Out-of-Distribution Dynamics Detection in Deep Reinforcement Learning

Ivalin Chobanov

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

Detecting Out-of-Distribution Dynamics (OODD) in Deep Reinforcement Learning (DRL) is a problem with significant importance for real-world applications employing DRL agents. Failure to detect OODD could lead to serious problems or even result in life-threatening scenarios. For example, an autonomous vehicle could crash due to its inability to drive in an environment with OODD. So far, research in the area has focused on noise-free environments with deterministic transitions. The purpose of this project is to develop a framework that can operate in non-deterministic and noisy environments. Our results demonstrate that the developed framework is capable of detecting OODD in both noisy and non-deterministic environments. This makes our framework a step towards developing OODD detection methods for noisy, non-deterministic environments.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ivalin Chobanov*)

Acknowledgements

I would like to thank Dr. Stefano Albrecht for giving me the opportunity to work on this project and for his invaluable supervision. His feedback has been pivotal in ensuring the dissertation was on the right track.

I am also grateful to Elliot Fosong and Trevor McInroe for discussing with me various ideas regarding the project, suggesting improvements and providing invaluable feedback. Their assistance has been instrumental in the realization of this project.

Table of Contents

1 Introduction **1**

2 Background material **4**

2.1 Brief introduction to RL 4

2.2 OOD detection in general 6

2.3 OOD detection in DRL 7

2.4 OODD detection 8

2.5 Modelling the environment dynamics 10

3 Methodology **12**

3.1 Motivation 12

3.2 Framework 14

3.2.1 Dynamics module 15

3.2.2 Inference masks module 18

3.2.3 Detectors module 20

4 Experiments and results **25**

4.1 Experiments with noise and perturbations 25

4.1.1 Purpose and setting of the experiments 25

4.1.2 Results 27

4.2 Experiment with a non-deterministic environment 31

4.2.1 Purpose and setting of the experiment 31

4.2.2 Results 32

5 Conclusions **35**

Bibliography **39**

A Hyperparameters for the frameworks used in the conducted experiments 46

A.1 Hyperparameters description 46

A.2 Hyperparameters of the framework used in the experiments with the MountainCar environments 47

A.3 Hyperparameters of the framework used in the experiments with the CartPole environments 48

A.4 Hyperparameters of the framework used in the experiments with the Grid World environments 49

Chapter 1

Introduction

Modern Reinforcement Learning (RL) could be traced back to the late 1980s when three different areas of research came together: optimal control theory, learning by trial and error, and temporal-difference methods [60]. RL became a paradigm for solving various challenging problems at the time, such as, playing games [5, 61], autonomous vehicle control [1, 44] and robotics [42, 59]. However, RL had limited scalability and it was not well suited for problems with high-dimensional state space due to the memory complexity, computational complexity and sample complexity of the algorithms at the time [4]. Deep Reinforcement Learning (DRL) overcomes these limitation by combining RL with Deep Learning techniques which can represent high-dimensional spaces in a compact manner. The power of DRL was first demonstrated by DeepMind when they developed the Deep Q-Networks (DQN) algorithm which showed a remarkable performance on several Atari games [45]. Since, DRL has become a popular area of research within Artificial Intelligence and a powerful instrument for problem-solving. Programs like AlphaGo [58] and OpenAI's Five [9], both powered by DRL, have made headlines in the news as being capable of defeating the best human players in the world in their respective games. Autonomous driving algorithms use DRL extensively to learn how to drive safely [34]. A new matrix multiplication algorithm was discovered by AlphaTensor [22] which is powered by DRL. Recently, DRL has also been used to tune large language models [10].

In the context of Deep Learning, we often assume that the training data is representative of the data at inference time and as such, the training data is sufficient for the model to learn the necessary patterns. For example, in a number plate recognition system meant for recognising number plates containing only digits, we might chose to deploy a digit recognition model. In this case, the model would be trained only with data containing digits and we would expect all the input images to be of digits. However, a problem arises when a number plate containing a letter is to be recognized. In this case, the model would classify incorrectly the letter as a digit which might be undesirable. A better system would be one that can detect that the input is not a digit and send the number plate to be inspected by a human. In this example, we can say that the letter at inference time does not come from the underlying distribution of the training data which consist of digits only. In general, data that does not conform to a specific distribution is referred to as out-of-distribution (OOD) with respect to the training data distribution. The task of detecting OOD data is known as OOD detection which is a thoroughly researched problem in many domains such as time series forecasting and image classification [3, 6, 15, 28, 38].

In RL, data comes in the form of observations from the environment. In theory, because these observations are produced by the same environment dynamics both during training and inference, the distributions over states induced by a given policy are the same. Factors like noise can cause observations to appear extreme or unlikely to have been generated by the underlying distribution of the training data in which case these observations would be considered OOD. An interesting scenario is the case where a change occurs in the environment dynamics.

As the underlying distribution of the data is dictated, in part, by the environment dynamics, changes in the dynamics will lead to changes in this distribution. Therefore, the problem of detecting changes in the environment dynamics is an OOD detection problem referred to as OOD dynamics (OODD) detection. Failing to detect OODD could result in bad consequences. Consider an example of an autonomous driving system that was trained on good road conditions: clean dry roads on a sunny day. In that case, it may not be safe to assume that the system can drive autonomously if the conditions change - e.g. the road is covered with ice and snow. In such a scenario the autonomous vehicle should be more cautious and drive slowly, especially when taking turns. If it takes a turn with the same speed as if the conditions were good, that could lead to the car sliding off the road and hitting a tree. In this case, using this system when the dynamics of the environment changes is a health hazard.

Another example is a trading bot that is trained to trade stocks. If the dynamics of the markets changes significantly as a result of certain local or global events, or the introduction of new bots with different trading strategies, then the trading bot might start making trades that result in a financial loss which would be undesirable. In both of the examples above we would have preferred to detect when the dynamics of the environment changed and handle these cases separately, for example, yield the control of the vehicle to a human and stop trading.

So far, there has been limited work in the area of OODD detection and previous work has focused on detecting OODD through the states or state-action prediction error in noise-free environments [17, 26]. Instead, the focus of this research was to build a framework for OODD detection capable of operating in noisy environments. Furthermore, [17,26] had only focused on deterministic environments. Hence, another objective of the research was to develop a framework capable of operating in non-deterministic environments. In the end, the results of the experiments demonstrate that the developed framework is capable of OODD detection in noisy and non-deterministic environments, making this research a step towards developing methods for OODD detection in noisy and non-deterministic environments.

This document is structured as follows. In Chapter 2 we introduce helpful background material. In particular, a brief introduction to RL is provided in 2.1 followed by an introduction to general OOD detection (2.2) and OOD detection in RL (2.3). The chapter is concluded with an overview of previous work in OODD detection in Section 2.4 and dynamics models in Section 2.5. Next, we discuss the methodology in Chapter 3. In particular, in Section 3.1 we discuss some of the limitations of the previous research and how the developed framework aims to solve these. Then, in Section 3.2 the developed framework is introduced and details are discussed. In Chapter 4 we introduce the experiments conducted with the framework and provide the achieved results. Finally, in Chapter 5, we discuss the achieved results, some limitations of the developed framework and discuss further improvements and research that can be conducted as future work.

Chapter 2

Background material

This chapter contains some background material helpful for understanding how the developed OODD detection framework works and how it fits in the context of OODD detection and DRL.

2.1 Brief introduction to RL

RL is a subdivision of Machine Learning (ML) that is closely related to decision-making. In its simplest form, an RL problem is comprised of an environment and an agent that interacts with the environment. The goal of the agent is to learn how to optimally complete a given task through interactions with the environment.

There are different ways to model the environment. The two most popular ways are the Markov Decision Process (MDP) [7] and the Partially Observable Markov Decision Process (POMDP) [33]. The MDP is characterized by the states S and actions A spaces, the reward function R and the environment dynamics:

$$p(s',r|s,a) = P\left\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\right\}. \tag{2.1}$$

Equation 2.1 is the result of the Markov property which is an important assumption that underpins MDPs. It states that the future state depends only on the current one, and not on the past states. The agent is characterized by a policy $\pi(a|s)$ which gives a probability distribution over actions conditioned on a given state. To learn how to perform optimally on a specific task, the agent interacts with the environment to receive a reward which is used towards computing an expected return. The goal is then to find a policy under which the expected return will be maximized. The return is defined as:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T = R_{t+1} + G_{t+1} \tag{2.2}$$

In practice is is more common to work with a discounted return, defined as

$$G_t = \sum_{k=0}^{T-(t+1)} \gamma^k R_{t+k+1}, \tag{2.3}$$

where γ is the discount factor. This is used to ensure that the return is finite.

A state value function is defined as the expected return of a particular state under a specific policy π:

$$v_\pi(s) = \mathbb{E}\left[G_t | S_t = s\right]. \tag{2.4}$$

Hence, the goal of the agent is to find an optimal policy π^* for which the value function is maximized.

Often in the real world, the agent cannot observe the current state of the environment directly. Instead, the observations can be noisy or incomplete. Such scenarios in RL are modelled as a POMDP. Because the agent does not have a direct observation of the environment's true state, its policy cannot map directly from states to actions. Instead, the agent forms a belief about the current state. That is achieved through keeping a history of the states or through a probability distribution over the states. When the agent observes a state, it updates its belief about the real state and maps that to an action.

In general, there are two types of RL: model-based and model-free. In the former, a model of the environment $\hat{p}$ is learnt and used for planning, whereas in the latter case, no such model is explicitly used in finding the optimal policy.

DRL is responsible for much of the recent success of RL. As mentioned earlier, DRL is a subfield of RL where Deep Learning methods are used. In general, DRL utilizes Deep Learning architectures for parameterizing functions that approximate states and actions. This is especially useful in cases where the state and action spaces are large and complex so alternative methods would be incapable of solving the tasks.

2.2 OOD detection in general

As discussed before, OOD detection is the problem of detecting data with an underlying distribution that is different from the underlying distribution of the training data. This problem is closely related to problems like anomaly detection (AD), novelty detection (ND) and outlier detection (OD) [15]. In AD, the model is concerned with finding abnormal instances. Abnormalities are often predefined or are assumed to be rare data instances or whole categories of data. For example, in time series data, a point that has a value that is extreme compared to surrounding points is considered to be anomalous and would be detected as such by an AD model. ND deals with instances that belong to a new class unseen during training. For example, if we train a model to recognise images of cats and dogs, ND methods would detect all images that are not of cats or dogs. OD is a more broad form of AD and detects all points that are significantly different from the remaining points in a sample of data.

