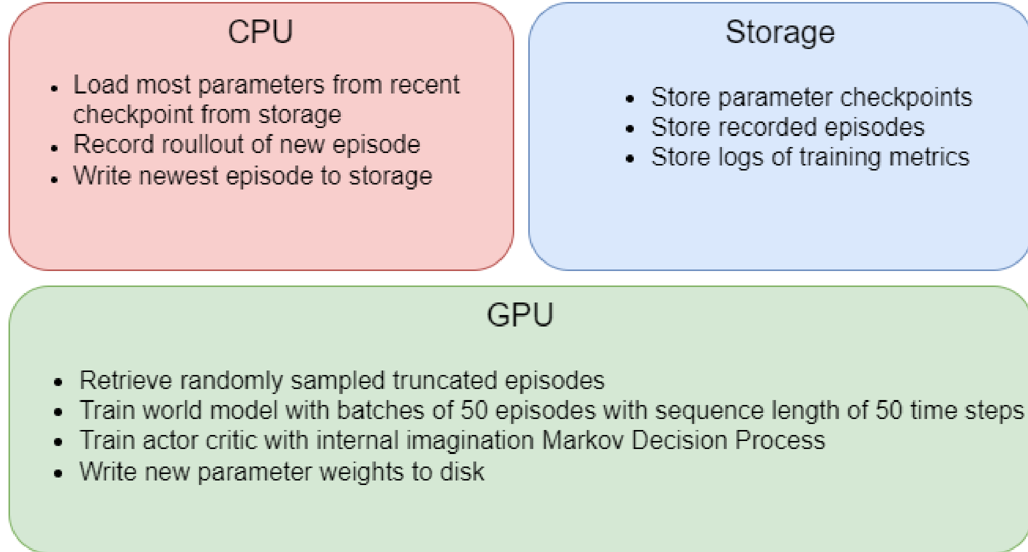


Figure 3.3: Training the DreamerV2 agent

$$\mathcal{L}(\phi) = E_{q_\phi(z_{1:T}|a_{1:T}, x_{1:T})}\Big[\sum_{t=1}^{T} - ln\, p_\phi(x_t|h_t, z_t) - ln\, p_\phi(r_t|h_t, z_t) - ln\, p_\phi(\gamma_t|h_t, z_t) + \beta\, KL[q_\phi(z_t|h_t, x_t)||p_\phi(\hat{z}_t|h_t)]\Big] \tag{3.4}$$

The first three components of loss function of the world model is:

- The log loss of the image predictor $-ln\, p_\phi(x_t|h_t, z_t)$, in this thesis it is not an image, but a vector observation.
- The reward log loss $-ln\, p_\phi(r_t|h_t, z_t)$.
- The discount log loss $-ln\, p_\phi(\gamma_t|h_t, z_t)$.

The final part of the loss function is the KL balancing loss of z and $\hat{z}$. The Kullback-Leibler divergence is a measurement of the difference between two probability distributions. More specifically in machine learning the KL divergence is used to calculate the amount of information lost when using the prior distribution, $\hat{z}$, compared to the posterior distribution, z. KL balancing is a variation utilised in the DreamerV2's loss function by applying a different learning rate to the approximate posterior distribution and prior distribution. The reasoning for this different learning rate is to avoid regularizing the approximate posterior to a poorly trained prior, and the learning rate for the prior is then $\alpha = 0.8$ and $1 - \alpha$ for the approximate posterior. This prioritizes the prior learning instead of posterior entropy which leads to the prior better approximating the aggregate posterior[14].

Since the transition predictor, see 3.1, can predict the next latent state representation $\hat{z}_{t+1}$ given the hidden state h_{t+1}, and the recurrent model calculates the hidden state given the previous hidden state, latent state representation and action, h_t, z_t, a_t, then it is possible to simulate an "imagination Markov Decision Process" inside the world model without the need for encoding new state representations or waiting for the environment to update. The DreamerV2 is able to simulate 2500 latent trajectories on a single GPU to facilitate faster training inside the world model compared to traditional training of agent. The output of the transition predictor is a sequence of latent states at length of the imagination horizon $H = 15$. Afterwards the reward predictor $p_\phi(\hat{r}_t|\hat{z}_t)$ predicts the reward for each imagined time step, and the discount predictor $p_\phi(\hat{\gamma}_t|\hat{z}_t)$ predicts a sequence of discounts, if the final time step is a terminal one. See figure 3.4 for an illustration.

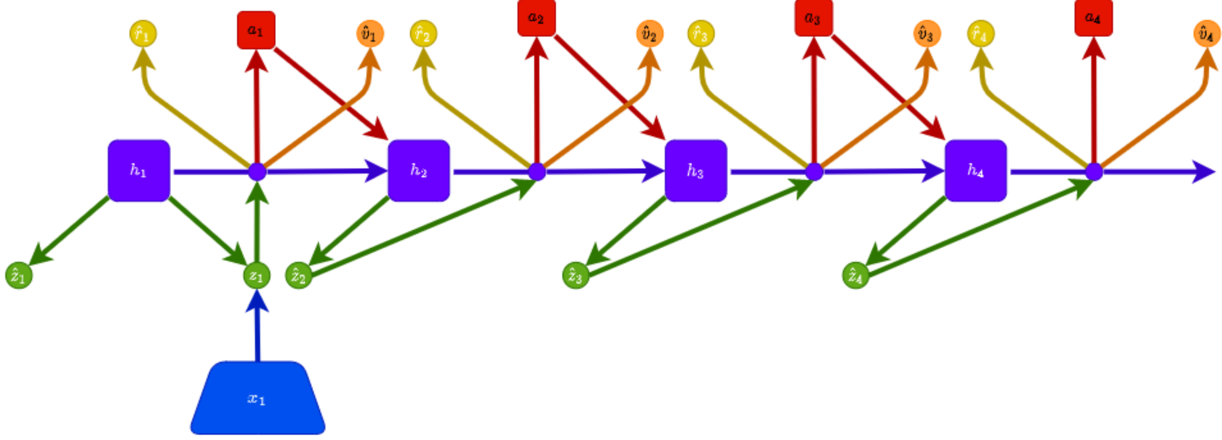


Figure 3.4: Training the Actor Critic with imgaination Markov Decision Process

Inspired by DreamerV2 illustrations[14]

At each time step of the imagination Markov Decision Process the critic is approximating the value of the expected return, that is used to update the actor with the loss function of the actor critic.

The loss function of the critic is defined as:

$$\mathcal{L}(\mathcal{E}) \doteq E_{p_\phi p_\psi}\Big[\sum_{t=1}^{H-1} \frac{1}{2}\Big(v_\mathcal{E}(\hat{z}_t) - sg(V_t^\lambda)\Big)^2\Big] \tag{3.5}$$

The loss function is optimized with the Adam optimizer[18], with respect to the critic parameters $\mathcal{E}$, where sg is the stop gradient function. V_t^λ is the λ-target[34], the λ-target is the weighted average of the n-step returns. In the "Mastering Atari with Discrete World Models"[14] the λ value is set to 0.95 to encourage long horizon targets. The λ target can be defined as:

$$V_t^\lambda \doteq \hat{r}_t + \hat{\gamma}_t \begin{cases} (1-\lambda)v_\mathcal{E}(\hat{z}_{t+1}) + \lambda V_{t+1}^\lambda, & \text{if } t < H \\ v_\mathcal{E}(\hat{z}_H), & \text{if } t = H \end{cases} \tag{3.6}$$

$$\mathcal{L}(\mathcal{E}) \doteq E_{p_\phi p_\psi}\Big[\sum_{t=1}^{H-1}(-\rho ln\, p_\psi(\hat{a}_t|\hat{z}_t)\ sg(V_t^\lambda - v_\mathcal{E}(\hat{z}_t))\ - (1-\rho)V_t^\lambda - \eta H[a|\hat{z}_t])\Big] \tag{3.7}$$

The actor loss comprises of three main components. The Reinforce gradient[37] $(-\rho ln\, p_\psi(\hat{a}_t|\hat{z}_t)\ sg(V_t^\lambda - v_\mathcal{E}(\hat{z}_t))$, the straight-through gradients[2] backpropagated through

the dynamics of the sampled actions and state sequences, and finally the entropy regularizer[22] $\eta H[a|\hat{z}_t]$.

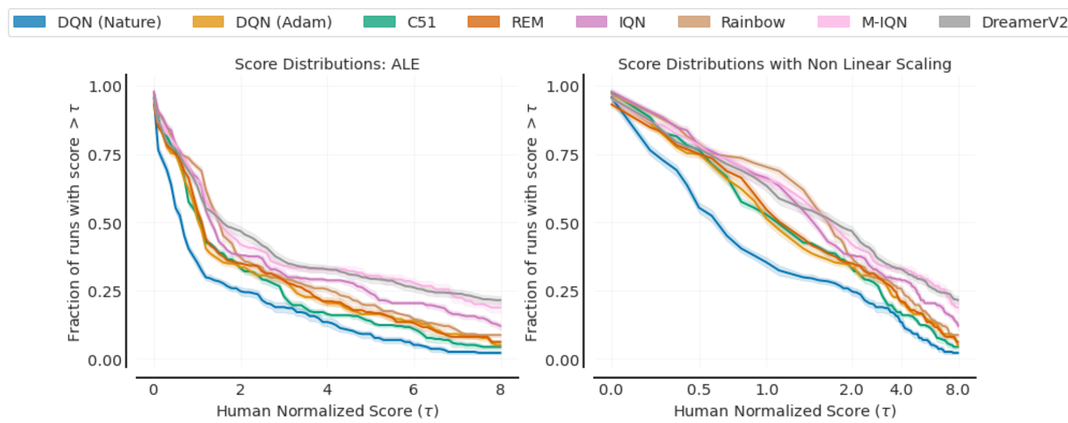
The Reinforce gradient is weighted with the ρ hyperparameter against the dynamics backpropagation straight through gradient $1 - \rho$. In the episodic Atari tasks the Reinforce gradient preformed better and was weighted higher, while in continuous tasks the dynamics backpropagation straight through gradient preformed better in the benchmarks evaluated in the DreamerV2 paper[14]. The η hyperparameter weights the entropy regularizer as proposed in the "Asynchronous Methods for Deep Learning"[22] to help actor critic methods to avoid favoring actions or action sequences, and therefore avoiding local maxima. While actor critic method offers an elegant solution on how to balance exploration and exploitation they can often over exploit good actions and get stuck in local maxima. Therefore the entropy regularizer utilised to encourage exploration of the environment.

3.1.4 Results

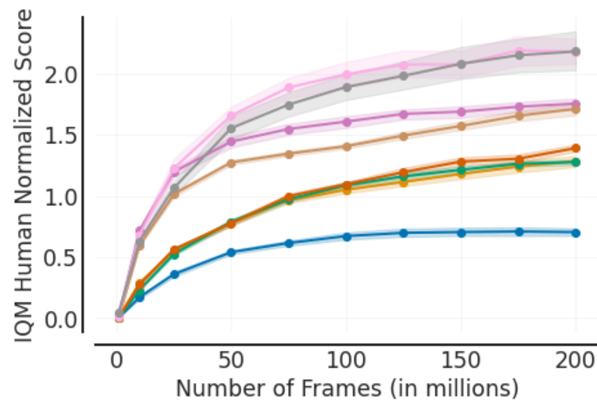
From the paper "Mastering Atari with Discrete World models"[14] the Atari 200M benchmarks resulted in the DreamerV2 claiming better performance than state of the art single GPU agents like Rainbow[16], IQN[5] and SimPLe[17]. In the paper "Deep Reinforcement Learning at the Edge of the Statistical Precipice"[1] the authors propose new evaluation methods for reinforcement learning benchmarks such as the Arcade Learning Environment. The motivation for this paper was to propose a more standardized evaluation of new methods on established benchmarks to try to remove some of the statistical uncertainty of having few training runs to evaluate since modern reinforcement learning agent has become computationally expensive to train. In this paper m-IQN [36] preforms similarly to the DreamerV2 agent, and these two agents preform better than other agents in most tasks, where only single GPU agents were evaluated, confirming the performance of the DreamerV2 agent. However the DreamerV2 agent has a higher variance in these result compared to the other agents evaluated.

Strengths and weaknesses

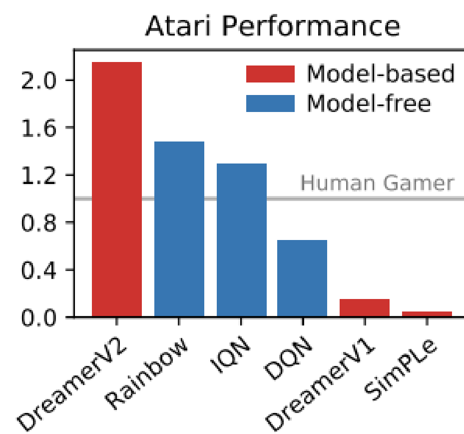
The benchmarks which were evaluated in the DreamerV2 paper[14] the DreamerV2 scores high results on visually complex tasks were there are a large amount of relevant information in the pixels. An interesting example of this is the game James Bond, where



(a) Performance profile on Atari 200M benchmark.[1]



(b) Sample efficiency in Atari 200M benchmark.[1]



(c) Relative performance of DreamerV2 on Atari 200M.[14]

Figure 3.5: Figure a and b is the results reported from the "Deep Reinforcement Learning at the Edge of the Statistical Precipice"[1] paper, and figure c is the performance reported from "Mastering Atari with Discrete World models"[14].

the DreamerV2 manages to exploit the game to a score over 10 times better than IQN, slightly worse than the human world record, but over 130 times higher score than the human gamer mark.

The DreamerV2 gets low results on Video Pinball, the authors, Hafner, Lillicrap, Norouzi, Ba[14] , hypothesize it is because of all the visual noise in the game and the important information, the ball, is only one pixel. So the World model struggles to create a meaningful latent representation.

The most important strength of the DreamerV2 for this thesis project is the combination of high level performance, low required computation, and the advantages of the world model. The world model training directory can be prefilled with offline data to facilitate faster training of the world model, to enable the world model to learn a good representation of the environment without waiting on data from the simulator. Another advantage is the imagination Markov Decision Process which gives the ability to train the actor critic independent of the speed of the environment, since it is trained by the imagined trajectories as illustrated in figure 3.4. Since OpenLab is a time consuming simulation, that we do not have the access to significantly speed up, and is being communicated over the internet, the DreamerV2's world model's training and sample efficiency is a major advantage.

3.1.5 PyDreamer

PyDreamer[24] is an implementation of the DreamerV2 agent developed by Jurgis Pasukonis and is utilised in this thesis. The main reasons for using the PyDreamer instead of the original DreamerV2 was the use of PyTorch and the readability of the PyDreamer project structure.

There are two main differences in the PyDreamer implementation compared to the original DreamerV2, that is the PyDreamer has a Advantage Actor Critic (A2C) instead of the standard actor critic and instead of TD-λ the critic calculates the Generalized Advantage Estimation. The benchmark results of both implementation are comparable.

A2C is a variant of the Asynchronous Advantage Actor Critic (A3C) from the paper "Asynchronous methods for deep reinforcement learning"[22]. The critic in A2C approximates the advantage formula instead of the value function, in the PyDreamer implementation the critic approximates the generalized advantage estimation.

Generalized Advantage Estimation[30] is an alternative function for calculating the value of a state. The advantage function for policy π given state s_t and action a_t, is defined as:

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{3.8}$$

Where $Q(s, a)$ is the action-value function and $V(s, a)$ is the value function. The advantage function calculates how much better it is to preform action a_t compared to the other possible actions in this state. With policy gradients the result has a lower variance than other value based functions because the gradient step should increase the most in the direction of the better than average actions in a state.

The general advantage estimation formula is then an attempt to approximate the actual advantage function with the lowest possible variance. The result from the Generalized Advantage Estimation paper is equation 3.9 where the γ discount is used as a parameter to reduce variance and the λ is used as a decay rate similarly to the $TD(\lambda)$. This gives an estimation of the advantage function instead of the value function or action value function, which compared to the other two functions the advantage function introduces tolerable bias and lower variance.

$$\begin{aligned} \hat{A}_t^{GAE(\gamma,\lambda)} &= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \\ \delta_t^V &= -V(s_t) + r_t + \gamma V(s_{t+1}) \end{aligned} \tag{3.9}$$

The DreamerV2 agent was chosen for this thesis because of the good results on previous benchmarks, the possibility of training on a singe GPU and the interesting concept of world models. To explore if world models are capable in a dynamic physics based environment, such as the high-fidelity drilling simulator OpenLab.

3.2 Data analysis

In this section investigations are carried out on what parameters from the OpenLab simulator that correlates with the down hole pressure, to identify what parameters is affecting this pressure and how it can be expected to behave during a given simulation. The information acquired from the analysis is used to evaluate how to design the reward function of the environment and how to prepare the data for efficient training of the agent.

3.2.1 Analysis of collected simulations

The different simulations used to prefill the training directory will be presented in this chapter. The simulations will be visualized, the different minimum and maximum values will be extracted to create a function to normalize future values. These precollected simulations with random set points for the adjustable OpenLab parameters will be referred to as the precollected data set and the state of a single time step is then defined as the parameters from the precollected data set excluding the flow rate parameter. The set points are parameters that is possible to change in the OpenLab simulator, for this thesis flow rate, top of string velocity and surface RPM are the set point values which were looked into.

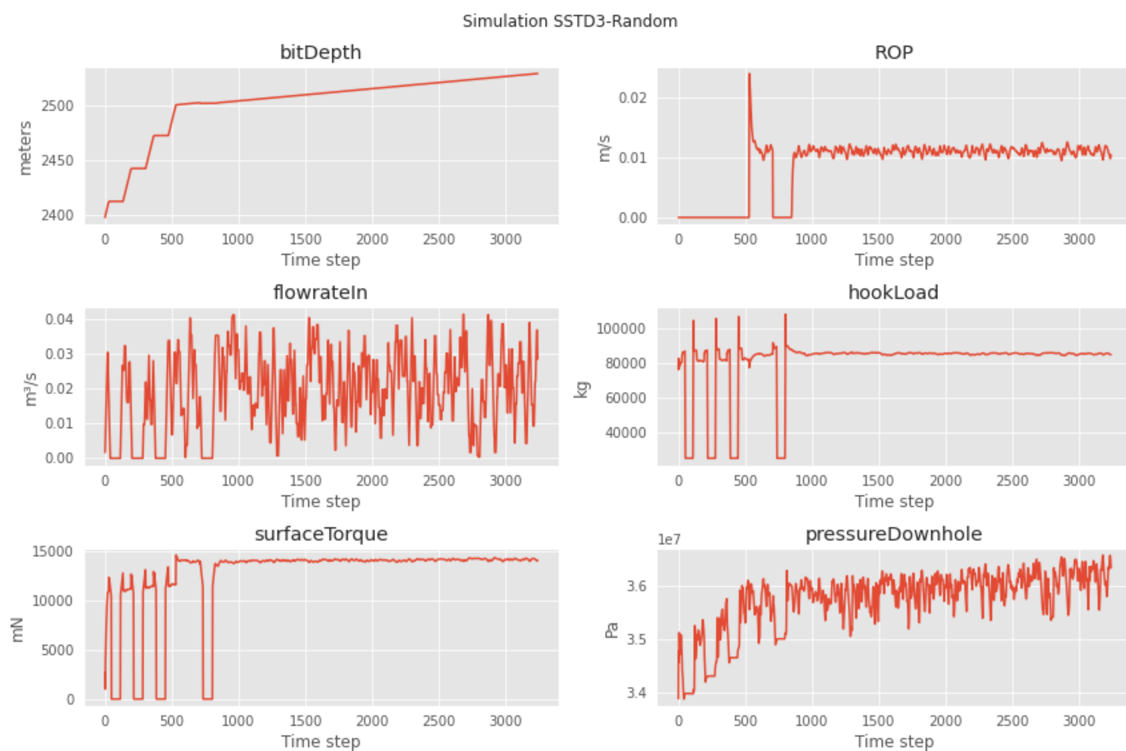


Figure 3.6: Simulation(1) with random set points parameters.

In the first precollected simulation shown in figure 3.6, the flow rate oscillate randomly with large amplitudes, which can be interpreted to have a large correlation with the pressure down hole, see figure 3.7. This observation indicates of how the flow rate clearly impacts the down hole pressure, thus increases the probability of keeping the pressure within a desired range. The reason for the random set point values are to encourage

different transitions for the world model to learn, as performed in previous training of world models[14, 11].

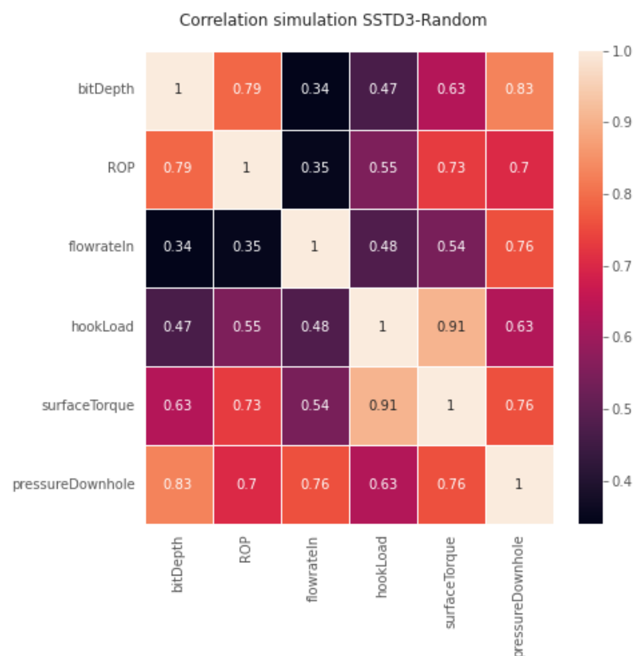


Figure 3.7: Correlation matrix of simulation 3.6

In the correlation matrix 3.7 the attribute with the highest correlation with the pressure is bit depth. This is an observation that is reoccurring in all simulations in this thesis, and is caused by the fact that all the wells used in the precollected data set and training simulation were inclined wells. In other well configurations this value might not correlate in the same manner. Another high correlation in the same simulation is the surface torque, but this correlation do not appear in every simulation, see figure 3.11, which will be discussed in the paragraph about figure 3.10.

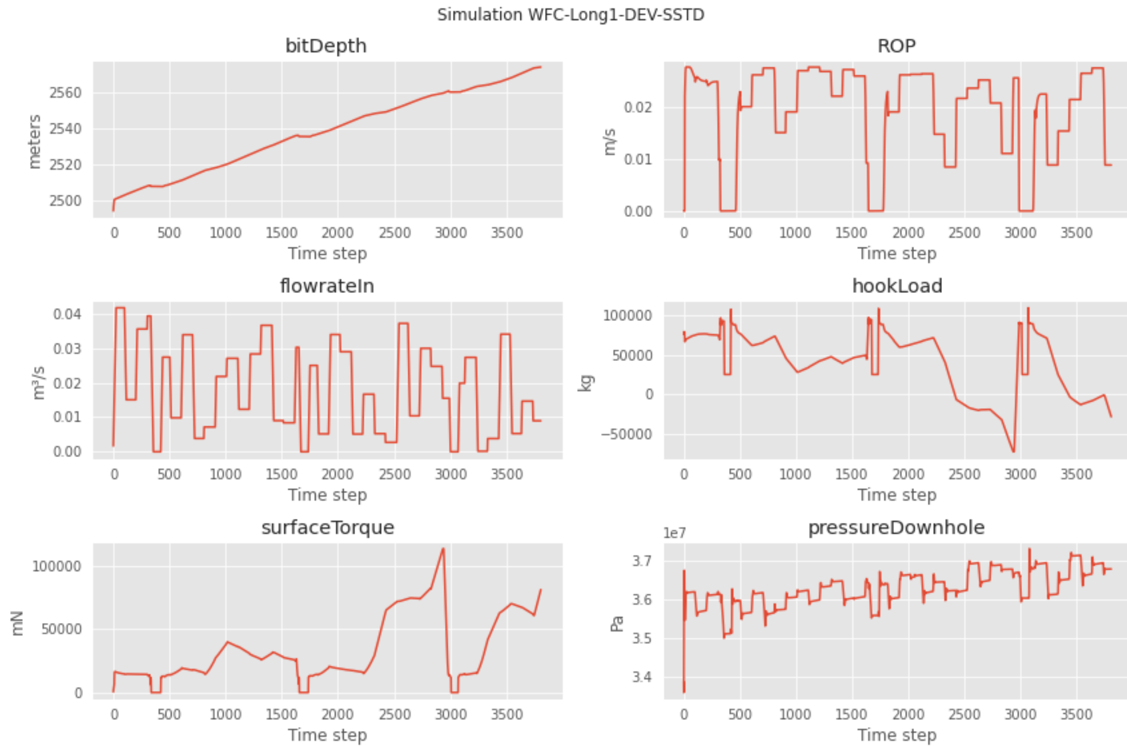


Figure 3.8: Simulation(2) data with random values for the set point parameters, with intervals between change.

In the second simulation from the precollected data set, figure 3.8, the set point values are again chosen at random, but with a longer period of time between each new value, about 100 time steps between each change. In this simulation all correlations with the down hole pressure are reduced significantly, with the exception of the bit depth. The reason for this reduction in correlation could origin from different parameters canceling each other out in such a way that they nullify the correlation with pressure. For example a decrease in flow rate causes a decrease in down hole pressure and an increase in surface torque resulting in increased down hole pressure, but the actual down hole pressure stays the same.

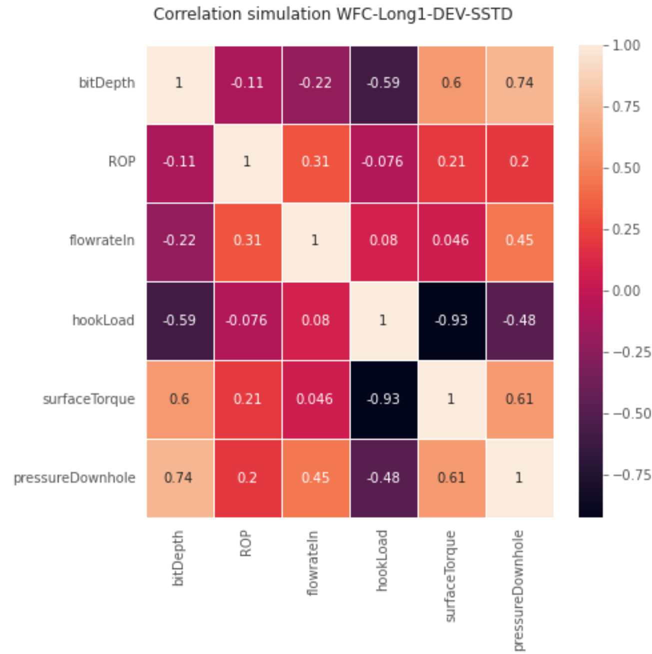


Figure 3.9: Correlation matrix of simulation 3.8

The final precollected simulation, figure 3.10, is similar looking to the previous one, figure 3.8, but this simulation 3.10 has more time steps, over a longer duration, and therefore goes to the depth of 2600m, compared to 2560m in the previous simulation. However the correlation between the different parameters is different. The larger surface torque and a higher Hook load of almost double the negative weight constitutes an important difference from the previous precollected simulations, shown in figures 3.6 and 3.8. The hook load is the total force pulling down on the hook[23]. The correlation matrix in figure 3.11 is from the final precollected simulation results in surface torque having no correlation with the down hole pressure. In this simulation only the flow rate and the Rate of penetration(ROP) has any significant correlation, in addition to the bit depth as mentioned in the previous paragraph.