Despite the differences between these problems, in all cases, we can identify in-distribution (ID) and OOD classes and the points belonging to these. Hence, techniques from these problems are applicable to OOD detection. [65] recognizes four general categories of OOD detection methods: classification-based, density-based, distance-based and reconstruction-based. Generally, in classification-based methods, the softmax probability is utilized to determine if data belongs to the ID or the OOD class [20,29,30]. In density-based methods, data points are considered to be OOD if their probability is low according to a probability distribution of the ID data [32,53,55]. Distance-based methods utilize clustering approaches. The distance between the data point and clusters is used to determine whether a point is OOD [13,54,62]. Reconstruction-based methods reconstruct data from a compressed representation. These models assume that the reconstruction error for OOD data will generally be higher [66,67]. Furthermore, in time series data, there are also prediction-based models [18]. These models predict future observations over a given horizon and then compare that to the actual observation. The error between the two is used as the OOD decision factor [21,47].

OOD detection is also related to one-class classification [52] as all of the training data is assumed to belong to the ID class and the inference data may contain points belonging to the OOD class. All one-class methods have one thing in common. They calculate an anomaly score for the data and decide if a point is OOD based on a threshold which is normally determined via validation data [52]. The anomaly score could be based on the distance or the magnitude of the error, depending on the model used.

2.3 OOD detection in DRL

Some of the OOD research in DRL has been focused on developing algorithms and exploration strategies robust to OOD data. As such, these algorithms implicitly address the problem of OOD detection. The authors in [64] develop a model that implicitly detects OOD state-action pairs and penalizes their contribution to the model training through regularization. To achieve this, the model uses Monte Carlo dropout [23] both during training and inference. The variance in the predictions in T stochastic forward passes is used as the regularization parameter. In [2], the authors penalize Q-value functions for state-action pairs that are deemed OOD. For that, an ensemble [36] of Q-value functions is used to estimate the epistemic uncertainty and during exploitation, the lower-confidence bound of these estimates is used which penalizes OOD state-actions. The proposed method in [39] utilizes bagging as a method for estimating the epistemic uncertainty where a bag of Gaussian dynamic models is used to calculate a statistic. The authors argue that a higher value of that statistic is indicative of an OOD state. They then further devise a probability function using the statistic which is used for regularization.

Other work has focused on explicitly detecting OOD states and state-action pairs. The UBOOD framework [56] defines OOD state-acton pairs to be the state-action pairs for which the estimated epistemic uncertainty is high as during training, the epistemic uncertainty for ID state-action pairs is reduced. The authors evaluate three different methods for estimating the epistemic uncertainty - Monte Carlo Concrete Dropout (MCCD) [24], bootstrapping [51] and Bootstrap-Prior networks [50]. In the MCCD case, the model is run T times to collect a sample of predictions and the uncertainty is calculated as the variance in the predictions. For the bootstrapping and the Bootstrap-Prior network cases, the models are comprised of shared input and intermediate layers and K output layers called “heads”. To calculate the uncertainty, K boolean masks are generated which select the output heads for each data point. The uncertainty is calculated as the variance of the K outputs [56]. To calculate a threshold for the OOD data, the authors in [56] treat the uncertainty of the ID data (training data) as a probability distribution. The threshold is then set to $\tau = \bar{u} + \sigma(u)$ where $\bar{u}$ is the mean of the calculated uncertainties of the training data and $\sigma(u)$ is the standard deviation of these uncertainties. A state-action pair is considered to be OOD if its uncertainty is greater than τ.

In [57], a classifier is developed that utilizes the entropy of the policy to detect OOD states. The authors hypothesize that the entropy of the distribution of the actions is reduced for states encountered during training (the ID states). They state that the entropy of the policy for ID data should be less than the entropy of the policy of OOD data given a successful training procedure. Hence, a decision boundary could be found that separates the ID and OOD states, however, in practice, this decision boundary would be soft and a perfect separation will not be possible [57].

2.4 OODD detection

The frameworks like [56] and [57] detect OOD states or state-action pairs. Such are considered to be the states or state-action pairs that have not been observed during training. The problem of OODD detection is different in nature. Here, we are interested in detecting changes in the environment dynamics and not exclusively unseen states. For example, in a simple grid world environment, we might observe all states during training. Hence, a change in the dynamics in this simple grid world could be associated with a change in the transitions dynamics from one state or state-action pair to another state even if all states have been previously observed.

The authors of [46] propose changes to RL environments that can be used to generate OOD data. The proposed changes are of two types: changes to the physical parameters of the environments and the addition of corruptions to the observations (noise). The former type closely resembles changes to the environment dynamics as the dynamics in these environments is dictated by these parameters. Observational noise is not generally associated with changes to the environment dynamics as in most real-world scenarios observation noise is present when the dynamics of the environment is ID. However, changes to the observation noise could potentially be considered as changes in the environment dynamics as increasing the observation noise could have a negative impact on the agent's ability to perceive the environment and make good decisions. In [46], no method for OODD detection was developed. Instead, the authors tested [56] with the proposed benchmarks.

The term "Out-of-Distribution Dynamics Detection" is mentioned for the first time in [17]. The authors of the paper develop a model for OODD detection and a set of benchmarks to evaluate the model on. The proposed benchmarks include the addition of noise and drift but also changes to some of the properties of the environments. The proposed model, Recurrent Implicit Quantile Network (RIQN) is based on the Implicit Quantile Network (IQN) [16] which is a generative model that can learn an implicit representation of the distribution of features and consequently, be used to generate samples of this distribution. The authors in [17] further extend the IQN model with a Recurrent Neural Network (RNN). The purpose of the RNN is to retain information from past observations and use it for the purpose of predicting future states. The model is also capable of predicting several steps ahead autoregressively. To prevent the accumulation of error due to the autoregression, the authors of [17] use a Sampling Scheduling approach [8]. The RIQN model accounts for the aleatoric uncertainty [17] which is the inherent randomness of the data. To account for the epistemic uncertainty, which is the uncertainty of the model due to the lack of training data, the authors use an ensemble of RIQN models. To compute an anomaly score for a time point t, a set of M autoregressive predictions are made for each model in the ensemble of size e. The anomaly score is then computed as the average L1 distance between the prediction and the true observation among the $M \times e$ predictions for time point t. During inference, a cumulative sum (CUMSUM) [17] of the anomaly scores across time and a threshold based on the CUMSUM are used for OODD detection. The authors of [17] also test a modified version of the RIQN model that includes a history window, however, they do not find it beneficial for the RIQN model.

Another framework for OODD detection is developed in [26]. It is comprised of a forward dynamics model, next-state sampling, error calculation and anomaly scoring. The forward dynamics model the authors use is based on Probabilistic Deep Neural Networks which parameterize a probability distribution. The authors of [26] choose to use a Gaussian distribution to model the next state. Hence, the forward dynamics model takes the current state and action and outputs the mean and variance of the distribution of the next state. The authors note that the probabilistic model models the aleatoric uncertainty but not the epistemic uncertainty. To model the epistemic uncertainty, the authors used bootstrapped ensembles where each member of the ensemble of size B is initialized with different parameters and is trained on a different bootstrap of the training set [26]. Next state sampling is used to sample states from the Gaussian distribution that is parameterized by the forward dynamics model. The authors refer to these samples as particles. Each member of the ensemble generates K particles. The error between the predicted state and the observed state is computed as the Mean Squared Error (MSE) of all particles in each ensemble. In the end, an anomaly score for each time point is calculated using an aggregation function over the computed prediction errors of all members of the ensemble. The authors find that the minimum works best for the aggregation function. During inference, the anomaly score is compared to a threshold which is calculated via validation data. All predictions for which the anomaly score is greater than this threshold are considered to be OOD. The authors consider different methods for calculating the threshold such as the mean and maximum anomaly score of the validation set.

2.5 Modelling the environment dynamics

Usually in RL, access to the true environment dynamics is not provided. Instead, the true environment dynamics is hidden and the agent can only receive observations from the environment based on the actions it takes. It is not uncommon, especially in model-based RL, to learn a dynamics model of the environment which can then be used to generate observations by providing it with actions. As we saw with [17] and [26], in OODD detection, if we learn a dynamics model that can approximate the environment dynamics, we can then assume OODD when the prediction error of this model is significantly high. The dynamics models in [17], [26] and [14] are examples of probabilistic dynamics models that map a state or state-action to a probability distribution of the next state. Models can also be deterministic. Such models do not generate a probability of the next state but instead, the output is the predicted observation. Such models are developed in [49] and [48]. Furthermore, the dynamics models in [26], [14], [49] and [48] only use the current state or state-action to predict the next state without a history of observations. In contrast, the dynamics models in [25], [37] and [43] are examples of dynamics models that use context (history), usually through a latent representation generated by an RNN.

Other models that are capable of modelling the environment dynamics are sequential modelling models based on the Transformer [63] architecture. Example of these include Decision Transformer [12] and Trajectory Transformer [31]. These models are utilized in the next token prediction task where they predict the next token autoregresssively.

The tokens can be observed states, actions, rewards or a combination of these. Another idea explored in this area is a BERT-style masking. BERT [19] is a popular model for natural language understanding. It resembles a bidirectional Transformer encoder which is pretrained and can be fine-tuned for specific tasks. One of the training objectives of BERT is masked token prediction where some of the tokens in the sentence are masked and the model tries to predict them using the unmasked tokens. This helps the model learn how to understand the context of the tokens. Both UNI[MASK] [11] and MaskDP [40] have utilized BERT-style masking in their sequential decision models and demonstrate that this makes them viable dynamics models.

In particular, UNI[MASK] focuses on building a generalized model for sequence modelling that can be used for tasks such as forward dynamics, inverse dynamics, future inference, past inference, behavior cloning, etc. Different masking schemes are investigated in [11] and the authors find random masking to be most effective for training. For each sample trajectory, a masking probability is uniformly randomly generated and then each token is masked with that probability. During inference, depending on the specific task at hand, a specific mask is used that prompts the model to fill in the required tokens, which are generated by stacking states, actions and rewards.

The main concept of MaskDP [40] is similar to the concept of UNI[MASK] [11], namely, the model is a bidirectional transformer that is trained on the task of masked prediction. However, MaskDP focuses on achieving high performance on three tasks: goal reaching, skill promoting and serving as a model for offline RL. The input sequence is comprised of alternating states and actions. Furthermore, the MaskDP model has an encoder that encodes only the unmasked states and actions and then a decoder that uses these encodings and the masked tokens to predict the masked tokens. In [40] the masking ratio for a trajectory is sampled from a set.

Chapter 3

Methodology

We begin by discussing some of the issues and limitations of previous research in the field and discuss how the developed framework aims at tackling these challenges. Afterwards, the framework is introduced and all of its components are discussed in detail.

3.1 Motivation

There are several limitations in the work in [17] and [26]. First, both publications focus on developing prediction-based models for MDP environments. Neither of the developed frameworks in these works has been tested in environments that are partially observable. In fact, the framework in [26] is not capable of operating in such an environment. That is due to the fact that the prediction-based model the authors have developed uses the observation of the current state and the action taken by the agent to predict the next state. There is no notion of memory meaning that it is unlikely for the model to be successful in a POMDP environment. The framework in [17] utilizes an RNN to encode information about the past state, so in theory, it is capable of working with POMDPs as having a notion about the past helps the model filter out noise and make more accurate forward predictions.