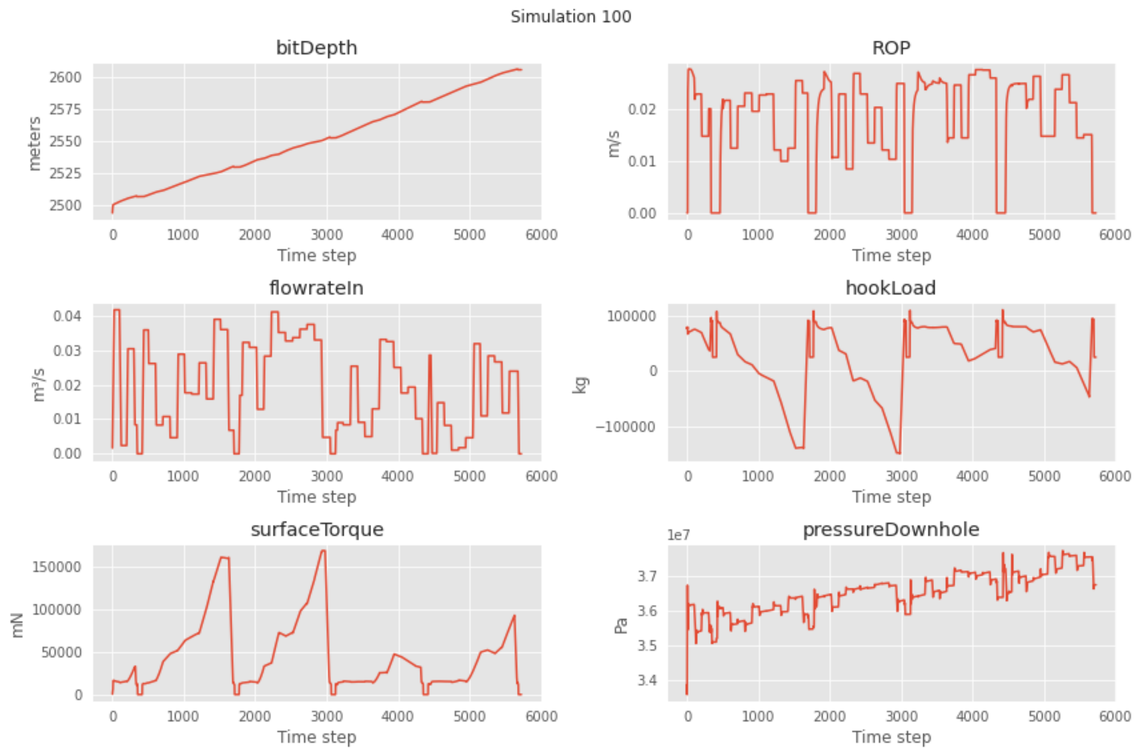


Figure 3.10: Simulation(3) with random set points for flow rate, with intervals between change

A possible reason for the low correlation between down hole pressure and adjustable set points parameters might be even though a set point parameter can cause a change in down hole pressure other parameters of situations can also affect the pressure. Looking at a single simulation will not illustrate the actual correlations. Further in chapter 3.2.2 it will be looked upon when all but one adjustable set points parameters is kept constant, and the down hole pressure values is always varying as the pressure is also influenced by other factors. For example if the flow rate has a constant value, while other parameters are affecting the pressure there would be low correlation between the flow rate and down hole pressure. In the precollected simulation 3.10 the surface torque is varying, and in a larger range than previously, yet it has close to no correlation, however this observation contradicts the previous two observations and causing uncertainty of how strongly the surface torque actually is correlated to the pressure.

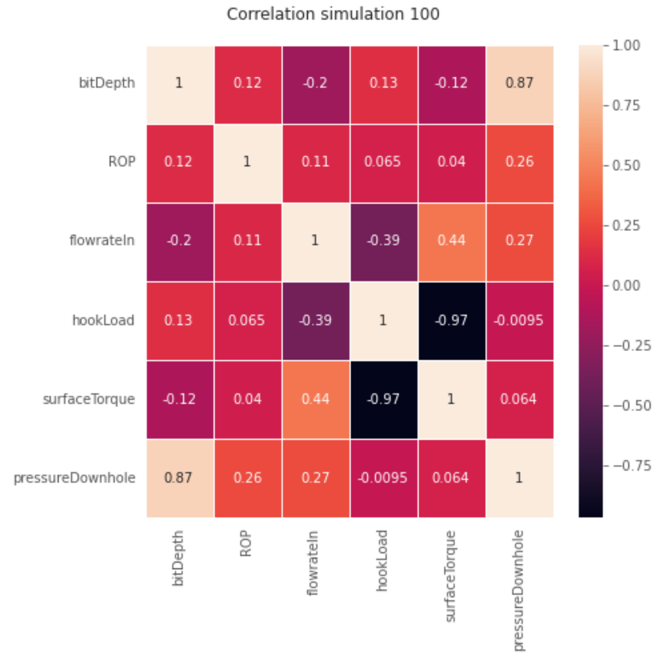


Figure 3.11: Correlation matrix of simulation 3.10

In these previously shown simulations, visualised in figures 3.6 to 3.11, the flow rate and rate of penetration retains at least some correlation with the down hole pressure. This establishes the basis for the possible control of the down hole pressure given control of the flow rate. There are other adjustable parameters in the OpenLab simulator which are not present in the precollected data set, but some of these will be included in the next analysis of short simulations where one of the adjustable parameters is adjusted, and the other parameters are kept constant. Another reason for performing this additional analysis is to remove some of the noise present in the precollected data set and only evaluate the relationship between chosen parameters and the down hole pressure.

The precollected data set and future data would need to be normalized in the range $[0, 1]$ to represent the values with half-precision floating-point number. A further advantage of this exercise is to avoid one feature from having a large effect on the result due to a larger value by measurement unit. For example the down hole pressure input values in the precollected data of ca $3.6e7\ Pa$ and the rate of penetration (ROP) with values around $0.015 m/s$. In the next paragraph it is shown the method for how the data was normalized to achieve a range of values between 0 and 1 $[0, 1]$, while taking into account the uncertainty of future simulations with possible larger or smaller values.

Data preparation

Based on the minimum and maximum values for each value in the offline data set from OpenLab a normalisation step for use in the environment was created. We added a 5% margin to each end of the normalisation range to account for the possibility of encountering higher or lower values than what existed in the offline data set. The normalisation formula can be expressed as seen in 3.10 for each value category c:

$$value_c = (value_c - min_c * 0.95)/(max_c * 1.05 - min_c * 0.95) \tag{3.10}$$

The figure 3.12 is the normalized version of the same simulation as 3.8.

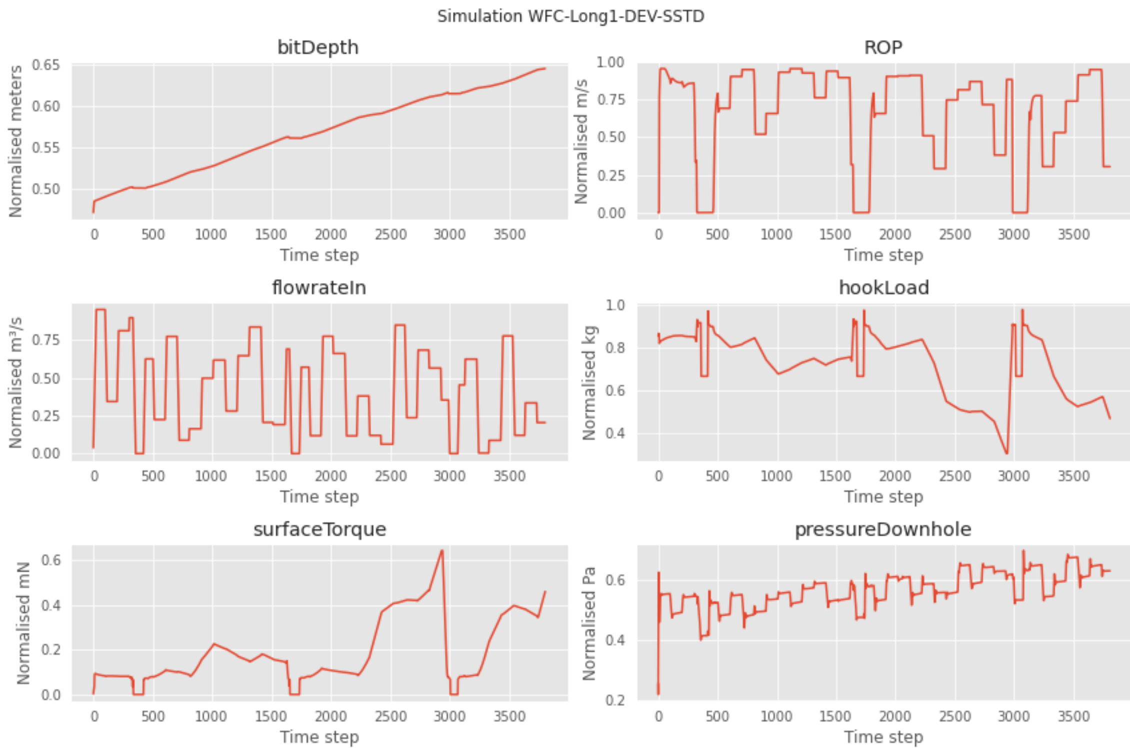


Figure 3.12: Simulation(2) with normalized values

The bit depth was the attribute with the highest correlation to the down hole pressure through all simulations. Although not always the case, in this thesis there are only inclined wells and therefore this correlation can be used to create a function for artificial pressure goals for the agent to reach during training.

	Min	**Max**
Bit depth	2278.257m	2736.899m
ROP	0.0m/s	0.0291667m/s
Flow rate	$0.0m^3/s$	$0.0437500m^3/s$
Hook load	-157039.280kg	116157.530kg
Surface torque	0.0mN	176471.646mN
Down hole pressure	31907764.0Pa	39623791.2Pa

Table 3.2: Minimum and maximum values for normalising future data

Initially a linear regression analysis to formulate expected pressure at a given depth was carried out. With the regression seen in figure 3.13 the expected increase in down hole pressure for each meter is 16200 Pa. With this information a function to create artificial pressure goals that increase following the depth and actual down hole pressure can be formulated, thus avoiding giving unreachable pressure goals to the agent.

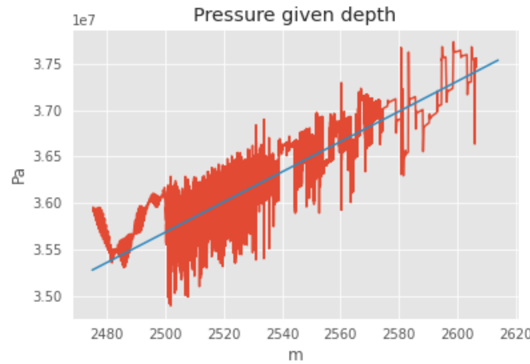


Figure 3.13: Linear regression of down hole pressure given bit depth

3.2.2 Analysis of isolated parameters

In this chapter short simulations where only one of the adjustable set point parameters from OpenLab is changed will be analysed, to see how it affects the down hole pressure. The goal is to analyse how much flow rate, surface RPM and top of string velocity affects the down hole pressure when all other parameters is kept constant. One time step is one second in real time and in all the visualisations the different values are normalized.

The first simulation only the flow rate was adjusted in a simple pattern, where the flow rate stayed at $0l/min$ for 25 time steps to let the simulation stabilise. While the other set point parameters were set to:

- Top of string velocity: 0.7
- Surface RPM: 120 RPM

After the simulation stabilised the flow rate was changed to $2500l/min$, then after around 100 time steps the flow rate was reduced back to $0l/min$. This is illustrated in figure 3.14.

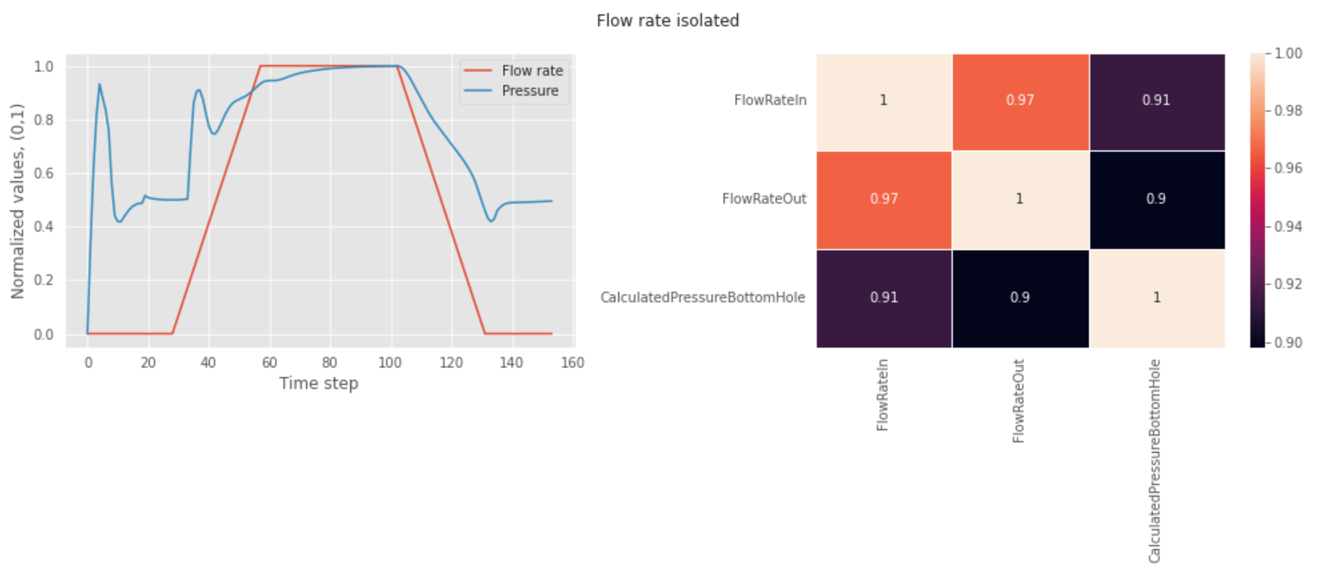


Figure 3.14: Simulation with only flow rate adjusted

In figure 3.15 the data starts at time step 20 to visualise the correlation between flow rate and down hole pressure in the case after the drill has started and the pressure has stabilised after initialisation. In this shortened view of the simulation the correlation is almost almost 1.

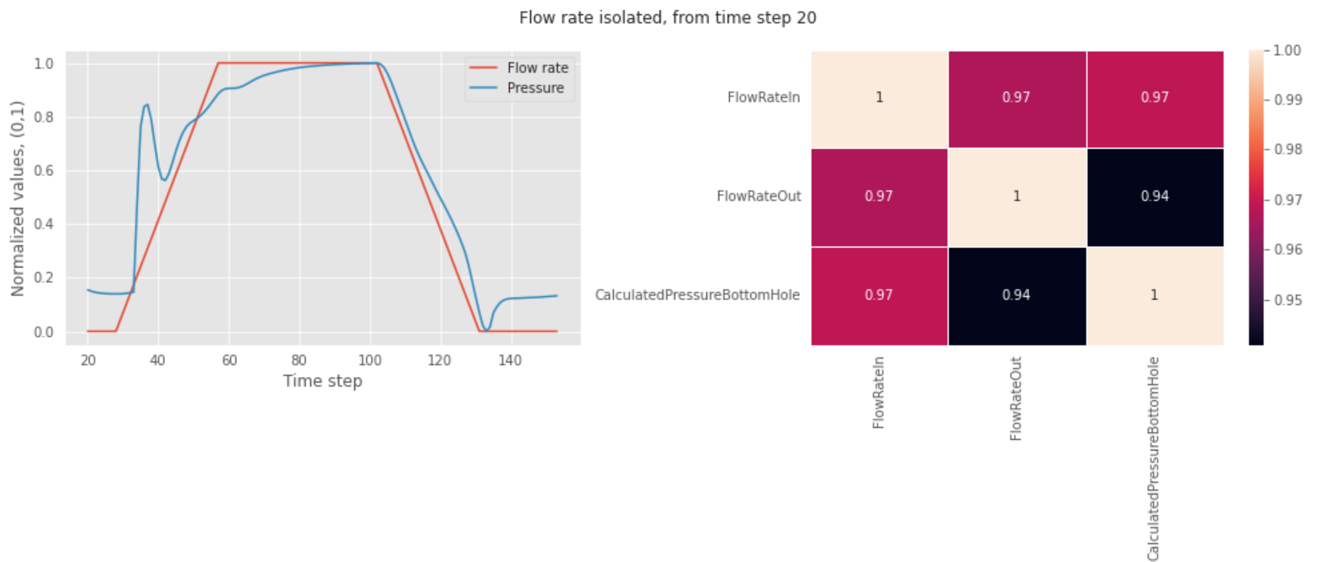


Figure 3.15: Simulation with only flow rate adjusted, shortened

Starting at time step 20 to ignore initialisation of simulation.

In the simulation shown in the figures 3.14 and 3.15 the correlation from the analysis of the precollected data still exists. Since the noise of the other parameters were largely removed the correlation is even larger, and with the shortened example having a correlation of 0.97 we can conclude that the down hole pressure is highly affected by the flow rate. However as can be seen in both the previous analysis and the full view of this simulation 3.14 the pressure can also be affected by other variables.

From the next simulation the top of string velocity will be evaluated, and similarly to the last simulation the other parameters will be kept constant, as described in 3.2.2 and the top of string velocity will be updated manually. The flow rate in this simulation is set to $2500l/min$ and the top of string velocity is $0m/s$ until the simulation is stable, and then is updated to $0.7m/s$. In figure 3.16 a similar pattern as in the figure 3.14 and 3.15 is seen where the top of string velocity is highly correlated with the down hole pressure.

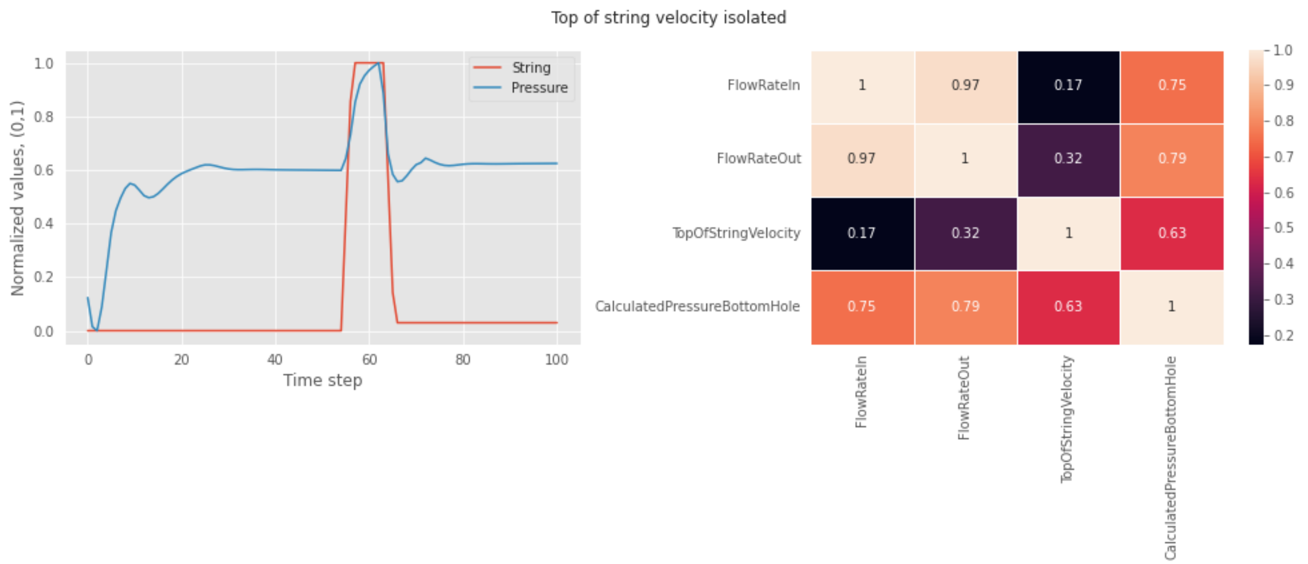


Figure 3.16: Simulation with only top of string velocity adjusted

In figure 3.17 the visualisation starts at time step 50 and it is possible to see that without the influence of the flow rate the top of string velocity is also correlated with the down hole pressure with a factor of > 0.9.

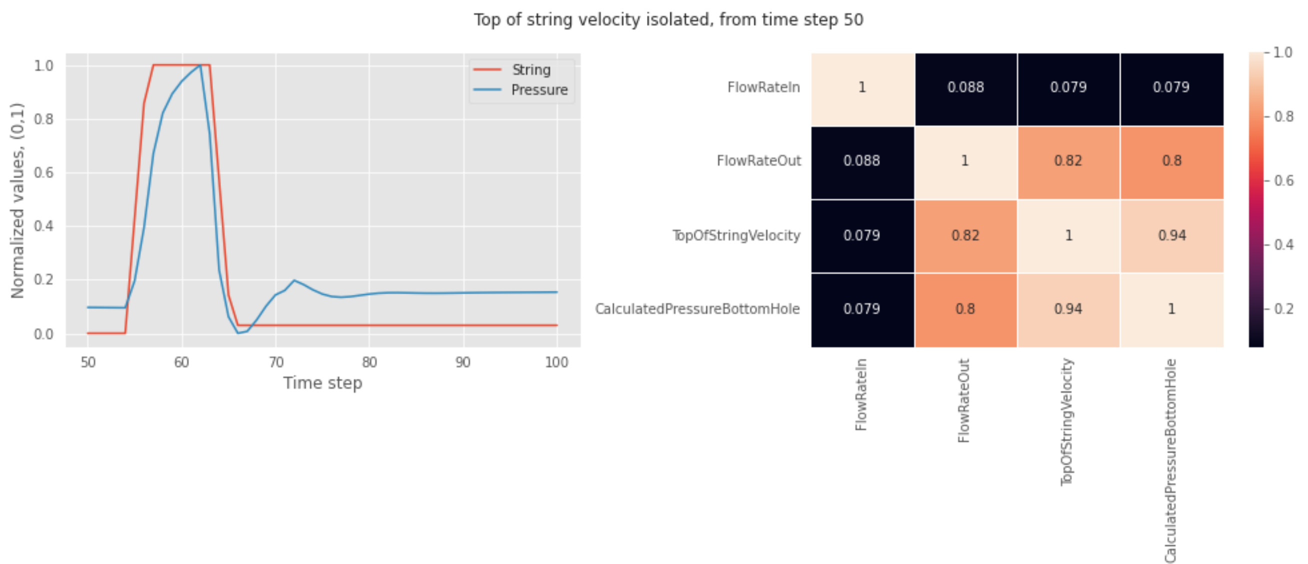


Figure 3.17: Simulation with only top of string velocity adjusted, shortened

Starting at time step 50 to ignore initialisation of simulation.

The simulation focusing on the top of string velocity, figures 3.16 and 3.17 exhibits similar results to the simulation focusing on flow rate. The top of string velocity can

then be used as a tool in designing the environment to make it more unpredictable for the agent. It might be more difficult to design an environment with focus on top of string velocity as an action, because top of string velocity is the amount of speed in m/s that the drill string is pushed downwards. Controlling this set point value would lead to spikes in pressure rather than a sustained value to keep within range of, but it could be an element to add to the environment.

The final parameter to be viewed is the surface RPM and surface torque which in the analysis of the precollected data had a high correlation with the down hole pressure in the two first episodes shown, figures 3.7 and 3.9. However the third simulation 3.11 had close to none correlation with the down hole pressure. In the analysis of this parameter the goal is to evaluate the surface RPM correlation with the down hole pressure and how much the surface RPM affects this pressure. Similar to the two previous simulations the other two set points is kept constant, however the flow rate is set to $1500 l/min$. The surface RPM is first raised to $120 RPM$ then to $0 RPM$ and to $200 RPM$, which can be seen as the two high points in the orange graph.

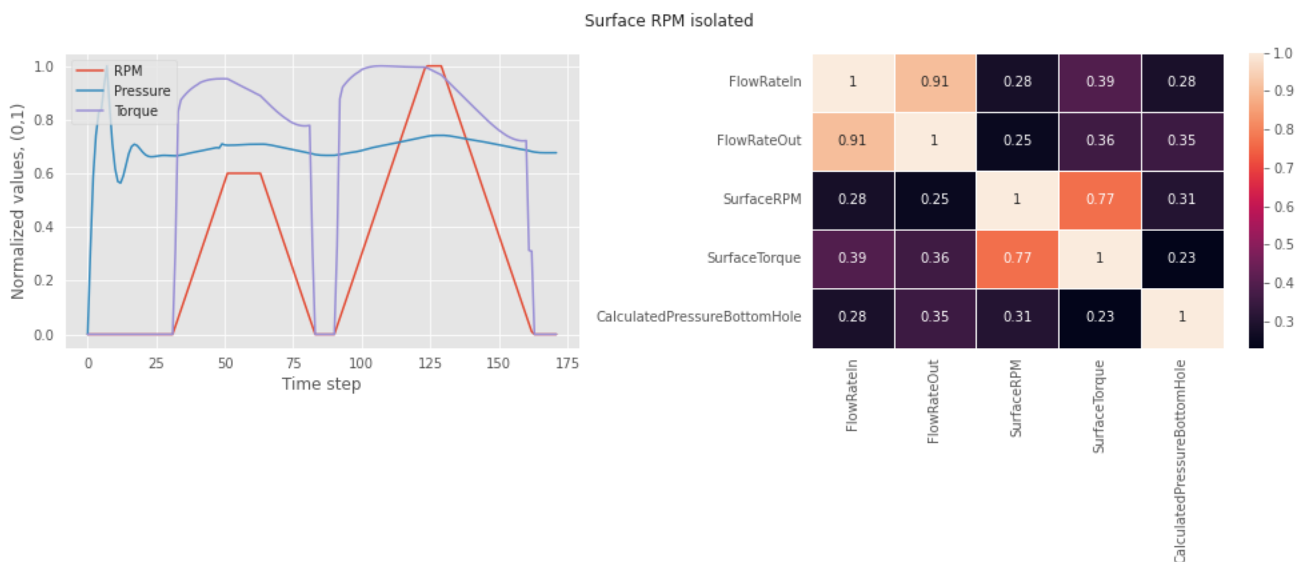


Figure 3.18: Simulation with only surface RPM adjusted

In the precollected data set it was only logged the surface torque, but the surface RPM is related to the surface torque, as the RPM is the rotational speed of the top drive and surface torque is the torque required to rotate the entire drill string and drill bit. In the correlation matrix 3.18 the surface RPM and surface torque is correlated. However neither surface torque or surface RPM is heavily correlated with the bottom

hole pressure, as can be seen in the both the correlation matrix and line graph in figure 3.18.