Furthermore, previous work has only focused on deterministic environments. However, stochastic environments are also common. The model in [26] assumes that the next state distribution is Gaussian. For non-deterministic environments, this model would only work if the stochasticity is Gaussian. Hence, this model is not appropriate for non-deterministic environments due to the assumption of the next state distribution. The model in [17] does not make an assumption of the next state distribution. This is why it makes M predictions for the next state in order to represent the predictive distribution [17]. Let's assume that according to the non-deterministic dynamics, there are N possible next states. However, only one of the N possible states would be observed at time $t+1$ if the dynamics is ID. Furthermore, only a portion of the M predictions would have correctly predicted the next state, with the remaining predictions contributing to the total prediction error. Depending on the next state distribution, it is possible that this prediction error is high enough for the model to label the next state as OOD. However, the state would not be OOD according to the non-deterministic dynamics. Hence, prediction-based models that utilize prediction error are limited by the sample of predictions they make. Generally, if the prediction error between this sample and the actual observations is low, the next state would be assumed ID otherwise OOD. This is why the model in [17] might not be very suitable for non-deterministic environments.

Another limitation of the work in [17] and [26] is the method of calculation of the OOD threshold. Neither work focuses explicitly on this method, but rather uses a property of the validation error. In [26], the authors test different methods for calculating the threshold including maximum validation error. However, neither work takes into consideration the shape of the error distribution.

Finally, a limitation of models that parameterize probability distributions of the next state is that they have to make assumptions about that distribution as is the case in [26] where the next state is assumed to have a Gaussian distribution. As mentioned earlier, this would be a problem if the environment dynamics is non-deterministic. However, it could also be a problem in deterministic environments as the true next-state distribution might be significantly different from the parameterized distribution. This is why assumptions regarding the next state distributions should generally be verified.

The framework designed in this project tackles the limitations mentioned above. By design, the developed framework is reconstruction-based and operates with trajectory snippets where the snippet is a fixed-size window of consecutive observations from the environment. The framework reconstructs parts of the snippet and uses the reconstruction error as an indication of OODD. This means that the framework is not limited by a sample of predictions but by the quality of the reconstruction. Provided that the dynamics model in the framework can understand the environment dynamics well, we

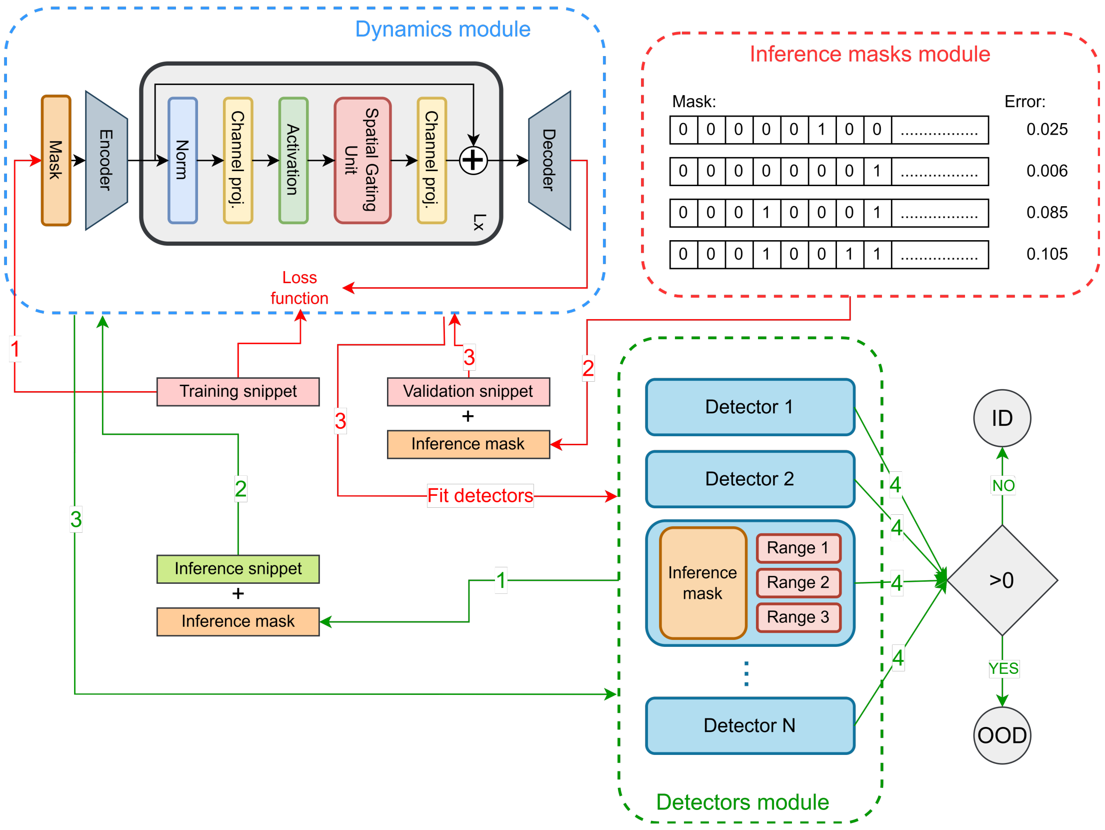

Figure 3.1: Diagram of the developed OOD Dynamics Detection framework. The red arrows follow the training procedure and are numbered in the order of the operations. The green arrows follow the framework at inference time. The numbers denote the order of operations. Dashed contours separate the different modules within the framework - the Dynamics module, the Inference masks module and the Detectors module.

can assume that the reconstruction error would be low for snippets with ID dynamics and higher for snippets with OOD dynamics. However, setting a threshold based on this assumption might be limiting, hence, the framework also takes the shape of the distribution of the reconstruction error into account when determining the thresholds for OODD detection.

3.2 Framework

The developed framework is shown in Fig 3.1. It operates with trajectory snippets. As mentioned previously, the snippet is a fixed-size window of consecutive observations from the environment. Mathematically, this could be expressed as follows:

$$s = O_1, O_2, ..., O_{\Delta-1}, O_\Delta, \tag{3.1}$$

where s denotes a snippet that is of size Δ and O_i denotes the i^{th} observation in the snippet. That means that the framework takes a snippet s as input and determines if the dynamics inside the snippet is OOD or not. The framework is comprised of three components: a Dynamics module, Inference masks module and a Detectors module. In the following sections, each of these components is explained in detail. Note that in the following sections, snippet and trajectory are used interchangeably.

3.2.1 Dynamics module

The Dynamics module is inspired by the UNI[MASK] [11] and MaskDP [40] models. It consists of an encoder, masking utility, a dynamics model and a decoder. The purpose of this module is to reconstruct a masked trajectory. The input to the module consists of trajectories of observed states and optionally actions. The output consists of trajectories of predicted states.

The module is designed to work with both continuous and discrete states and actions. Any discrete states and actions are first one-hot encoded. Then, the state and action embeddings are concatenated if actions are also part of the input. The combined embedding is then masked. Continuous states and actions are represented by embeddings. These embeddings are vectors of size equal to the number of features of the states or actions.

Each trajectory is comprised of two parts: a context window and a reconstruction window. The context window provides a history of observations that the model can use to help predict the masked tokens in the reconstruction window. This is why when masking, the embeddings of the observations in the context window at the start of the trajectory are always kept unmasked so that they are visible to the model. During training, random masking is applied to the embeddings in the reconstruction window whereas, during inference, a mask is provided by the Detectors module instead of randomly generating one.

To generate the masks for training, a masking ratio is randomly sampled from a set of masking ratios, similarly to [40]. Using this ratio, the appropriate number of 1s (masked) and 0s (unmasked) are mixed and shuffled to create a boolean mask for the reconstruction window. A mask of 0s for the context window is then added at the start of this mask to form the final mask for the trajectory.

The mask is applied to the embeddings in the trajectory as follows. First, all embeddings for which the value in the mask is 1 (meaning they are to be masked) have all entries set to 0. Next, an extra dimension is added to all embeddings. The new entry is set to the value of the corresponding entry in the mask for that embedding, meaning that a masked embedding would become a vector with all entries equal to 0 except the last one which equals 1. The idea is that the last entry serves as means to distinguish a masked embedding from an embedding that has all its entries set to 0.

Having applied masking to the embeddings, all embeddings are then encoded using a linear encoder. This transforms the embeddings into embeddings of a specified size. This can enhance the feature representation and also reduce the dimensions in the case of discrete input features. After this operation, the input for the dynamics models consists of trajectories of embeddings of a specified number of dimensions.

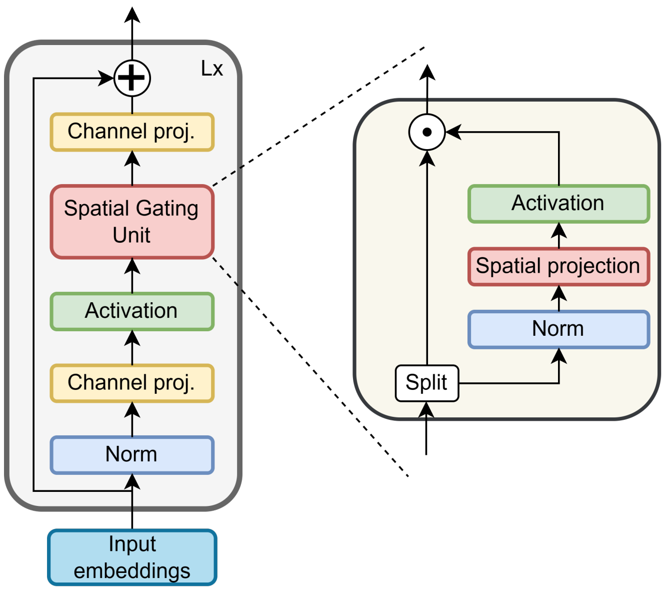

Figure 3.2: Architecture of the gMLP model. The network is comprised of a stack of L identical blocks. All projection operations are linear. The activation function used in the gMLP block is GeLU. Also shown is the architecture of the Spatial Gating Unit (SGU). The activation inside the SGU is SELU.

Unlike the architecture in [11] and [40], the architecture of the dynamics model here is not a Transformer. Instead, a Gated Multilayer Perceptron Network (gMLP) [41] is used, the architecture of which is shown in Fig 3.2. In the original publication [41], the authors demonstrate that the gMLP is capable of achieving performance comparable to the Transformer in vision and language tasks. In fact, the authors compare the gMLP with BERT and find that on some Natural Language Processing (NLP) tasks, gMLP outperforms BERT. As seen in Fig 3.2, the architecture is comprised of L stacked layers. First, the input is normalized before passing it through a linear projection layer that applies the linear transformation to the embeddings. The projected output is then passed through an activation function. The used activation function is GeLU [27]. The output is then processed by the Spatial Gating Unit (SGU) [41] before it is transformed back to the original dimensions of the input to the gMLP block. Finally, the output of the linear projection layer is added to the input to the gMLP block via a residual connection. Here, the SGU is what makes this architecture different from ordinary neural networks.

The SGU resembles the output of linear gating and provides the model with similar power to self-attention in the Transformer architecture, namely, cross-token communication [41]:

$$s(X) = X \odot f_{W,b}(X), \tag{3.2}$$

where $s(\cdot)$ is the SGU, $\odot$ is the Hadamard product and $f_{W,b}(\cdot)$ is a linear projection. This linear projection projects the input using a matrix $W \in \mathbb{R}^{n \times n}$ where n is the length of the sequence in the input, and a bias $b \in \mathbb{R}^n$:

$$f_{W,b}(X) = WX + b. \tag{3.3}$$

Hence, the spatial projection matrix W takes the role of the self-attention matrix, with the difference that it is static and not dynamically calculated based on the input like in the Transformer. In [41], the authors further find it beneficial to split the input X into two parts X_1 and X_2 along the dimension of the token embedding. Further to this, we also add an activation function after the linear projection to introduce non-linearity. The activation function used is SELU [35]. Hence, the gating function becomes

$$s(X) = X_1 \odot \phi(f_{W,b}(X_2)), \tag{3.4}$$

where $\phi(\cdot)$ is the SELU activation. The output of the function in Equation 3.4 provides the output of the SGU. One other difference between gMLP and the transformer architecture is that the gMLP requires no positional embeddings as the SGU is able to capture that information [41].

Once the embedded trajectories are encoded, they are fed through the dynamics model which produces the reconstructed trajectories. Finally, the reconstructed trajectories are decoded using a decoder which puts the embeddings back in the original dimensions of the observations.

The training loss function for the dynamics model depends on the input type. For continuous state inputs, Mean Squared Error (MSE) is used. For discrete state inputs, Cross-Entropy Loss is used instead. The loss is computed on the entire trajectory, including the embeddings on unmasked positions. This ensures that the model is capable of reconstructing the trajectory as a whole which leads to a better understanding of the dynamics inside the trajectories, similarly to [40].

3.2.2 Inference masks module

This module generates inference masks which are the masks that are provided to the Dynamics module at inference time. These masks have two purposes.

First, they provide a way to use the reconstruction error of the dynamics model as a means of OODD detection. Consider a specific mask that we can use with the Dynamics module and validation data. Then, we can compute the distribution of the errors in the validation data given this mask. At inference time, we can use the same mask to compute the error in the trajectory and compare that to the distribution of errors for the validation data. An extreme error could be considered a sign of OODD. If random masks are used for inference, we would have to learn the distribution of errors for all possible masks. That would be difficult as the number of possible masks increases with the size of the reconstruction window. In that case, we may have to resolve to use one distribution of errors independent of a mask. That, however, may lead to inaccuracies and bias which are the result of the different distribution of the errors between the masks. Knowing the distribution of errors on a particular mask makes it easy to check if the error for a trajectory is extreme given the mask.

Second, the inference masks can help with non-deterministic environments. In Section 3.1, we mentioned that the reconstruction-based approach is not disadvantaged in terms of non-determinism. Consider a non-deterministic environment and a mask where all tokens in the reconstruction window are masked. That in a way makes the model behave like a prediction-based model as the only provided states are in the context. In that case, there might be multiple ways of filling in the masked states and the reconstruction error could be high, depending on the actual observation. If we unmask one entry somewhere in the reconstruction window, then the dynamics model is forced to predict a trajectory that would fit around that unmasked state. Hence, the unmasked state constrains the prediction the dynamics model can make.

Unmasking many states in the reconstruction window would constrain the model predictions more but could also lead to more accurate predictions when the dynamics is OOD due to the generalization power of the model. Predicting a longer sequence of masked tokens would be more difficult for a trajectory with OODD. Hence, the inference masks have to be somewhat balanced: they should contain a low number of unmasked positions in the reconstruction window while still having enough unmasked states to constrain the predictions for non-deterministic environments.

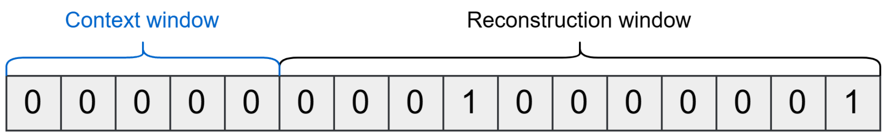

Figure 3.3: Visualization of an inference mask of a trajectory. The context window is used as a history and all positions are unmasked (value 0). It is of length 5 in the example. The reconstruction window contains unmasked (0s) and masked positions (1s) and its length is 11 in the example. The length of the snippet is 16.

The proposed module works as follows. It generates a certain number of masks per number of unmasked positions in a range. This range is defined by the minimum and maximum number of unmasked positions which are controlled through hyperparameters. For example, if we do not expect any non-determinism, we may use just a few unmasked positions, otherwise, we might want to use more to greater constrain the predictions. Given the range of the number of unmasked positions, we could expect that masks with more unmasked positions will potentially have a lower error as the higher number of unmasked positions would provide extra information for the reconstruction. To promote masks with fewer unmasked positions, a regularization of the following form is used to adjust the mask error:

$$\varepsilon_{\text{adj}} = N^{\beta}\varepsilon, \tag{3.5}$$

where ε is the reconstruction error on the validation data, N is the number of unmasked positions for the given mask and β is the regularization parameter. If $\beta = 0$, then the error is not adjusted. If $0 < \beta < 1$, the mask error is adjusted. This regularization forces masks with more unmasked positions to have lower errors in order to be selected. In the end, the top M masks with the lowest ε_{adj} error are selected. These masks are then provided to the Detectors module where a detector is created for each mask. A visualization of an inference mask can be seen in Fig 3.3. The Inference masks module only runs during training and is not used during inference.

The reconstruction error here is calculated with the same loss function as the loss function used by the Dynamics module. However, the error here is only computed on the masked observations inside the trajectory and not on the entire snippet. This ensures that the masks are compared fairly as otherwise, masks with more unmasked states would tend to have a lower error due to the the more unmasked positions.

3.2.3 Detectors module

The role of this module is to decide if the dynamics inside a given snippet is OOD at inference time. Commonly, this decision is based on a threshold derived from the anomaly score on a validation set. One common way of setting the threshold, similar to the method [56], is $\tau = \mu + k\sigma$ where μ is the mean validation score and σ is the standard deviation of the validation score. Another way of setting the validation score, used in [26], is to set it to some constant times the maximum validation score in the validation data. Both of these approaches can work well in specific scenarios. The former works well if the distribution of scores is approximately Gaussian. The latter one works best when the distribution of the validation scores is relatively compact and does not have a long right tail. The developed Detectors module here does not make assumptions regarding the distribution of the validation errors and instead uses the shape of the distribution of the errors to decide if a snippet contains OODD.

The Detectors module contains M detectors equal to the number of inference masks generated by the Inference masks module. Each detector is associated with one inference mask. During training, the generated inference masks are given to the Detectors module which fits a detector on each of the inference masks. The fitting operation involves computing the distribution of the reconstruction errors on the validation data given the inference mask, finding low-density regions in that distribution and saving intervals of errors that correspond to the OOD regions.

At inference time, the detector takes a snippet and sends the inference mask and the snippet to the Dynamics module which returns back the reconstructed trajectory. The error is then calculated similarly to how it is calculated in the Inference masks module. The calculated reconstruction error is then checked against the saved error intervals. If the error is found to be inside an OOD interval, the detector detects the dynamics inside the snippet as being OOD. Each trajectory is tested by every detector. In the end, if one detector detects the snippet's dynamics as OOD, the snippet is deemed to have OODD.

As mentioned earlier, the detectors make no assumptions regarding the distribution of the errors, hence, thresholds are not set via statistical properties of the distribution of the validation errors. Instead, a method is developed to detect low-density error regions in the distribution of the errors. Because few of the validation errors are located in the low-density validation error regions, the assumption is that if the error of a snippet at inference is located in these regions, then we can consider this error atypical and label the snippet OOD.

As explained earlier, the detectors in the Dynamics module find OOD intervals which are intervals of values of reconstruction errors that should be considered OOD. These intervals are of the form $[start; end]$ where $start$ and end denote the beginning and ending of the interval respectively. To find these intervals, the detectors first use an algorithm that finds the errors that are in the low-density regions of the distribution of the reconstruction errors in the validation data before using these errors to compute the OOD intervals. A pseudocode of this is found in Algorithm 1. To achieve this, we first sort the errors by their value. Then, the algorithm finds all errors in a window around error i such that this window contains the k closest errors to error i that are smaller and k closest errors to error i larger than it. The value of the error is calculated as the average difference between consecutive points in the region. The idea is that if an error is in a dense region, this value would be small. Contrary to this, if error i is in a low-density region, this value would be relatively high. If k is too small, then only immediate neighbors are considered and it would be difficult to estimate the density of the region. If k is too big, then errors in a low-density region can contain in their windows parts of a dense region which would skew the calculations. Instead, multiple values of k are used that range from 1 to max_win. max_win is calculated as $\frac{1}{2} \times rate \times len(errors)$, where

$rate$ defines the rate of the validation errors which are located in the low-density regions of the distribution of the validation errors. Having calculated the value of each error i for all $k \in 1,...,max\_win$, we combine these values by taking their average. The final value determines the density of the region error i is located in. High values indicate low density. In the end, we choose the top $2 \times max\_win$ errors with the highest values to define the OOD regions. Note that this corresponds to the chosen percentage of errors in low-density regions by the $rate$ hyperparameter.

Algorithm 1 Find errors that would define OOD intervals

```
Input: $rate$, $max\_horizon$, $errors$
  $v \leftarrow list$
  $max\_win \leftarrow 0.5 \times rate \times len(errors)$
  $errors \leftarrow sort(errors)$
  for $k$ in $range(1, max(max\_win, max\_horizon) + 1)$ do
    for $i$ in $range(len(errors))$ do
      $e \leftarrow$ all $k$ errors before and after error $i$ and error $i$
      $d \leftarrow$ average difference between two consecutive errors in $e$
      $v[i] \leftarrow$ the mean between previous values of $v[i]$ and $d$
    end for
  end for
  $ood\_errs \leftarrow$ the $2 \times max\_win$ errors with highest $v$
  $ood\_errs \leftarrow sort(ood\_errors)$ by original index in the sorted $errors$
Output: $ood\_errors$
```

Having found the errors in the low-density regions, the next step is to define the OOD intervals in the form $[start; end]$. Algorithm 2 defines these intervals. Essentially, the algorithm defines an interval to be a subsequence of consecutive errors in the low-density regions where these errors must be consecutive in the original list of sorted errors. After finding all errors in an interval, the $start$ is then defined to be the value of the smallest error in the subsequence and end is defined to be the value of the next largest error in the original sorted list of errors after the largest error in the subsequence. Some corrections are also made, such as, ensuring that the last interval ends at infinity. One correction for the calculated intervals is performed at the end of the algorithm where if the length of the interval is below min_dist then that interval is removed from the detector. This helps keep only significantly big intervals.