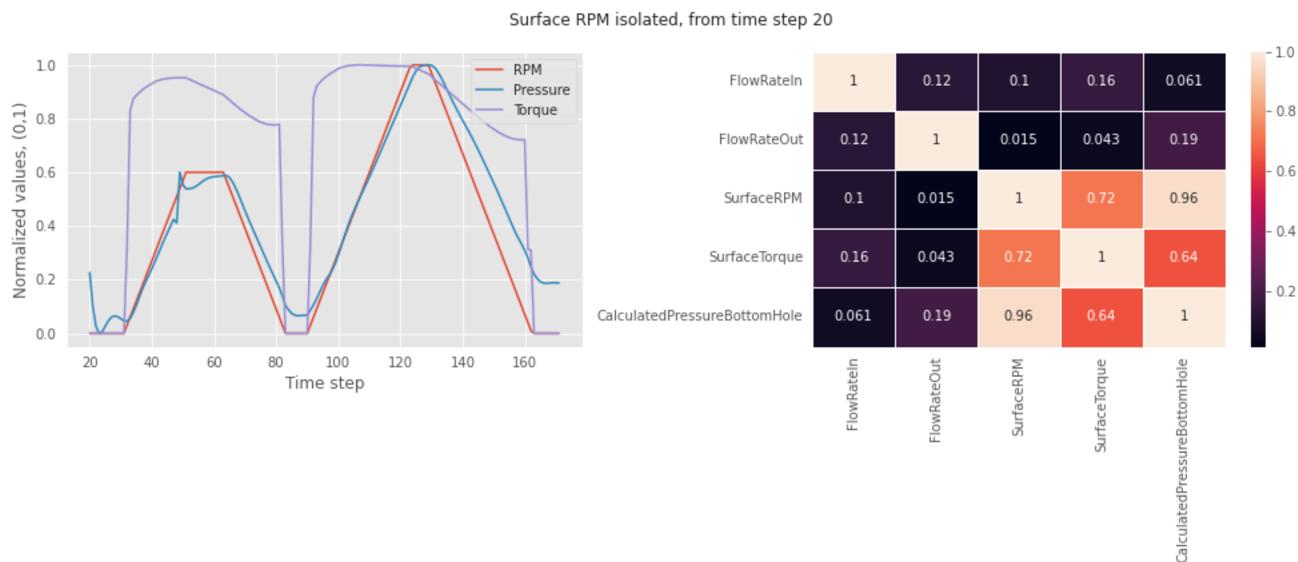


Figure 3.19: Simulation with only surface RPM adjusted, shortened

Starting at time step 20 to ignore initialisation of simulation.

The simulation from time step 20 display a large correlation between surface RPM and down hole pressure, figure 3.19. When all other values are stable the surface RPM manages to increase and decrease the pressure, but as seen in the figure 3.18 not in a substantial amount compared to the two other adjustable set points. For reference the real value difference between the maximum and minimum pressure in figure 3.19 was 231332 Pa, which is insignificant compared to the values flow rate and top of string velocity managed to influence the pressure.

In this analysis it has been observed that all three adjustable set points can affect the down hole pressure, but in differing degree. The surface RPM while in isolation has a high correlation with the down hole pressure does not affect the pressure in a large numeric amount compared to the other two parameters. The flow rate and top of string velocity can both affect the pressure in a significant amount, but the top of string velocity has more difficult mechanics to take into account and therefore would be more difficult to use to control the down hole pressure. An important note to make is that in this analysis we have only observed these set points in isolation there could be combinations or special case incident that were not found in this analysis that could have a large effect on the down hole pressure due to the nature of a dynamic physics based simulation.

3.2.3 Insights from the analysis

Based on the information gathered in the analysis the flow rate was the set point parameter chosen to use as a controllable action in the experiment in this thesis. This was a result of the flow rate having simple mechanics on how it is controlled and having a large range of pressure it could affect compared to the other two alternatives.

When designing an efficient reward function for the pressure, it is noted that the pressure operates in a range between 3.4e7 and 3.8e7 in the collected simulation, but when only adjusting for flow rate it is a narrower range. Also looking at the regression analysis of depth and pressure we can see that creating reward goals for pressure needs to take into account how deep the bit is to create feasible goals. These reward goals will only work for the configuration of OpenLab used in this thesis as they are built upon assumptions based on the data from a single well configuration.

The pressure goal function is the defined as:

$$\begin{aligned} pressure_goal =& 3.57e7 + (normal(0.45, 0.2) * 2 - 1) \\ & * 1e6 + (bitDepth - 2500) * 16200 \end{aligned} \tag{3.11}$$

The pressure goal can be divided into three parts, the base value $3.57e7$, the random adjustment $(normal(0.45, 0.2) * 2 - 1) * 1e6$ and accounting for the depth of the drill bit $(bitDepth - 2500) * 16200$. The value of $3.57e7$ was chosen as a base value to work around since it corresponds to median value at 2500m depth when only adjusting for flow rate. The Gaussian distribution was chosen to reflect the maximum and minimum values possible to reach when only adjusting for flow rate. Finally, for adjusting bit depth calculation using the coefficient term in the regression analysis, see figure 3.13.

The rewards for each time step can then be formulated as:

$$reward = \begin{cases} 1, & \text{if } (pressure_goal - 1e5) < \\ & actual_pressure < (pressure_goal + 1e5) \\ \frac{-abs(pressure_goal - actual_pressure)}{1e5}, & \text{otherwise} \end{cases} \tag{3.12}$$

This rewards the agent when managing to keep within $1e5$ Pa, but also punishes the agent more the further away from the pressure goal it is. The reasoning for this reward model was to have a simple model that rewards the agent for reaching a goal, but also penalises the agent for moving further away from the goal. A disadvantage of this reward

model is if there exists situations where the DreamerV2 agent first needs to increase the difference of the down hole pressure and pressure goal to be able to reach the pressure goal afterwards. In this situation it might be more difficult for the agent to learn how reach the wanted pressure goal.

3.3 Experiment setup and training

Chapter 3.3 will explain the definition, setup and execution of the experiment of comparing DeamerV2 agent's output to the results from the OpenLab simulator. Chapter 3.3.1 details the problem statement and motivation for the experiment. While in chapter 3.3.2 the OpenLab simulator was configured, the OpenLab simulator as a reinforcement learning environment setup and how the DreamerV2 was configured. Chapter 3.3.3 details of how the DreamerV2 agent was trained in the environment, and what goals and results was expected from the training. Finally in this chapter there is an overview of the criteria of how the Dreamer V2 agent will be evaluated in chapter 3.3.4.

3.3.1 Problem

This thesis' aim is to provide answers to the following three key questions:

- Can the world model of the DreamerV2 agent model future states of the OpenLab simulator given the actions preformed during the simulation?
- Can the DreamerV2 agent control the flow rate in the drill string in the OpenLab simulator to reach the given goal for the down hole pressure?
- Given a specific change in the OpenLab environment after finished training, does the DreamerV2 agent benefit from continuous updates from a short-term perspective?

We aim to explore the capabilities of reinforcement learning, specifically the DreamerV2 agent, in a complex dynamic environment to see if the world model of the DreamerV2 with sparse latent variables can simulate future episode trajectories. We also investigate the possibility of controlling the flow rate in the drill string given the wanted pressure as a target in the state representation. From a reinforcement learning viewpoint it is interesting to see how well a model based reinforcement learning agent performs a dynamic and accurate physics simulator like OpenLab. To observe if the categorical latent state representation of the Dreamer V2 agent, manages to create representations of the vector input from the OpenLab simulator, that provides both good decisions and adaptable modeling capabilities is the last focus point of this thesis

3.3.2 Environment

OpenLab configuration

The OpenLab simulator has two steps of setup the configuration, and the simulation setup. Two parameters indicate what type of rig the drilling is executed on, and what type of well template to choose. A rig is the machine used to drill a well, in the OpenLab configuration it is possible to choose between four different rigs:

- Generic offshore
- Generic onshore
- Mariner
- Ullrigg

In this thesis the generic offshore rig is used since this is the standard option. For the second parameter there is the well template, where it is possible to choose horizontal, inclined or vertical wells with different depths. We use the $InclinedWell$ $2500m$ as the well template.

For the simulation setup it is possible to configure the initial values of several parameters, and the values used in this thesis can be seen in table 3.3.

OpenLab as a reinforcement learning environment

To implement the OpenLab simulator as a reinforcement learning environment there are two main functions that needs to be expressed. The transition function $s_{t+1} = P(s_t, a_t)$, and the reward function $r_{t+1} = R(s_t, a_t)$. These two functions are unified to a single function $step(s_t, a_t)$ to follow the OpenAI gym library[3] since the PyDreamer implementation is built upon this template. The action space is defined as three possible actions, increase in flow rate, decrease in flow rate, or no change in pressure. Each action changes the wanted output of the flow rate with $100l/m$ since this is the amount of fluid the OpenLab simulation is able to change per second. Following the OpenAI gym library there are three main function that needs to be implemented to make the DreamerV2 agent work in conjunction with the OpenLab simulator: The step function, the reset function, and the init function.

Listing 3.2: Episode following OpenAI gym[3] template.

```
"""
The environment variable is a reinforcement learning environment
The state variable is initialized as the first state in the environment
"""
environment = ReinforcementLearningEnv()
state = environment.reset()
done: bool = false
while not done:
    action = agent.get_action(state)
    new_state, reward, done = environment.step(action)
    state = new_state
next_initial_state = environment.reset()
```

The init function initializes the environment class instance, establishes a connection with OpenLab, and chooses a configuration of the well. The reset function deletes the previous simulation, and configures and initializes a new simulation. The step function handles changing the set points of OpenLab based on the action from the agent, completes a single step in OpenLab with new set points. The step function then retrieves the new state from OpenLab and calculates the reward from the new pressure. Finally the step function normalizes the new state and returns the normalized new state and reward.

For this experiment we chose to keep the adjustable set points that were not controlled by the agent as constants similarly to the analysis of isolated parameters 3.2.2. The top of string velocity and surface RPM were then initialised in the reset function at the start of each new episode with the values:

Surface RPM	$120RPM$
Top of string velocity	$0.7m/s$

Table 3.3: Initial set point values

Listing 3.3: Pseudocode of OpenLab environment

```
class OpenLabEnvironment:

    def __init__(config_name: str, credentials: str, initial_bit_depth: int = 2498):
        """
        Establish a connection with OpenLab
        Choose configuration
        """
        self.session = openlab.connect(credentials, config_name)
        self.simulation = None
        self.initial_bit_depth = initial_bit_depth

    def reset():
        """
        Delete previous simulation
        initialize and configure new simulation
        """
        if self.simulation is not None:
            openlab.delete(self.simulation.id)
        self.simulation = self.session.create_simulation(self.config_name, self.
    initial_bit_depth)

        # Set wanted initial set points
        self.simulation.setpoints.surfaceRPM = ...
        self.simulation.setpoints.flowRateIn = ...
        self.simulation.setpoints.topOfStringVelocity = ...
        self.timestep = 1
        self.simulation.step(self.timestep)
        intial_state = self.simualation.get_results(self.timestep)
        self.timestep = self.timestep + 1
        return self.normalize(initial_state)

    def step(action):
        """
        Change set points based on action
        Simulate one step with OpenLab
        Retrieve new state from OpenLab
        Calculate reward
        Return new state and reward
        """
        self.change_setpoints(action)
        self.simulation.step(self.timestep)
        new_state = self.simualation.get_results(self.timestep)
        self.timestep = self.timestep + 1
        reward = self.reward_function(new_state)
        return self.normalize(new_state), reward
```

DreamerV2 configuration

The DreamerV2 parameters configuration can be viewed in table 3.4. It's configuration is similar to previous benchmarks, with changes only made to hardware limitations.

Hardware is defined in table 3.5.

Name	Value	Name	Value
default:		**# Generator**	
n_steps:	1 000 000	generator_workers:	5
n_env_steps:	1 000 000	generator_workers_eval:	1
offline_prefill_dir:	precollected data directory	generator_prefill_steps: 100_000	100 000
log_interval:	100	generator_prefill_policy:	precollected random
logbatch_interval:	1000	**vectorenv:**	
save_interval:	100	env_id:	OpenLabLive
eval_interval:	20 000	action_dim:	3
data_workers:	5	vecobs_size:	6
buffer_size:	10 000 000	**debug:**	
kl_balance:	0.8	device: cpu	cpu
kl_weight:	1.0	log_interval:	5
vecobs_weight:	1.0	save_interval:	10
reward_weight:	1.0	eval_interval:	20
terminal_weight:	1.0	data_workers:	1
adam_lr:	3.0e-4	generator_workers:	1
adam_lr_actor:	1.0e-4	generator_prefill_steps:	10 000
adam_lr_critic:	1.0e-4	batch_length:	50
adam_eps:	1.0e-5	batch_size:	5
batch_length:	50	imag_horizon:	15
batch_size:	50	amp:	False
# Actor Critic			
gamma:	0.995		
lambda_gae:	0.95		
entropy:	0.003		

Table 3.4: DreamerV2 configuration

GPU	Nvidia 3060 TI
CPU	AMD Ryzen 5600X
RAM	32GB

Table 3.5: Hardware

3.3.3 Training description

The PyDreamer model was first trained for 1 000 000 environment steps in the online OpenLab environment. Where the first 100 000 steps was the precollected simulations analysed in chapter 3.2.1 which was loaded into the training data directory and the remaining 900 000 steps were trained online with the OpenLab simulator. Five generator workers were utilised to collect data during training. This was limited by the OpenLab license where there could only be run 5 simulations simultaneously. The rest of the parameters were set to the recommended training values from PyDreamer benchmarks.

Difficulties and solutions during training

There were some limitations that had to be added during training due to OpenLab server instabilities. The first of these limitations were to the length of the drilling episodes, where longer episodes had a greater tendency to crash. This led to implementing a step length limit of 4500 steps to avoid crashing episodes. Under the license from OpenLab there was the possibility of having episodes up to the length of 18 000 steps, but this could not be utilised. Another challenge was if the simulation crashed, the last time steps before the crash the values reported was outside the range of values accounted for in the normalization step, which led the neural networks to crash, for example pressure up to the value of $1.6e10\ Pa$. Therefore a safety measure was implemented to abort episodes that reported over $4e7\ Pa$ which was larger than any value observed in the precollected data.

A final problem was seeding the random generators used in the environment since they were run in parallel, the solution for this was seeding the different class instances of the environment with the associated worker id of that thread.

Goal for training

The goal of the training is to see if the DreamerV2 is able to improve over 1e6 environment time steps. It would be preferable if we could have done more training since in the DreamerV2 paper[14] it takes closer to 1e7 time steps in most environments before it starts to converge. However this would take several days of training and simulation both for the DreamerV2 and the OpenLab simulator.

3.3.4 Evaluation

The first evaluation is to evaluate the performance of the DreamerV2 agent on how well the world model manages to model the OpenLab environment, we will run a simulation in OpenLab, and the repeat the same actions done within the world model. Here we can then give an estimate on how well the model can predict the environment. One drawback of this approach is that we are also evaluating the reconstruction of the latent state. The latent space itself might be a more accurate representation, but we cannot easily interpret it.

Next we evaluate how well the DreamerV2 agent manages to control the flow rate to reach down hole pressure values matching given pressure goals that the agent aims to achieve. In this evaluation the goal is to see if the DreamerV2 agent can control a part of a drill system, the flow rate, to reach a given pressure goal. Here, the online learning capabilities of the DreamerV2 agent will also be evaluated. If there is a change in the environment will the DreamerV2 be able to adapt in a short time period?

Chapter 4

Results and Evaluations

In this chapter the results obtained during training of the DreamerV2 agent will be presented and discussed as well as the results from the evaluation will be presented and discussed in chapter 4.2. The final chapter 4.3 will detail recommendations for further research combining OpenLab and reinforcement learning.

4.1 Results obtained during training

The results obtained during training of the DreamerV2 agent is analysed in this chapter, to try to gain insights in trends in the quality of the output and especially focus on where in the process the DreamerV2 agent starts to learn. First of all an episode from the start of training will be presented, to visualise how the algorithm preforms with only the precollected data set and gradient updates with the first 100 batches. Then we will look at a selection episodes in the middle of training period where the DreamerV2 agent starts to improve its performance. At last how the DreamerV2 performs at the end of the training session.

The figures 4.1, 4.2, 4.3 and 4.4 is each illustrating one episode during training. The top left graph showing the comparison between the wanted pressure goals given to the agent, represented by the blue line, and the the down hole pressure achieved during the simulation, represented by red line, these values are normalised with the formula detailed in equation 3.10. The top right graph is a visualisation of the actions preformed by the agent, that is increase action is +1, decrease action -1, and no change in flow rate action

represented by 0. Since these values are not representative of any real measurement they are scaled to visualise the similarities between actions carried out by the agent and pressure in the bottom graph.

After training 100 batches of 50 random sampled, 50 sequence length trajectories from the offline precollected data set, the world model and actor critic has not yet learned representations and strategies to a quality high enough to earn any good return. This can both be observed in the visualisation of the down hole pressure compared to the pressure goal in the first episode 4.1, and in the model loss in figure 4.6.

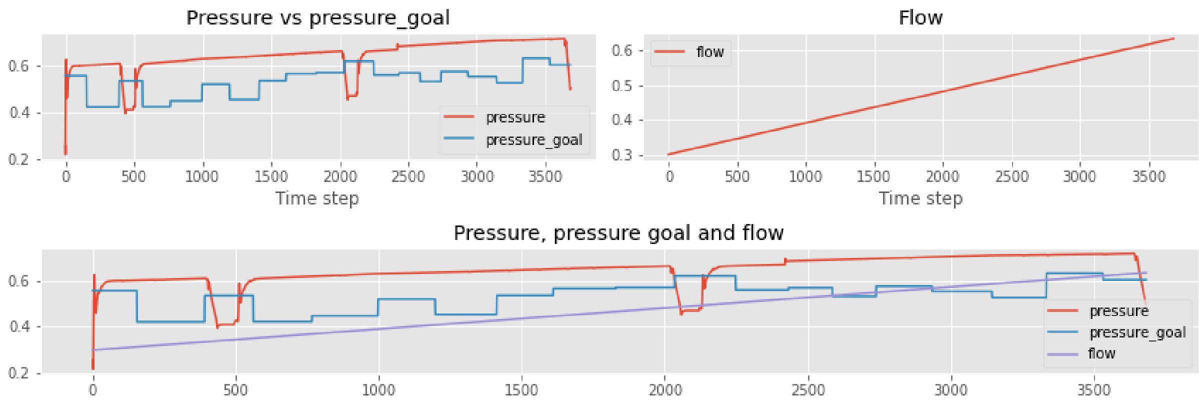


Figure 4.1: Example of episode 1 during training

The drop in down hole pressure around time step 500 and 2000 is a result of the top of string needing to be replaced as a result of being around 1 meter above the drill floor and then replaced with a new section of drill string[23]. This process cuts of circulation, but the DreamerV2 agent neither knows the actual flow rate, and since the reward model does not take this mechanism into account the agent often tries to compensate for this pressure drop in future episodes by increasing the flow rate without being able to preform any increase, since the circulation is cut off.

Already in episode 35, figure 4.2, it is possible to see the improvement of the agent. While not able to maintain a stable pressure, the agent has stopped performing only one action. In this episode the agent is clearly trying to reach the given pressure goal, but it has a tendency to overcompensate and therefore increasing and decreasing the pressure too much.

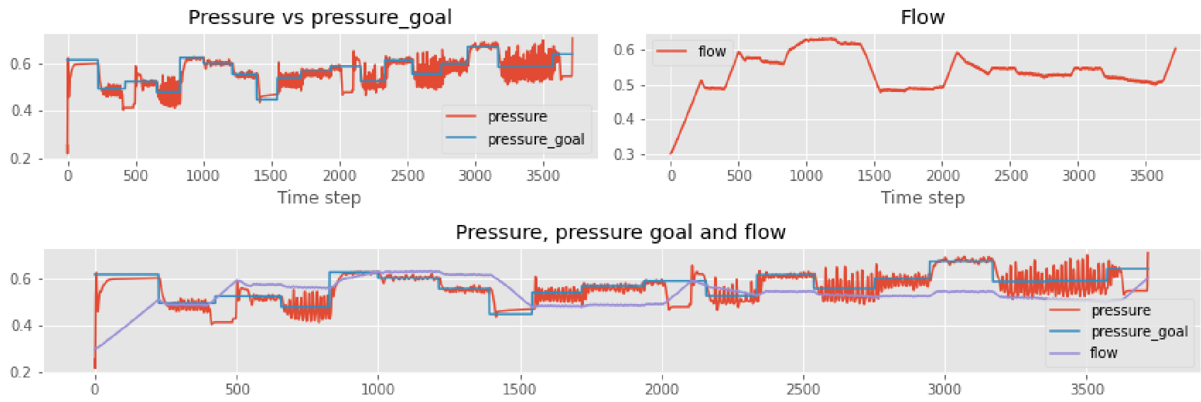


Figure 4.2: Episode example after around 35 episodes of training

In episode 85, figure 4.3, there are more stability in the down hole pressure during the middle of the episode. However at the start and the end of the episode, there is an instability when the agent is trying to control the flow rate to reach the given pressure goals. In both cases this instability occurs when the agent is trying to decrease the pressure and this ends in instability as can be observed around time step 200 and time step 3100. When looking at the actions shown in the flow graph, figure 4.3, there is only a slightly staggered decrease in flow rate and it is not obvious why this gives such large oscillations in the down hole pressure. In the middle sections the pressure is decreased with a continuous reduction without small stops for each increment and this does not give the same instability in pressure values.

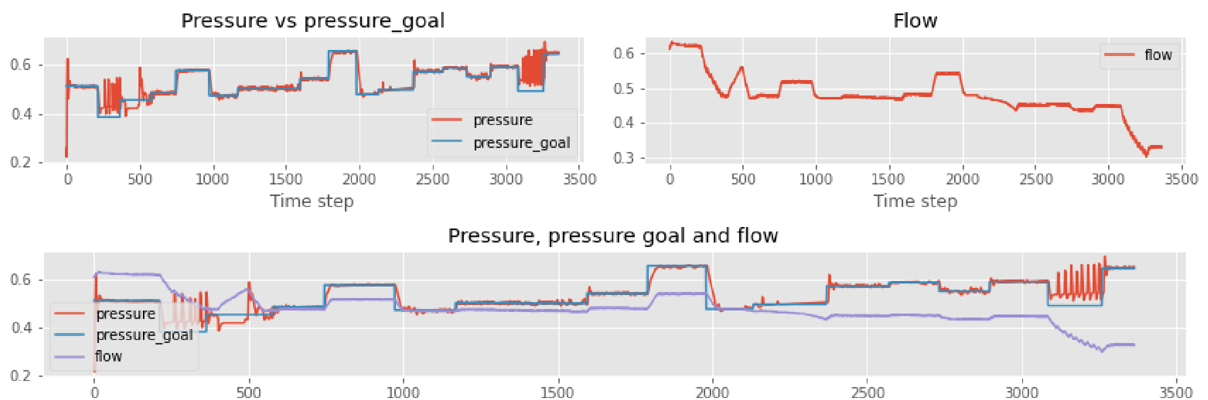


Figure 4.3: Episode example after around 85 episodes of training

Episode 275, figure 4.4, illustrates that the agent is capable to reach a given pressure goal by controlling the flow rate. Compared to the previous example when decreasing the

pressure, this time it is done with a continuous reduction of the flow rate which could be the reason for the unstable pressure in the last example. In episode 275 in time step 2000 the DreamerV2 agent tries to increase the pressure by increasing the flow rate to reach the given pressure goal, even though the circulation is cut off as a result of changing the drill string..

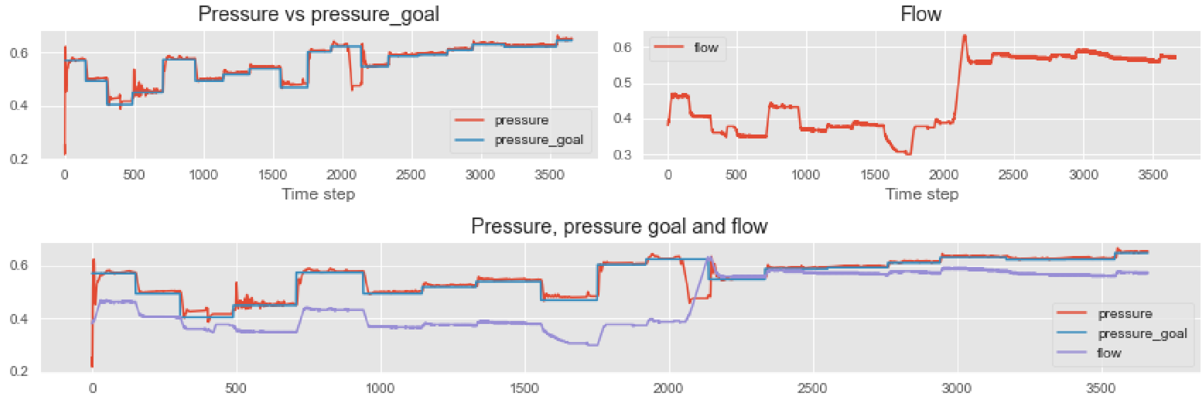


Figure 4.4: Episode example after around 275 episodes of training

In one of the last episodes of the training, figure 4.5, though this had a return of −1809, this episode was chosen as an example to illustrate how the DreamerV2 agent preforms in an episode with a large negative return. In this episode the agent manages to reach and hold the given pressure goals fairly well, but around time step 1000 there is a pressure goal with a too high value, that leads to negative rewards thus the agent receives more negative rewards. In the later half of the episode the agent also does not reach its pressure goals as well. The reason for this could be that the agent had more training on the first 2000 time steps, while this episode lasted for more than 4000 time steps. Most training episodes reached 2000 time steps, but fewer reached 4000 time steps, since the length of an episode is chosen randomly at the start of the episode and in longer episodes the simulator crashes more often. In the second half of the training episode the DreamerV2 agent preforms acceptable, but the instability in down hole pressure appears again when decreasing the flow rate in a staggered manner.

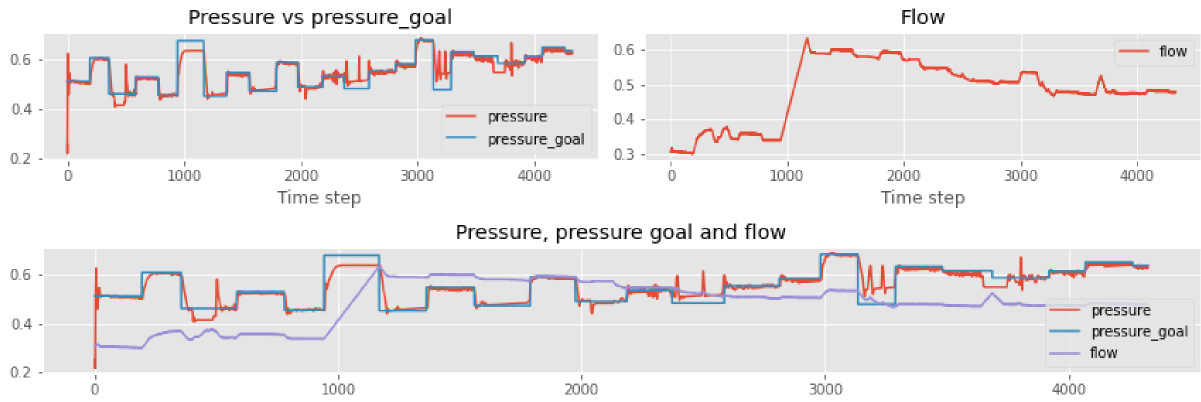


Figure 4.5: Episode example after around 300 episodes of training

From the graphs of the different loss functions during training, shown in figure 4.6, it is possible to observe that when the model loss and KL balancing loss starts to stabilise, and the world model is able to create both good representations and good imagined trajectories, the actor loss improves. A further observation is that the critic loss does not stabilise, that could be explained by as long as the actor is finding better solutions, the previous approximation the critic had given for the advantage function is now outdated. Thus when the actor improves of the critic becomes worse and needs to improve.