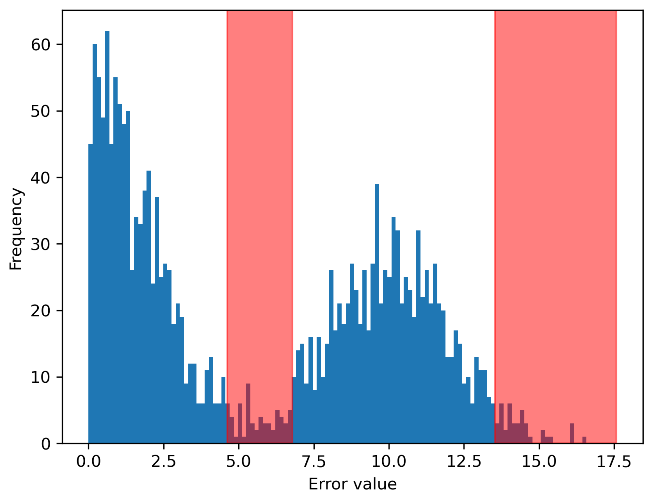

Figure 3.4: OOD intervals found by the algorithm (in red) using an OOD rate of 0.05. The Gaussian distributions used for the samples have means of 0 and 10 and standard deviation of 2. The absolute values of the samples have been used.

An example of the OOD intervals found by the Detectors module can be seen in Fig 3.4. There, the errors are generated by taking the absolute values of samples from two Gaussian distributions with means of 0 and 10 and a standard deviation of 2. The found OOD intervals are shown in red. You can observe that the method detects OOD intervals between the peaks of the two distributions where the density of errors is low, and at the tail of the Gaussian distribution with a mean of 10. Note, the last interval goes to infinity (displayed finite in the graph). This shows that the developed method of finding OOD intervals in the distribution of the errors takes into account the shape of the distribution.

Algorithm 2 Find OOD intervals

```
Input: $ood\_errs$, $min\_dist$, $errors$
  $intervals \leftarrow list$
  $start \leftarrow ood\_errs[0]$
  $end \leftarrow ood\_errs[1]$
  for $i$ in $range(1, len(ood\_errs))$ do
    if $ood\_errs[i].index == end.index + 1$ then
      $end \leftarrow ood\_errs[i]$
    else if $start.index == end.index$ and $end.index! = len(errors)$ then
      $start \leftarrow ood\_errs[i]$
      $end \leftarrow ood\_errs[i]$
    else
      if $end.index == len(errors)$ then
        $intervals.append([start.value, \infty])$
      else if $start.index == 0$ then
        $intervals.append([-\infty, end.value])$
      else
        $intervals.append([start.value, ood\_errs[end.index + 1].value])$
      end if
    end if
  end for
  if $end.index == len(errors) - 1$ then
    $intervals.append([start.value, \infty])$
  else
    $intervals.append([start.value, ood\_errs[end.index + 1].value]])$
    $intervals.append(ood\_errs[len(ood\_errors)].value, \infty)$
  end if
  for all $interval$ in $intervals$ do
    if width of the interval $< min\_dist$ then
      remove $interval$ from $intervals$
    end if
  end for
Output: $intervals$
```

Chapter 4

Experiments and results

4.1 Experiments with noise and perturbations

4.1.1 Purpose and setting of the experiments

This set of experiments aims to evaluate the framework's performance in environments where the dynamics is POMDP. It tests how the amount of change in the dynamics of the environment and the levels of observation noise affect the framework's performance.

These experiments are conducted in two classic control environments from Gymnasium: MountainCar and CartPole. In the MountainCar environment, a car sits at the bottom of the valley. The goal is to reach the top where the flag is located. See Fig 4.1. There are 3 possible actions that the car can take: accelerate left, accelerate right or do not accelerate. Accelerating to the right only does not solve the problem as the car does not have enough force to climb the hill. Instead, the agent has to learn how

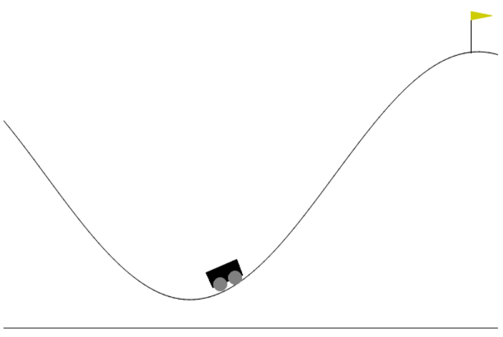
Figure 4.1: A frame in the MountainCar environment

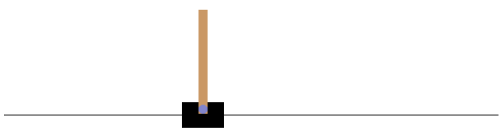
Figure 4.2: A frame in the CartPole environment

to gain momentum by driving back and forth. In the CartPole environment, there is a cart with an attached pole that sits upright. The pole is attached using an un-actuated joint. The goal for the agent is to balance the pole for as long as possible (within a time limit). To achieve this, the agent can take 2 actions: push the cart to the left and push the cart to the right. A frame of the visualized environment is shown in Fig 4.2.

Both MountainCar and CartPole environments are MDPs. The observations in MountainCar are in the form of a vector with two entries: the position of the car on the x-axis and velocity. In CartPole, the observations are comprised of 4 entries: cart position, cart velocity, pole angle and pole angular velocity. To turn the environments into POMDPs, they have been modified to add noise to the observations generated from a noise distribution. In this experiment, the noise distribution is the Gaussian distribution with a mean of 0 and some standard deviation (st.d.). The values of the standard deviation used in these experiments are 0.001, 0.005, 0.01, 0.02, 0.05. In MountainCar, the noise is added to the position component of the observation. Perturbations to the dynamics of this environment are made to the force and the gravity constants with default values of 0.001 and 0.0025 respectively. To create perturbations, these values are multiplied with a perturbation constant. The perturbation constants used for these experiments are $[0.005, 0.01, 0.1, 0.2, 0.5, 0.75, 0.9, 0.95, 1.05, 1.1, 1.25, 1.5, 2, 3]$. For example, multiplying the force by 0.95 is equivalent to reducing it by 5% which in this case would be considered a small perturbation. Multiplying the force by 2 is equivalent to increasing it by 100% which would be considered a big perturbation. In CartPole, the noise is added to all components of the observation. Perturbations to the environment dynamics are made to the gravity and length constants with default values of 9.8 and 0.5 respectively. To create the perturbations, these values are also multiplied with a perturbation constant. For gravity, the constants used in the experiment are the same as the ones used in MountainCar. For length, which resembles the length of the pole, the multipliers are $[0.2, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.8]$.

In total, 5 models were trained for both MountainCar and CartPole, each on the environment with different noise values. Each model used a context window of size 5 and a reconstruction window of size 27, meaning that the snippets were of size 32. Ten inference masks were used, each with a number of unmasked positions in the reconstruction window between 2 and 5. The masking rate for the random masks the Dynamics module generates was set to 0.9. Drawing it from a set of rates showed to degrade the performance in preliminary tests. In the dynamics model, 24 layers of gMLP blocks were used. A full list of hyperparameters for the framework used in the experiments with the MountainCar environments can be found in Appendix A.2. For the framework in the experiments with the CartPole environments, a list of the hyperparameters is provided in Appendix A.3.

To collect data from both the MountainCar and CartPole environments, trained DQN agents were used. The training and validation sets in all experiments were comprised of 200 and 50 episodes of the environment respectively. The testing data in MountainCar was comprised of 50 episodes with a change in the environment dynamics injected between steps 20 and 70 in the episode, and 50 episodes with no changes in the dynamics. For CartPole, the testing data was comprised of 200 episodes with a change in the dynamics injected randomly between steps 150 and 250. The dynamics model in each instance of the framework for the MountainCar and CartPole environments was trained for 100 and 30 epochs respectively. The models with the lowest validation error were saved and used.

4.1.2 Results

4.1.2.1 Results for MountainCar environments

The results of the experiments with the MountainCar environment are shown in Fig 4.3. In all plots, the x-axis is the perturbation multiplier and the y-axis is the metric used for evaluation. In the plots, each line corresponds to an environment with a specific amount of noise, dictated by the standard deviation in the Gaussian distribution.

When it comes to OODD detection, the precision of the ID trajectories is an important metric. Low precision indicates that many OOD samples are detected as ID. In safety-critical systems, that would be a problem as not detecting OOD snippets could be a health hazard. Looking at the plots we can observe that the precision of the ID trajectories is generally higher for the environments with less noise: st.d. of 0.001, 0.005, 0.01, and higher for the environments with st.d. of 0.02 and 0.05. We can also see that the precision drops down when the perturbation in the dynamics is low (up to +/- 25%). For bigger changes in the dynamics, the precision tends to reach a value above 0.9, around 0.97 for environments with lower amounts of noise. Interestingly, perturbations to the force seem to affect the precision more than perturbations to the

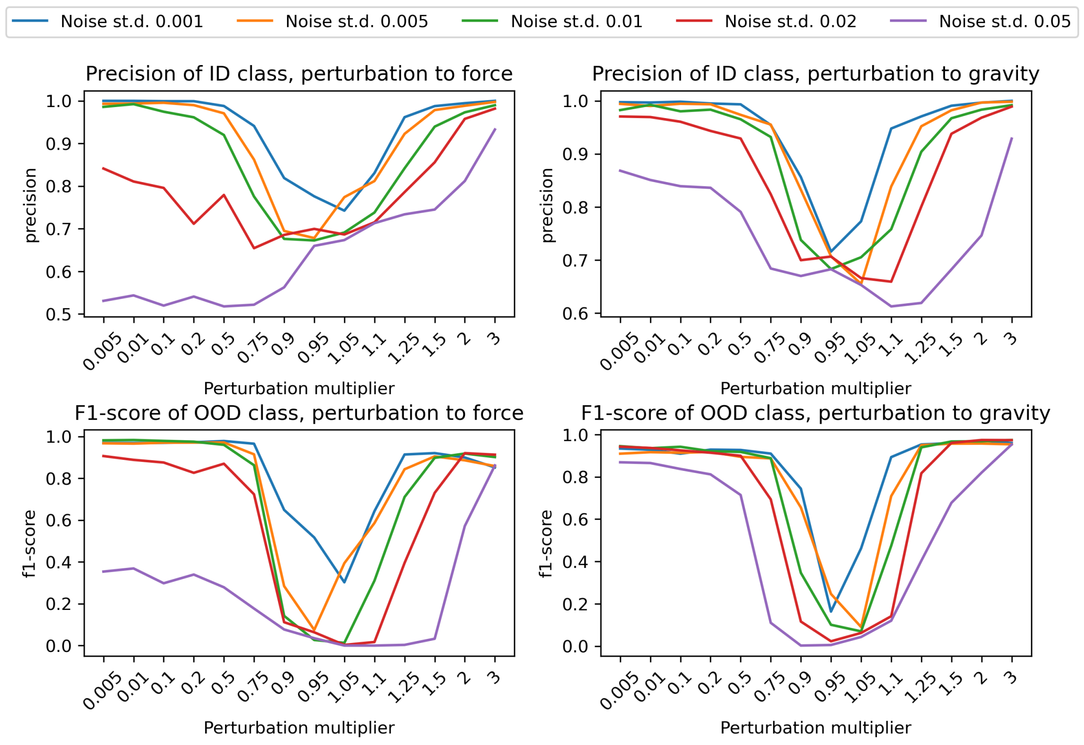

Figure 4.3: Results of the experiments with the MountainCar environments

gravity. Namely, we can observe that the precision for the ID trajectories when the force is small and the noise is somewhat bigger (st.d. 0.02 and 0.05) is very low: around 0.53 for the environment with noise st.d 0.05, whereas, the precision is higher for the same environment when the perturbation is to the gravity. In fact, all environments show a positive trend in the precision of the ID snippets as the perturbation to gravity is increasing.