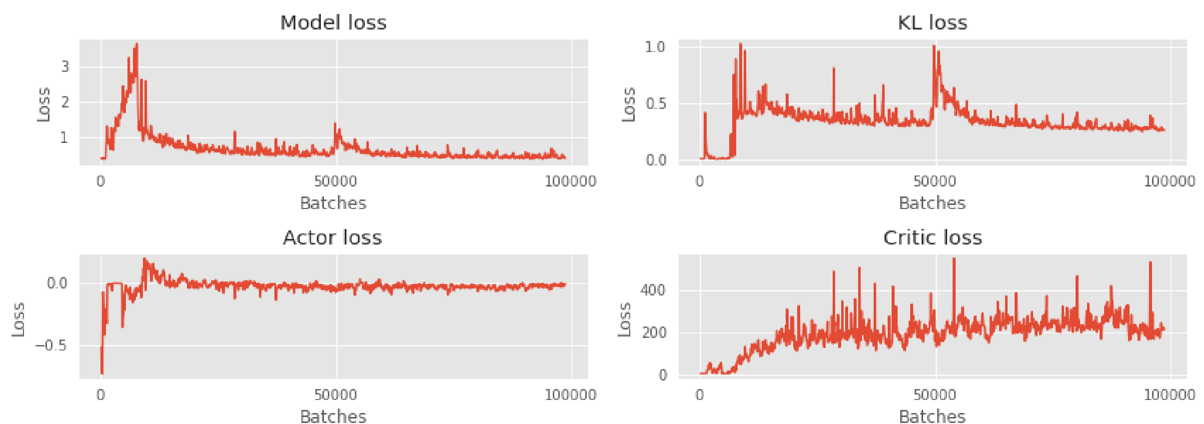


Figure 4.6: Loss during training

Figure 4.7 represents the interquartile mean(IQM) of the return for every 25 episodes during training, where it can be seen that the DreamerV2 agent continues to improve during the entire training session and does not converge towards the end, therefore the

DreamerV2 could benefit from more training. This is expected since in this experiment the DreamerV2 only received 1 000 000 environment samples, compared to in the evaluation the the original DreamerV2 paper the agent started to converge around 10 000 000 time steps[14].

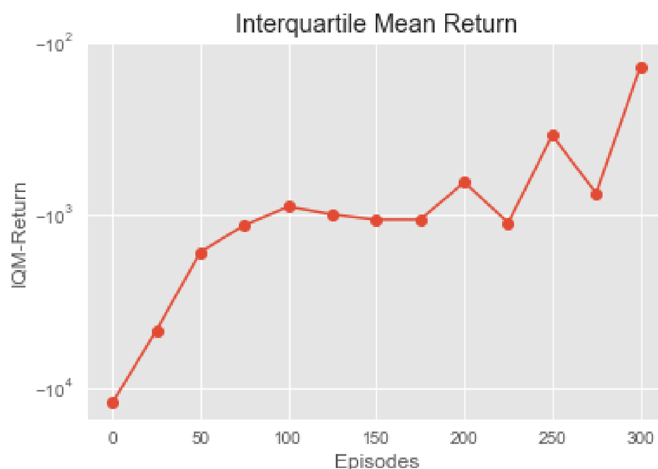


Figure 4.7: Return during training

At the end of the training session the DreamerV2 agent has improved in being able to control the flow rate to reach and hold a given pressure goal of the down hole pressure. The agent has learned that in this configuration of OpenLab when decreasing the flow rate it needs to be decreased continuously until reaching the wanted down hole pressure, because decreasing the flow rate in a staggered manner will lead to large oscillations of the down hole pressure. It is also indications of that the DreamerV2 agent preforms worse in longer episodes, where it is less able to hold a stable pressure towards the end of the episode. Observing the improvement of return during training it does not seem to converge towards the end of the training session and therefore could benefit of more training.

4.2 Evaluation

In this chapter the DreamerV2 agent's capabilities in the OpenLab environment will be evaluated, together with a discussion of the strengths and failings of the agent. Initially the modeling capability's of the DreamerV2's world model will be evaluated. In this evaluation the DreamerV2 was evaluated on the quality predictions it managed to make

about future states. Then the control ability of the DreamerV2 will be evaluated, where the environment is changed slightly in form of the top of string velocity and surface RPM are changed randomly. Where both the control abilities of the DreamerV2 in the case of new data will be looked at, and the online abilities of how fast does the DreamerV2 adapt.

4.2.1 Modelling results

To evaluate the modelling capabilities of the DreamerV2's world model we seek to answer this question:

- Is the world model capable to reproduce some of the dynamics of the simulator when executing the same actions?

To evaluate if the world model is capable to reproducing the dynamics from the OpenLab simulator, its behaviour is observed if given the same actions as the OpenLab simulator, does the internal dynamics of the world model react similarly?

Figure 4.8 is a visualisation of the predicted pressure compared to the actual pressure received from the OpenLab simulator. This prediction was calculated by using the 5 first initial states, where each state lasts for 1 second, of a completed simulation. The world model were then given the flow rate actions preformed during the simulation to try to reproduce the simulation episode by use of the recurrent model and transition model of the world model. Comparing the actual pressure reported from OpenLab and the predicted pressure simulated internally in the world model it is possible to observe a similar trajectory between the predicted pressure and actual pressure. The mean absolute error(MAE) comparison was 320715.359 Pa.

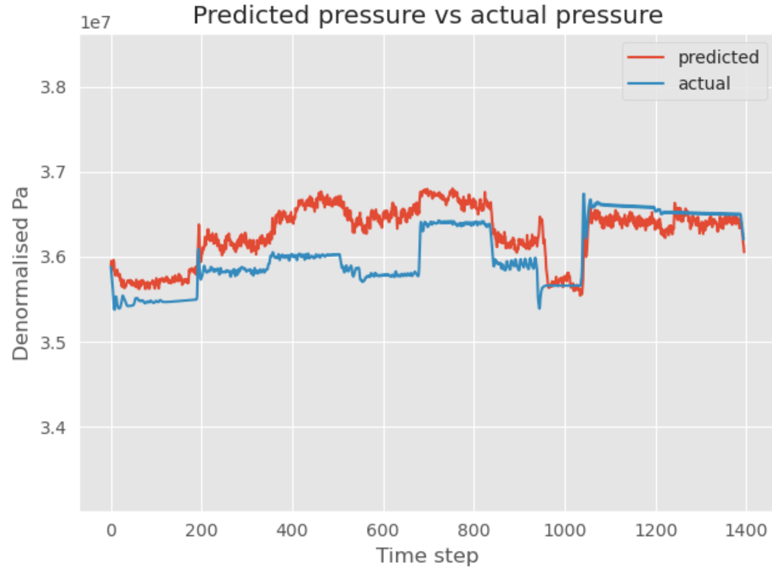


Figure 4.8: Predicted future pressure given first 5 states and actions, short episode.

Figure 4.9 illustrates a similar pattern with a MAE of 295844.09 Pa which is a significant amount. Some of this error is caused from the changing of the top of string which happens at around time step 500, 2000 and 3500 which the world model simulation has no information of, resulting in the world model simulation to simulate with a larger error.

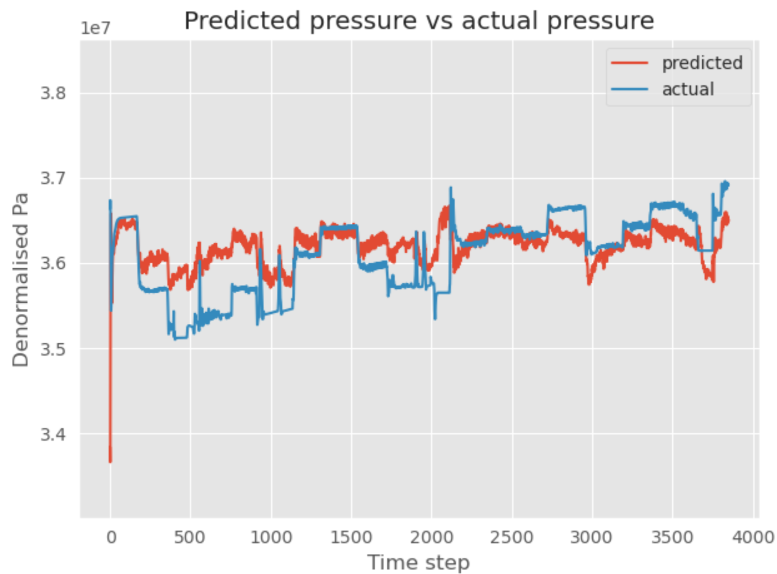


Figure 4.9: Predicted future pressure given first 5 states and actions, long episode.

There are several reasons that could affect the precision of the simulation. The reconstruction of the latent state representation could introduce errors. The lack of training

samples the DreamerV2 was trained on compared to previous evaluations implying the latent space transitions could improve with more training. The architecture of the DreamerV2 agent might not be complex or large enough to better approximate the transitions as calculated in OpenLab. The state representation the world model of the DreamerV2 receives might not contain enough information to better approximate future transitions. Also the discrete categorical latent state might generalise too much with respect to similar states, which might lead to less accurate representations.

Looking at the loss of the reconstruction of the input state in figure 4.10 it is more likely that the inaccuracies in figure 4.8 and 4.9 is a result of the transition prediction rather than the reconstruction of the latent representations, seeing how low the loss of the state reconstruction is.

Figure 4.10: Log loss of the state reconstruction

To answer the question "Is the world model capable to reproduce some of the dynamics of the simulator when executing the same actions?", the answer would be it is capable of recreating similar dynamics, but in the examples which were addressed in this thesis the world model is inaccurate when calculating the transitions. This might be a result of lack of training, a world model not complex enough or a input state representation without enough information. However if it is a result of the discrete categorical latent state not representing the information precisely, it might be that the world model is able to generalise to utilise similar previous states to make decisions in unseen states.

4.2.2 Control and online results

In this chapter it is evaluated how the DreamerV2 agent controls the flow rate in OpenLab to reach and keep the down hole pressure at given pressure goals. A new element where the top of string velocity and surface RPM values will be changed randomly during simulation to create new and different states will also be introduced, and an evaluation of how the DreamerV2 agent manages the control both with and without continuous updates of the weight parameters of the agent. The evaluation is based upon the follow three questions:

- Is the agent able to reach and maintain pressure goals given control of the flow rate?
- Is the agent able to reach and maintain pressure goals given control of the flow rate when presented with a change in the behaviour of the environment?
- Is the DreamerV2 agent able to improve its performance by updating its weight parameters in a within a few episodes, when encountering a change in the behaviour of the environment?

When carrying out updates to the weight parameters a good solution to save unfinished recorded episodes to facilitate learning during the episode due to how the wrapper environment in the PyDreamer implementation was implemented, was not found. This means that in this evaluation the updates to the agent acting in the OpenLab environment will only update its improvements to the policy at the start of each simulated episode.

In the first visualisation, figure 4.11, the three different evaluation scenarios are compared. These evaluation scenarios are:

- Evaluation of the DreamerV2 agent trained with 1 000 000 environment steps, tested in 10 episodes in the same environment as it was trained, without updating its weight parameters.
- Evaluation of the DreamerV2 agent trained with 1 000 000 environment steps, tested in 10 episodes in an environment where the top of string velocity and surface RPM is uniformly random and changed at random with a probability of 0.005%, without updating its weight parameters.
- Evaluation of the DreamerV2 agent trained with 1 000 000 environment steps, tested in 20 episodes in an environment where the top of string velocity and surface RPM is uniformly random and changed at random with a probability of 0.005%, with updating its weight parameters.

The figure 4.11 compares how many of the evaluation episodes for each scenario reach a given return score τ. This comparison make it possible to see that the randomly changing top of string velocity and surface RPM do affect the ability of the DreamerV2 to control the flow rate in order to reach wanted a down hole pressure value, is reduced. However, the comparison between the other two scenarios are more relevant, random top of string and surface RPM with and without updates to the weight parameters. Comparing these two scenarios with figure 4.11 it is difficult to establish whether one is strictly better than the other.

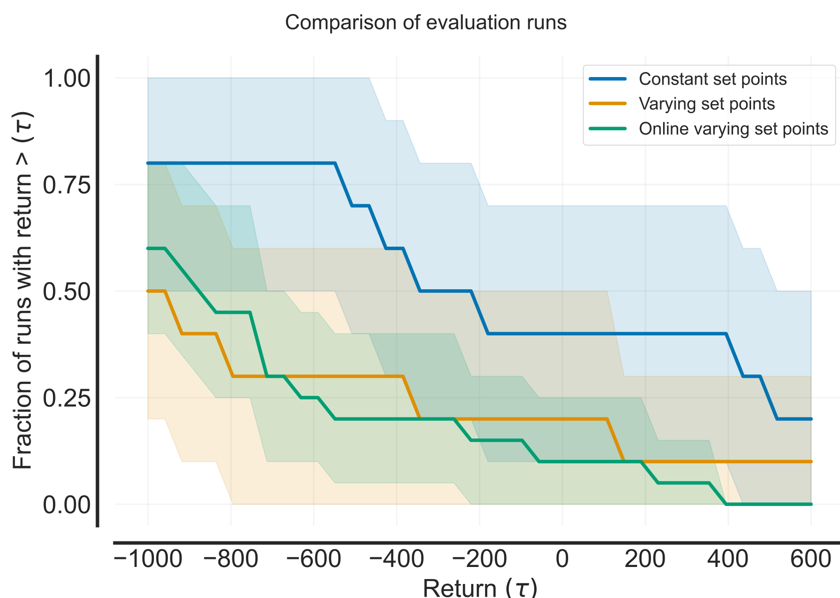


Figure 4.11: Comparison of evaluation scenarios.

Shaded area are 95% confidence bands based on percentile bootstrap with stratified sampling. Implemented using library from "Deep Reinforcement earning at the Edge of the Statistical Precipice"[1].

The interquartile mean is used to get a measure of the performance in terms of return for the three evaluation scenarios, the IQM values for each scenario are:

1. Constant top of string velocity and surface RPM without updates: **-566.20**
2. Varying top of string velocity and surface RPM without updates: **-944.51**
3. Varying top of string velocity and surface RPM with updates: **-1084.40**

According to the interquartile mean continuous updates of the weight parameters of the agent leads to slightly worse performance of -1084.40, compared to not updating the weight parameters in a short-term perspective.

In figure 4.12 one of the last episodes with continuous updates is presented, where the return of this episode was -873. As can be seen from time step 800 to time step 1000 the agent does not manage to keep the pressure stable in this period, even though the top of string velocity and surface RPM values only change around every 200 time step. The figure is structured in the same manner as the figures in chapter 4.1 with a comparison of the normalised actual down hole pressure and normalised pressure goal in the top left of the figure. In the top right a visualisation of the action performed to the flow rate by the agent, scaled in a range to more easily compare all three plots in the bottom graph of the figure.

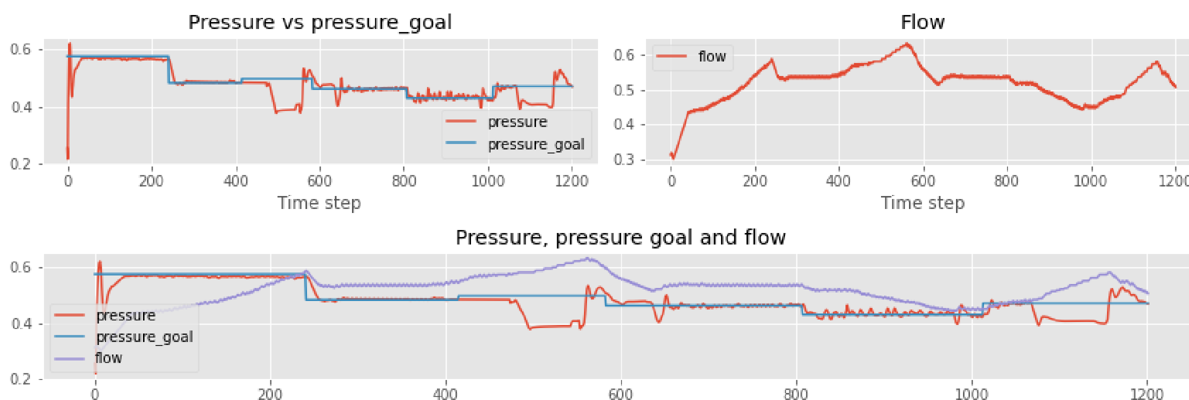


Figure 4.12: Episode from the end of evaluation with weight parameter updates.

It can be seen in the chapter 4.1 and in the figure 4.11 that the DreamerV2 agent manages to reach and maintain the pressure goals in the OpenLab simulator environment given control of the flow rate. The agent exhibits the ability to reach pressure goals both with higher and lower pressure values than the current pressure in an effective manner, for example if the agent decreases the pressure it has to be done in a continuous manner to avoid large oscillations of the down hole pressure.

When encountering new unseen states the DreamerV2 agent preforms as expected worse than the previous evaluation scenario. The agent is however, by the measure of return performing more than 10 times better than it did at the start of the training session. By this measure it is possible to determine that the DreamerV2 agent is then able to generalise and preform to some degree even when the behaviour of the environment change.

Based on the results of the evaluation of preforming online updates while presenting a change in the behaviour of the environment, there was no improvement in the performance

of the agent. Rather there was a slight decline in performance compared to the case of not performing updates to the weight parameters. A possible reason could be that while the world model is learning new and better representations for a change in behaviour of the environment, the actor critic also has to adapt to the new representations from the world model. This adaption could lead to the actor critic needing longer training than was given during this evaluation.

4.3 Further research

OpenLab represents an interesting environment for further reinforcement learning in a high-fidelity physics simulator, as a step towards the use of reinforcement learning in a real world scenario. More specifically OpenLab has several different well configurations of interest. This is a possibility to explore reinforcement learning agents for a larger general problem with the use of different well configurations.

To explore the capabilities of reinforcement learning agents in a even more accurate environment with the usage of OpenLab's transient torque-drag model. Which is more advanced compared to the steady state torque and drag model utilised in this thesis.

Creating a more complex goal for the agent might be interesting to further explore reinforcement learning in OpenLab.

Exploring more reinforcement learning agents in OpenLab would give further knowledge on the capabilities and limitations of reinforcement learning in high-fidelity physics simulators.

Chapter 5

Conclusion

In this thesis we explored model-based reinforcement learning in a high-fidelity physics simulator. By exploring this combination of model-based reinforcement learning and a high-fidelity physics simulator, the modelling capabilities of the DreamerV2's world model to model the OpenLab simulator environment was investigated.

The design of the experiment was made with the purpose of investigating the DreamerV2 agent's ability to control the flow rate in the drill string to reach and maintain given pressure goals and further the DreamerV2 agent's ability to adapt to a change in the environment.

In the Background materials chapter the basis for the thesis experiments was established. The theoretical material regarding machine learning, reinforcement learning, online and offline learning and the OpenLab simulator was presented. The concept of world models used by reinforcement learning agents to learn an internal generalised model of the environment was introduced. Furthermore a discussion of adding an evaluation of the online learning capabilities to this thesis, to evaluate the agent's adaptability to changes in the behaviour of the environment was carried out.

The Methodology and Experiment chapter the main functionality of DreamerV2 agent is described, its components and how the agent is trained. A study of how the different OpenLab parameters and set point parameters were affecting each other was analysed, and this insight was utilised to design the experiment where the goal is to explore the DreamerV2's ability to reach pressure goals for the down hole pressure.

With the outcome from the Results and Evaluation chapter it was found that the world model of the DreamerV2 agent was capable of modelling the environment to a

certain degree. When only given a few initial states of information and the flow rate actions performed in OpenLab, the world model was able to reproduce of the down hole pressure that was comparable to the original simulator episode. This result demonstrates that the 32 categorical variables of the latent state is capable of creating a satisfactory representation of the OpenLab environment. As a result of this the recurrent model and the transition predictor is able to predict future transitions. In this new framework with a low dimensional vector as input contrarily to high dimensional tensor images, as used in previous external evaluations[14][1].

In the Results and Evaluation chapter, 4.1 and 4.2.2, the DreamerV2 agent was capable of controlling the flow rate to reach the desired pressure goal. The agent's control capability improved during the training. One example of this improvement was observed when it learned that a reduction in flow rate had to be carried out with a continuous decrease to the desired flow rate value. This was due to the fact that halting the reduction caused large oscillations in the down hole pressure. The agent was also able to generalise from the changes in behaviour of the environment when presented with new unseen states, where it preserved some of its ability to control the flow rate to reach the pressure goals.

During the experiment in chapter 4.2.2 with online updates to the DreamerV2 agent, an improvement in the agent's ability to adapt to a change in the behaviour of the environment was not observed. The results of this experiment showed a slight decrease in the agent's performance compared to not carrying out updates to its parameters. A possible explanation for a reduced improvement could be that the world model was learning new representations as a result of the change in the states encountered. If these representations from the world model changes significantly the actor critic then encounters states and transitions that are not only new, but could also be differently represented.

It was observed during the experiment that the world model is capable of modelling future states when given the actions performed from the OpenLab simulator within less than 1% deviation from the OpenLab simulated down hole pressure value.

The DreamerV2 agent has shown capability of controlling the flow rate in the drill string to reach the given goal of the down hole pressure in OpenLab.

When given a specific change in the environment we did not observe benefits from continuous update of the weight parameters in a short-term perspective.

Glossary

Action How the agent interacts with the environment.

DreamerV2 Is a model-based reinforcement learning agent that utilises a world model and a discrete latent state representation to achieve high results in previous benchmarks[14, 1].

flow rate Amount of drill mud pumped into the bore hole.

Gated Recurrent Unit The Gated Recurrent Unit is a version of a recurrent neural network that uses a forget gate to decide how much of the new input should be included in the hidden state, or how important is the new state compared to the previous states for future states.

Hook load The total force pulling down on the drill string hook.

Markov Decision Process A mathematical framework for modeling decision making, discussed in 2.2.6.

Model-based reinforcement learning Model-based reinforcement learning is defined as an agent utilising a model of the environment to preform planning to improve or produce a policy. See chapter 2.2.8.

NORCE The Norwegian Research Centre(NORCE)(https://www.norceresearch.no/).

Offline learning Learning from accumulated batch data.

Online learning Learning from a stream of sequential data.

OpenLab OpenLab is the simulator that is produced by Norce Drilling Well Modeling group. It is a physics based simulation of well drilling.

Rate of penetration Is the progression speed of the drill bit, measured in m/s.

Return Sum of rewards, the value which a reinforcement learning agent attempts to maximize.

Reward The immediate feedback from the environment to the agent.

State Representation of the environment at a given moment.

surface RPM The rotational speed of the top drive.

top of string velocity The amount of speed in m/s that the drill string is pushed

downwards.

Bibliography

[1] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. 2021. doi: 10.48550/ARXIV.2108.13264.
URL: `https://arxiv.org/abs/2108.13264`.

[2] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.
URL: `https://arxiv.org/abs/1308.3432`.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.
URL: `https://arxiv.org/abs/1406.1078`.

[5] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning, 2018.
URL: `https://arxiv.org/abs/1806.06923`.

[6] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego Casas, Craig Donner, Leslie Fritz, Cristian Galperti, Andrea Huber, James Keeling, Maria Tsimpoukelli, Jackie Kay, Antoine Merle, Jean-Marc Moret, and Martin Riedmiller. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602:414–419, 02 2022. doi: 10.1038/s41586-021-04301-9.

[7] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[8] Jay W. Forrester. Counterintuitive behavior of social systems. *Technological Forecasting and Social Change*, 3:1–22, 1971. ISSN 0040-1625. doi: https://doi.org/10.1016/S0040-1625(71)80001-X.
URL: https://www.sciencedirect.com/science/article/pii/S004016257180001X.

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

[10] Nils Gundersen and Nils H. Lundberg. Petroleumsutvinning. 2021.
URL: https://snl.no/petroleumsutvinning.

[11] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.
URL: http://arxiv.org/abs/1803.10122.

[12] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels, 2018.
URL: https://arxiv.org/abs/1811.04551.

[13] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[14] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models, 2020.
URL: https://arxiv.org/abs/2010.02193.

[15] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models, 2021.
URL: https://ai.googleblog.com/2021/02/mastering-atari-with-discrete-world.html.

[16] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.
URL: https://arxiv.org/abs/1710.02298.

[17] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari, 2019.
URL: https://arxiv.org/abs/1903.00374.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.
URL: `https://arxiv.org/abs/1412.6980`.

[19] Oleg Klimov. Car racing.
URL: `https://www.gymlibrary.dev/environments/box2d/car_racing/`.

[20] Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are sample efficient world models. *arXiv preprint arXiv:2209.00588*, 2022.

[21] Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

[22] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016. doi: 10.48550/ARXIV.1602.01783.
URL: `https://arxiv.org/abs/1602.01783`.

[23] NORCE. Openlab user guide, 2022.
URL: `https://openlab.app/user-guide/`.

[24] Jurgis Pasukonis. Pydreamer, 2022.
URL: `https://github.com/jurgisp/pydreamer/tree/525eb64c128515a9d169fb6f41b7093452df879f`.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[26] A. Richards and J. How. Robust model predictive control with imperfect information. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 268–273 vol. 1, 2005. doi: 10.1109/ACC.2005.1469944.

[27] D. Rumelhart, G. Hinton, and Williams R. Learning representations by back-propagating errors. 1986. doi: https://doi.org/10.1038/323533a0.
URL: `https://www.nature.com/articles/323533a0`.

[28] Nejm Saadallah, Jan Einar Gravdal, Robert Ewald, Sonja Moi, Adrian Ambrus, Benoit Daireaux, Stian Sivertsen, Kristian Hellang, Roman Shor, Dan Sui, Stefan Ioan Sandor, Marek Chojnacki, and Jacob Odgaard. Openlab: Design and applications of a modern drilling digitalization infrastructure, 05 2019.
URL: https://doi.org/10.2118/195629-MS. D011S002R002.

[29] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, dec 2020. doi: 10.1038/s41586-020-03051-4.
URL: https://doi.org/10.1038%2Fs41586-020-03051-4.

[30] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.
URL: https://arxiv.org/abs/1506.02438.

[31] David Silver. Rl course by david silver - lecture 2: Markov decision process, 2015.
URL: https://www.youtube.com/watch?v=lfHX2hHRMVQ.

[32] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

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

[34] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
URL: http://incompleteideas.net/book/the-book-2nd.html.

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

[36] Nino Vieillard, Olivier Pietquin, and Matthieu Geist. Munchausen reinforcement learning, 2020.
URL: https://arxiv.org/abs/2007.14430.

[37] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696.
URL: https://doi.org/10.1007/BF00992696.