We evaluate the performance of the model in the task of OODD detection using the F_1-score of the OOD trajectories as this is a standard metric for performance on OOD detection tasks and has been used in previous research [26]. Looking at the subplots, we can see that the F_1-score was high, around 0.97 for environments with low to moderate noise levels (st.d. of 0.001, 0.005, 0.01) and a larger perturbation to the dynamics such that the force is reduced by more than 25%. For big perturbations to the force such that the force is increased by more than 25%, the F_1-score averaged around 0.85 for the same environments. For the environments with noise with st.d. of 0.02 and 0.05, the F_1-score was lower. The maximal F_1-score achieved by the environment with noise st.d. of 0.05 was around 0.4 when the force was small (big perturbation to the dynamics). Similarly to the precision results for the ID snippets, the F_1-score increased for all environments as the perturbation to the force increased in the positive direction (when the force increased). In the gravity plot for OOD trajectories, we can see that the plot of the F_1-score had a similar shape for all environments. The difference is in the values, namely, the environments with higher noise st.d. have lower F_1-scores.

These results show that the developed framework is capable of achieving high performance in environments where the noise levels are low to medium (noise with st.d. of up to 0.01 in the experiments) and the perturbations are medium to high (at least +/-25% the original value). In such scenarios, the framework can achieve an F_1-score for the OOD trajectories and precision of the ID snippets of around 0.97.

4.1.2.2 Results for CartPole environments

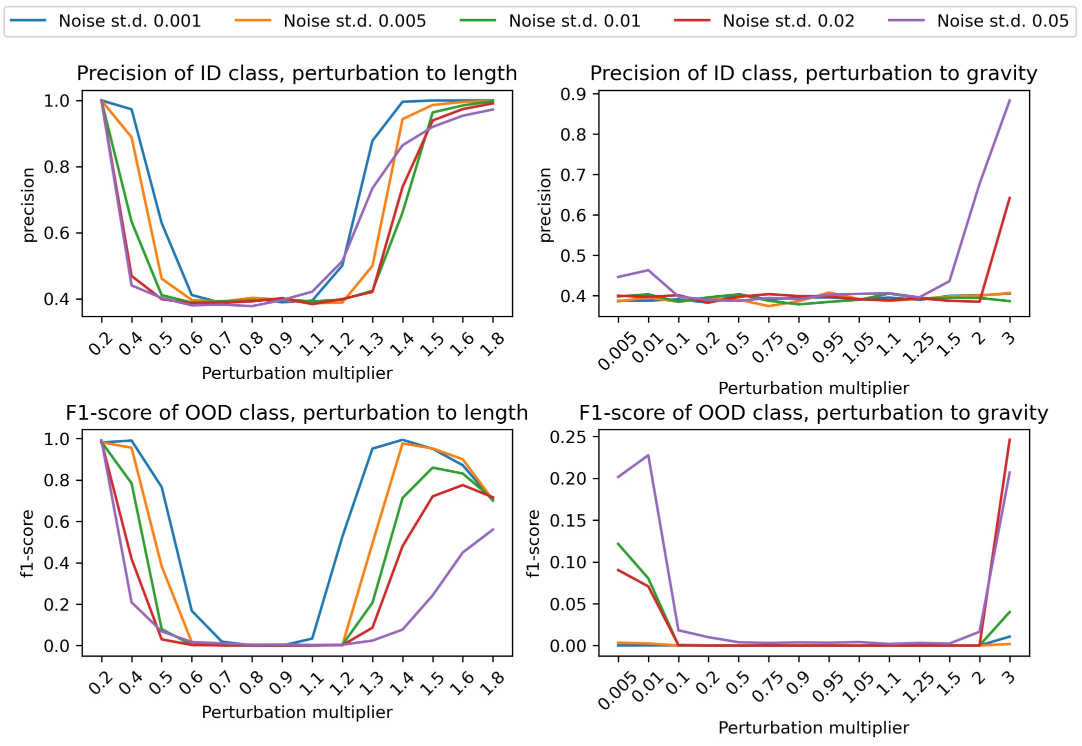

Figure 4.4: Results of the experiments with the CartPole environments

The results of the experiments with the CartPole environment are shown in Fig 4.4. Similarly to the results for the MountainCar environment, in all plots, the x-axis is the perturbation multiplier and the y-axis is the metric used for evaluation. In the plots, each line corresponds to an environment with a specific amount of noise, dictated by the standard deviation in the Gaussian distribution.

Here we can observe that the precision of the ID trajectories and the F_1-score of the OOD trajectories with perturbations to the length of the pole are high when the length of the pole has been increased. In particular, the smaller increases in the length of the pole are detectable in environments with lower amounts of noise. For example, in the environment with noise with st.d. of 0.001, the achieved F_1-score of the OOD trajectories was 0.95 and the achieved precision for the ID trajectories was 0.93. In all environments, the precision of the ID snippets for bigger increases in the length of the pole ($\geq +50\%$) is above 0.95. The highest F_1-score of the OOD trajectories is achieved in environments with lower amounts of noise (st.d. of 0.001 and 0.005) for a 40% increase to the length of the pole. In both cases, the F_1-score is above 0.97. The F_1-score for the other environments is lower. We can also notice that the F_1-score tends to decrease as the length of the pole increases. We can explain this by the fact that when the length of the pole increases considerably, the agent tends to quickly fail and the episode terminates. This leads to a low number of OOD trajectories compared to ID trajectories which reduces the score. Finally, we can also notice that the framework has difficulty detecting a decreasing length of the pole. In fact, both the precision for the ID trajectories and the F_1-score of the OOD trajectories are high (around 0.99) when the length of the pole is 20% the original length. In this case, the agent does not fail so the number of the OOD trajectories remains high which is why the score is also high. Finally, we note that smaller decreases in the length of the pole seem to be undetectable by the framework.

The results with perturbations to the gravity are rather interesting. In all cases, both the precision of the ID trajectories and the F_1-score of the OOD trajectories are low. We believe that this is due to the value of the gravity having little effect on the performance of the agent.

The poor results for the perturbations to the gravity and the results for the small perturbations to the length of the pole (especially when the length is decreased) demonstrate that the framework is mostly capable of detecting changes in the environment when these changes affect the performance of the agent. In fact, increasing the length of the pole has the largest impact on the performance of the agent. When the length increases significantly ($\geq +50\%$), the agent tends to fail to balance the pole which leads to the pole falling and the episode terminating. Perturbations to the gravity and small

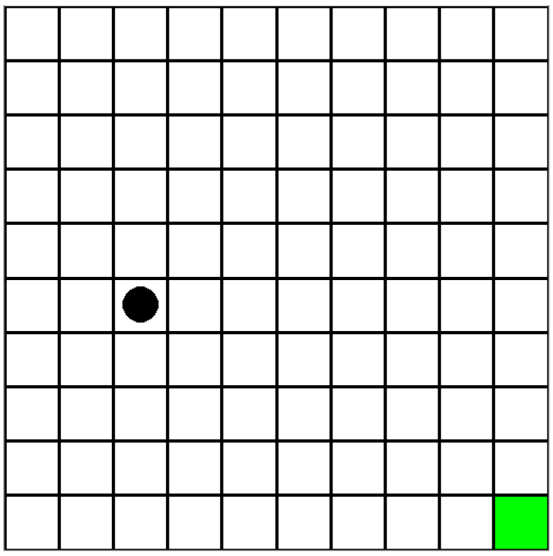

Figure 4.5: A frame in the Grid World environment. The black circle is the agent's location and the green square is the goal state.

decreases in the length of the pole do not affect the performance of the agent enough to trigger a detection.

4.2 Experiment with a non-deterministic environment

4.2.1 Purpose and setting of the experiment

One of the design goals of the developed OODD detection framework was to create a framework, capable of operating with non-deterministic environments. The purpose of this experiment is to investigate if the framework can learn the non-deterministic dynamics of the environment and successfully detect trajectories with OODD.

The environments in this experiment are discrete Grid World environments consisting of a 10×10 grid. The goal of the agent in this environment is to reach a pre-defined goal state which is the bottom right square of the grid. See Fig 4.5. The starting position of the agent is in the top left corner. The agent can take one of 4 actions: move up, move down, move left, move right. If the action that the agent takes leads to a position outside of the grid, the agent is prevented from moving in that direction and remains in its previous position. The environment is non-deterministic so when the agent takes an action, it will go in the desired direction with probability $1-p$ and the opposite direction with probability p, which we call the stochasticity parameter. Further to this, the observations the agent receives are noisy, namely, there is a 0.1 probability that the agent observes one of the 4 neighboring positions. One perturbation to the dynamics of the environment was added such that when this perturbation is activated, the map between the actions and directions of movement changes. This map maps an action with a direction that is left of the direction of the original map. For example, move up is mapped with the direction of move left.

To conduct the experiment, 4 environments with different values for p were used. The used values of p were 0.1, 0.2, 0.3 and 0.4. Perturbations to the environment dynamics are made to the map between the actions and the directions of movement. Namely, the directions change by 90 degrees so that move up moves the agent to the left, move left moves the agent down, etc. The trained frameworks have 48 layers of gMLP blocks, a context window of size 5 and a reconstruction window of size 8. Each used 10 inference masks with 3 to 6 unmasked positions. The masking rate for the random masks of the Dynamics module was 0.8. The top 1% of errors in the lowest density error regions define the OOD intervals in the Detectors module. In this experiment, apart from the observed states, the actions of the agent were also used. A full list of hyperparameters for the framework used in the experiments with the Grid World environment can be found in Appendix A.4.

To collect data from the environments, a trained Q-Learning agent was used. For each environment, 200 episodes of data were used for training and 50 for validation. The test data consist of 50 episodes with ID dynamics and 50 episodes with OODD. The episodes with OODD are generated by perturbing the dynamics between steps 5 and 9 of the episode.

4.2.2 Results

The results of the experiment are shown in Fig 4.6. There you can see the recall of the ID trajectories and the F_1-score of the OOD trajectories for the 4 environments with different values of the stochasticity parameter.

An important metric for non-deterministic environments is the recall of the ID trajectories. The recall is defined as $\frac{TP}{TP+FN}$, where TP is true positives and FN is false negatives. A high recall means that the count of false negative ID trajectories is low.

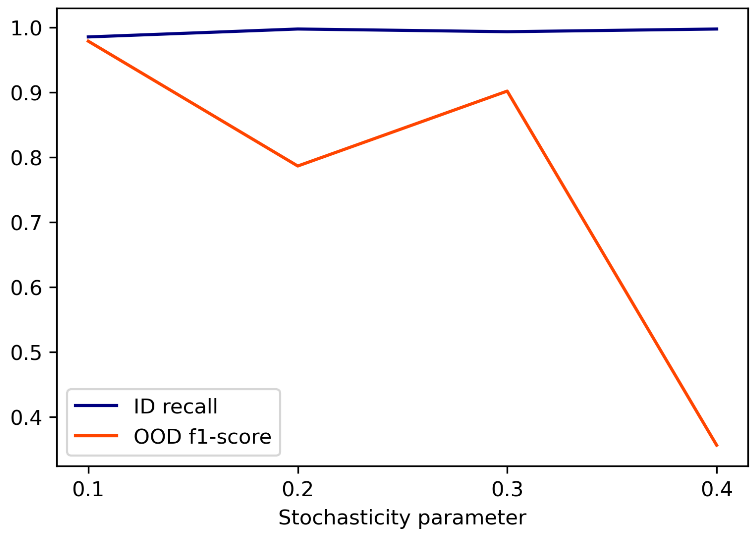

Figure 4.6: Results for the experiments with Grid World environments

Hence, a low number of ID trajectories have been predicted to be OOD. The high recall of 0.99 of the ID trajectories shows that the framework is capable of identifying correctly ID trajectories of a non-deterministic environment.

We also evaluate the framework on the OODD detection task using the F_1-score for the OOD trajectories. We can observe that the score is highest, about 0.98, for the environment with the smallest stochasticity parameter which denotes the probability that the agent does not move in the direction of the action but the opposite direction due to the non-determinism of the environment. The F_1-score for the environments with a value of the stochasticity parameter of 0.2 and 0.3 remains high: 0.79 and 0.9 respectively. The drop in the performance for the former one is rather unexpected and is likely due to a problem with the training of the model or with the data and would require further investigation. The F_1-score for the environment with a value of the stochasticity parameter of 0.4 is rather low: 0.36. This result would normally be concerning, however, we can offer an explanation. Since the stochasticity parameter controls the probability that the opposite direction is taken, a value near 0.5 would mean that there is about equal probability that the agent moves in either direction and so we can expect that the agent would often oscillate around one location. The added noise makes it appear in many positions around that location. When the dynamics is changed, the agent would continue oscillating around the same location (even if the directions are changed). This makes it difficult to distinguish between the two, which can explain the low F_1-score.

The high F_1-score for the OOD trajectories in the environments with lower values of the stochasticity parameter, combined with the high recall of 0.99 for the ID trajectories demonstrates that the developed framework is capable of operating in non-deterministic environments and that it is a step towards building OODD detection methods for such environments.

Chapter 5

Conclusions

In this project, we investigated the problem of Out-of-Distribution Dynamics Detection in Deep Reinforcement Learning. We began by introducing the problem and explaining why it matters. We then presented some of the background material related to it, including previous work in the area and more general approaches to OOD detection. We also discussed models that can model the environment dynamics. After this, we looked at some limitations in the previous research and identified how the proposed framework differs from the previous work before presenting the developed framework. We discussed each of the components of the framework in detail, explaining their purpose, how they relate to the other components and how they works. Afterward, we tested the proposed framework in two different experiments. The first experiment aimed to test the performance in POMDP environments and the amount of perturbation that was detectable by the framework. The second experiment tested whether the framework can operate in a non-deterministic environment.

The results from the first experiment demonstrated that the developed framework is capable of detecting changes in the environment if these changes affect the performance of the agent as evidenced by the results in the CartPole environment. There, changes to the gravity constant and small changes to the length of the pole did not affect the agent's ability to balance the pole. Neither did they change the way the agent balanced it. The results further demonstrated that the framework is suitable for detecting OOD trajectories in POMDP environments where the changes to the environment dynamics is reflected by the performance of the agent. In particular, in the MountainCar environments, the model achieved a high F_1-score of 0.97 in environments with low to moderate observation noise and perturbations to the dynamics of the environment, where the perturbation was in the form of a change to a parameter of the environment of at least 25%. For environments with high levels of noise, or when the perturbations to the dynamics were small, the framework performance was lower, namely, the F_1-score was below 0.4. In the CartPole environments, the agent managed to achieve high precision for the ID trajectories and high F_1-score for the OOD trajectories when changes of at least $+50\%$ or -60% were made to the length of the pole. In particular, the precision of the ID trajectories was above 0.95, while F_1-score for OOD trajectories reached above 0.97 for environments with low observation noise (st.d. 0.001 and 0.005). We noted that the F_1-score for OOD snippets was decreasing in the results when the pole length was increasing due to the number of OOD trajectories decreasing. This is caused by the termination of episodes when the agent fails to balance the pole. The higher number of ID trajectories means that the precision of the OOD trajectories was lower due to the higher number of false positives (FP) compared to the number of true positives (TP). Furthermore, we believe that small changes to the dynamics are hidden by the observation noise and so they will always be difficult to detect in POMDP environments. Environments with high levels of observation noise are problematic in general as agents also struggle to solve these environments. Hence, OODD detection is also difficult in such environments as evidenced by the results. Hence, the results in this experiment demonstrate that the developed framework is suitable for detecting changes in the environments with low to moderate levels of observation noise, as long as the changes in the environment affect the performance of the agent. We argue that changes that affect the performance of the agent are more crucial as such changes could for example be a health hazard in the case of autonomous vehicles and lead to a loss of revenue in the case of trading bots.

The second experiment demonstrated that the framework is capable of working with non-deterministic environments. In particular, the achieved high recall of 0.99 for ID trajectories shows that the framework can correctly identify the non-deterministic dynamics from the training examples and label trajectories such trajectories as ID. The achieved OODD detection performance in this experiment was also satisfactory, achieving F_1-score of 0.9+ in two cases and 0.79 in one. The F_1-score was also low for the environment where the probability that the chosen action does not move the agent in the desired direction was 0.4. We explained that this result is most likely due to the inherent randomness of the movements when this value is close to 0.5 as is the case of 0.4. Nevertheless, the high F_1-scores for OOD trajectories in the other cases, combined with the recall of 0.99 for ID trajectories demonstrates that the developed framework is a step towards developing methods for OODD detection that can work with non-deterministic environments.

Apart from the advantages of the developed framework, namely, the ability to work with POMDP environments and non-deterministic environments, we identify two disadvantages of the current framework. First, the current framework cannot detect changes to the dynamics in a non-deterministic environment if that change is to the stochasticity of the dynamics. For example, the framework cannot detect changes in the case where the probability of observing a particular state as the next state increases (given that it was not 0 to begin with). This is because OOD trajectories (trajectories under the OOD dynamics) would also be possible under the ID dynamics and the framework would not be able to detect the OOD trajectories as such. In order to detect these, the framework would have to detect changes to the probability distribution of the next states. Hence, as future work, the framework could also be extended to detect such changes.

Second, we note that the trajectory should contain enough of the perturbed dynamics for it to be detectable. What this means is that there is normally a lag between the moment the dynamics changes and the moment the framework can detect it. While using smaller snippets can reduce this lag, we believe that using the framework purely in a reconstruction-based manner would not offer the best solution for eliminating the lag. Therefore, as future work, the framework can be extended to also operate partially as a prediction-based model. This could be achieved with the introduction of a second type of mask, namely, prediction masks which would be used for predicting a number of steps forward. Then, the framework can use the prediction masks as a method of detecting changes early and later confirming them using the current methodology.

As further future work, we would also like to improve the stability of the framework as at times, the dynamics model in the Dynamics module may not converge optimally, leading to decreased OODD detection performance. One possible approach would be to build an ensemble of frameworks that work together and combine the decisions from their detectors to provide a unanimous decision. Furthermore, we are interested in conducting a study on the effects different parameters have on the performance and flexibility of the framework. For example, we would investigate whether decreasing the reconstruction window can reduce the lag without sacrificing the performance of the framework.

Bibliography

[1] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Ng. An application of reinforcement learning to aerobatic helicopter flight. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems*, volume 19. MIT Press, 2006.

[2] Gaon An, Seungyong Moon, Jang-Hyun Kim, and Hyun Oh Song. Uncertainty-based offline reinforcement learning with diversified q-ensemble. *CoRR*, abs/2110.01548, 2021.

[3] Anonymous. CODit: Conformal out-of-distribution detection in time-series data. *Submitted to Transactions on Machine Learning Research*, 2022. Rejected.

[4] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[5] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. *Reinforcement Learning and Chess*, page 91–116. Nova Science Publishers, Inc., USA, 2001.

[6] Taha Belkhouja, Yan Yan, and Janardhan Rao Doppa. Out-of-distribution detection in time-series domain: A novel seasonal ratio scoring approach, 2023.

[7] RICHARD BELLMAN. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[8] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks, 2015.

[9] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki,

Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[11] Micah Carroll, Orr Paradise, Jessy Lin, Raluca Georgescu, Mingfei Sun, David Bignell, Stephanie Milani, Katja Hofmann, Matthew Hausknecht, Anca Dragan, and Sam Devlin. Unimask: Unified inference in sequential decision problems, 2022.

[12] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

[13] Xingyu Chen, Xuguang Lan, Fuchun Sun, and Nanning Zheng. A boundary based out-of-distribution classifier for generalized zero-shot learning. *CoRR*, abs/2008.04872, 2020.

[14] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *CoRR*, abs/1805.12114, 2018.

[15] Peng Cui and Jinjia Wang. Out-of-distribution (ood) detection based on deep learning: A review. *Electronics*, 11(21), 2022.

[16] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning, 2018.

[17] Mohamad H. Danesh and Alan Fern. Out-of-distribution dynamics detection: Rl-relevant benchmarks and results. *CoRR*, abs/2107.04982, 2021.

[18] Zahra Zamanzadeh Darban, Geoffrey I. Webb, Shirui Pan, Charu C. Aggarwal, and Mahsa Salehi. Deep learning for time series anomaly detection: A survey, 2022.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[20] Terrance DeVries and Graham W. Taylor. Learning confidence for out-of-distribution detection in neural networks, 2018.

[21] Nan Ding, HaoXuan Ma, Huanbo Gao, YanHua Ma, and GuoZhen Tan. Real-time anomaly detection based on long short-term memory and gaussian mixture model. *Computers & Electrical Engineering*, 79:106458, 2019.

[22] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 10 2022.

[23] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2016.

[24] Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout, 2017.

[25] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.

[26] Tom Haider, Karsten Roscher, Felippe Schmoeller da Roza, and Stephan Günnemann. Out-of-distribution detection for reinforcement learning agents with probabilistic dynamics models. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '23, page 851–859, Richland, SC, 2023. International Foundation for Autonomous Agents and Multiagent Systems.

[27] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[28] Yen-Chang Hsu, Yilin Shen, Hongxia Jin, and Zsolt Kira. Generalized ODIN: detecting out-of-distribution image without learning from out-of-distribution data. *CoRR*, abs/2002.11297, 2020.

[29] Rui Huang, Andrew Geng, and Yixuan Li. On the importance of gradients for detecting distributional shifts in the wild. *CoRR*, abs/2110.00218, 2021.

[30] Rui Huang and Yixuan Li. MOS: towards scaling out-of-distribution detection for large semantic space. *CoRR*, abs/2105.01879, 2021.

[31] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem, 2021.

[32] Dihong Jiang, Sun Sun, and Yaoliang Yu. Revisiting flow generative models for out-of-distribution detection. In *International Conference on Learning Representations*, 2022.

[33] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998.

[34] Bangalore Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Kumar Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *CoRR*, abs/2002.00444, 2020.

[35] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017.

[36] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles, 2017.

[37] Kimin Lee, Younggyo Seo, Seunghyun Lee, Honglak Lee, and Jinwoo Shin. Context-aware dynamics model for generalization in model-based reinforcement learning. *CoRR*, abs/2005.06800, 2020.

[38] Jingyao Li, Pengguang Chen, Shaozuo Yu, Zexin He, Shu Liu, and Jiaya Jia. Rethinking out-of-distribution (ood) detection: Masked image modeling is all you need, 2023.

[39] Jinning Li, Chen Tang, Masayoshi Tomizuka, and Wei Zhan. Dealing with the unknown: Pessimistic offline reinforcement learning. *CoRR*, abs/2111.05440, 2021.

[40] Fangchen Liu, Hao Liu, Aditya Grover, and Pieter Abbeel. Masked autoencoding for scalable and generalizable decision making, 2023.

[41] Hanxiao Liu, Zihang Dai, David R. So, and Quoc V. Le. Pay attention to mlps. *CoRR*, abs/2105.08050, 2021.

[42] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2):311–365, 1992.

[43] Trevor McInroe, Lukas Schäfer, and Stefano V. Albrecht. Learning representations for control with hierarchical forward models, 2022.

[44] Jeff Michels, Ashutosh Saxena, and Andrew Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML '05, page 593–600, New York, NY, USA, 2005. Association for Computing Machinery.

[45] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[46] Aaqib Parvez Mohammed and Matias Valdenegro-Toro. Benchmark for out-of-distribution detection in deep reinforcement learning. *CoRR*, abs/2112.02694, 2021.

[47] Mohsin Munir, Shoaib Ahmed Siddiqui, Andreas Dengel, and Sheraz Ahmed. Deepant: A deep learning approach for unsupervised anomaly detection in time series. *IEEE Access*, 7:1991–2005, 2019.

[48] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *CoRR*, abs/1708.02596, 2017.

[49] Thanh Nguyen, Tung Minh Luu, Thang Vu, and Chang D. Yoo. Sample-efficient reinforcement learning representation learning with curiosity contrastive forward dynamics model. *CoRR*, abs/2103.08255, 2021.

[50] Ian Osband, John Aslanides, and Albin Cassirer. Randomized prior functions for deep reinforcement learning, 2018.

[51] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. *CoRR*, abs/1602.04621, 2016.

[52] Pramuditha Perera, Poojan Oza, and Vishal M. Patel. One-class classification: A survey, 2021.

[53] Stanislav Pidhorskyi, Ranya Almohsen, Donald A. Adjeroh, and Gianfranco Doretto. Generative probabilistic novelty detection with adversarial autoencoders. *CoRR*, abs/1807.02588, 2018.

[54] Jie Ren, Stanislav Fort, Jeremiah Z. Liu, Abhijit Guha Roy, Shreyas Padhy, and Balaji Lakshminarayanan. A simple fix to mahalanobis distance for improving near-ood detection. *CoRR*, abs/2106.09022, 2021.

[55] Mohammad Sabokrou, Mohammad Khalooei, Mahmood Fathy, and Ehsan Adeli. Adversarially learned one-class classifier for novelty detection. *CoRR*, abs/1802.09088, 2018.

[56] Andreas Sedlmeier, Thomas Gabor, Thomy Phan, Lenz Belzner, and Claudia Linnhoff-Popien. Uncertainty-based out-of-distribution classification in deep reinforcement learning. *CoRR*, abs/2001.00496, 2020.

[57] Andreas Sedlmeier, Robert Müller, Steffen Illium, and Claudia Linnhoff-Popien. Policy entropy for out-of-distribution classification. *CoRR*, abs/2005.12069, 2020.

[58] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George 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*, 529:484–489, 01 2016.

[59] W.D. Smart and L. Pack Kaelbling. Effective reinforcement learning for mobile robots. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 4, pages 3404–3410 vol.4, 2002.

[60] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[61] Gerald Tesauro. Temporal difference learning and td-gammon. *J. Int. Comput. Games Assoc.*, 18(2):88, 1995.

[62] Joost van Amersfoort, Lewis Smith, Yee Whye Teh, and Yarin Gal. Simple and scalable epistemic uncertainty estimation using a single deep deterministic neural network. *CoRR*, abs/2003.02037, 2020.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[64] Yue Wu, Shuangfei Zhai, Nitish Srivastava, Joshua M. Susskind, Jian Zhang, Ruslan Salakhutdinov, and Hanlin Goh. Uncertainty weighted actor-critic for offline reinforcement learning. *CoRR*, abs/2105.08140, 2021.

[65] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. Generalized out-of-distribution detection: A survey. *CoRR*, abs/2110.11334, 2021.

[66] Yijun Yang, Ruiyuan Gao, and Qiang Xu. Out-of-distribution detection with semantic mismatch under masking, 2022.

[67] Yibo Zhou. Rethinking reconstruction autoencoder-based out-of-distribution detection, 2023.

Appendix A

Hyperparameters for the frameworks used in the conducted experiments

A.1 Hyperparameters description

A description of the hyperparameters of the OODD detection framework are provided in Table A.1

Hyperparameter	**Description**
layers	Number of layers of gMLP blocks for the dynamics model in the Dynamics module
states_input_size	The number of dimensions of the states vector if continuous or the number of states in the environment if discrete
states_output_size	The number of dimensions of the observations produced by the model
embed_size	The size of the embeddings produced by the encoder.
context_window_size	Length of the context window in the input snippet
window_size	The length of the snippet
actions_input_size	The number of dimensions of the actions vector if continuous or the number of actions if discreet. It is set to None if no actions are used
discrete_actions	Whether the actions are discrete (if used)
discrete_states	Whether the states are discrete
masking_rate	A list of masking rates used by the Dynamics module for creating random masks

train_loss_fn	The loss function used for training the Dynamics module
inf_mask_err_fn	The error function used to calculate the error of the inference masks by the Inference masks module
num_inf_masks	Number of inference masks that are selected in the end by the Inference masks module to be used by the Detectors module
min_unmasked	Minimum number of unmasked positions in the reconstruction window of the generated inference masks
max_unmasked	Maximum number of unmasked positions in the reconstruction window of the generated inference masks
num_per_unmasked	Number of masks generated for each number of unmasked positions in the reconstruction window
pwr	The value of β, the regularization parameter in the Inference masks module
detection_err_rate	The rate of validation errors in low-density regions of the distribution of the validation errors per inference mask
max_horizon	The maximum size of the window used to calculate the density of the region in which a given error of the validation set is located
min_error_dist	The minimum length of an OOD interval
detector_loss_fn	The loss function used by the Detectors module to calculate the reconstruction error of a trajectory

Table A.1: Description of hyperparameters of the OODD detection framework

A.2 Hyperparameters of the framework used in the experiments with the MountainCar environments

The hyperparameters for the OODD detection framework used in the experiments with the MountainCar environments can be found in Table A.2. Description of the hyperparameters is available in Table A.1.

Hyperparameter	**Value**
layers	24

states_input_size	2
states_output_size	2
embed_size	32
context_window_size	5
window_size	32
actions_input_size	None
discrete_actions	False
discrete_states	False
masking_rate	[0.9]
train_loss_fn	MSE
inf_mask_err_fn	MSE
num_inf_masks	10
min_unmasked	2
max_unmasked	5
num_per_unmasked	10
pwr	0.27
detection_err_rate	0.05
max_horizon	50
min_error_dist	0.05
detector_loss_fn	MSE

Table A.2: Hyperparameters for the framework used in the MountainCar environments

A.3 Hyperparameters of the framework used in the experiments with the CartPole environments

The hyperparameters for the OODD detection framework used in the experiments with the CartPole environments can be found in Table A.3. Description of the hyperparameters is available in Table A.1.

Hyperparameter	**Value**
layers	24
states_input_size	4
states_output_size	4

embed_size	32
context_window_size	5
window_size	32
actions_input_size	None
discrete_actions	False
discrete_states	False
masking_rate	[0.9]
train_loss_fn	MSE
inf_mask_err_fn	MSE
num_inf_masks	10
min_unmasked	2
max_unmasked	5
num_per_unmasked	10
pwr	0.27
detection_err_rate	0.05
max_horizon	50
min_error_dist	0.05
detector_loss_fn	MSE

Table A.3: Hyperparameters for the framework used in the CartPole environments

A.4 Hyperparameters of the framework used in the experiments with the Grid World environments

The hyperparameters for the OODD detection framework used in the experiments with the Grid World environments can be found in Table A.4. Description of the hyperparameters is available in Table A.1.

Hyperparameter	**Value**
layers	48
states_input_size	100
states_output_size	100
embed_size	32

context_window_size	5
window_size	13
actions_input_size	4
discrete_actions	True
discrete_states	True
masking_rate	[0.8]
train_loss_fn	Cross-entropy loss
inf_mask_err_fn	Cross-entropy loss
num_inf_masks	10
min_unmasked	3
max_unmasked	6
num_per_unmasked	6
pwr	0.27
detection_err_rate	0.01
max_horizon	50
min_error_dist	0.05
detector_loss_fn	Cross-entropy loss

Table A.4: Hyperparameters for the framework used in the Grid World environments