Data Augmentation for Regularizing Learned World Models in Reinforcement Learning

Ron van Bree

January 19, 2021

Supervisors: Christoph Brune, Mannes Poel
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Acknowledgement

First, I would like to thank Christoph Brune for his supervision during this thesis, but also the preparatory Research Topics course. During our meetings he always took the time to discuss things properly, for which I am very grateful. I would also like to thank Mannes Poel for being a supervisor of this thesis and for the many discussions we've had. Mannes looks at things with a critical eye, which I have (almost) always appreciated. The feedback provided by Christoph and Mannes have been essential to the creation of this work.
Secondly, I am very thankful for the support and encouragement received by my family throughout my studies, as well as my friends, on who I could always count for necessary distractions when working on my thesis. Especially in these times, this has made working on my thesis a lot more enjoyable.

Abstract

This work has investigated the effect of data augmentation on learning environment models for data-efficient reinforcement learning. Data augmentation has received relatively little attention in reinforcement learning, compared to its common occurrence in supervised learning. However, the invariance assumptions that allow the augmentation of images have long been recognized to be usable for the simplification of reinforcement learning problems [1]. Recent successes in applying data augmentation to model-free reinforcement learning algorithms [2, 3] raised the question whether data augmentation could be effective enough to enforce these assumptions for learned environment models, for an increase in data efficiency. This has been numerically investigated using an algorithm that is currently among the state of the art for learning environment models, called the Recurrent State Space Model (RSSM). The use of data augmentation was found to have no significant effects on the consistency of the RSSM when faced with state-action sequences that have an equivalent reward expectation.

Contents

1 Introduction **6**

2 Background **8**
2.1 Markov Decision Process . . . 8
2.2 Policy optimization in MDPs . . . 10
2.3 MDP Homomorphisms . . . 11
2.4 Partial Observability . . . 12
2.5 Bayes Filter . . . 13
2.6 Recurrent State Space Model . . . 14
2.6.1 Training objective . . . 15
2.7 Bias-Variance Trade-off . . . 17
2.8 Cross-Entropy Method . . . 18
2.8.1 Relation to policy optimization in MDPs . . . 20
2.9 Data Augmentation . . . 20
2.9.1 Regularization . . . 21

3 Related Work **23**
3.1 Planning in learned environment models . . . 23
3.2 MDP homomorphisms . . . 24
3.3 Data augmentation . . . 24
3.3.1 Data augmentation in reinforcement learning . . . 24

4 Methodology **26**
4.1 PlaNet . . . 26
4.2 Data Augmentation . . . 26
4.2.1 Mirror Augmentation . . . 27
4.2.2 Translate Augmentation . . . 27
4.2.3 Regularization . . . 28
4.3 Evaluation . . . 28
4.4 Experiment Setup . . . 29
4.4.1 Experiment 1 - Performance Upper Bound . . . 29
4.4.2 Experiment 2 - Environment Model and Data Augmentation 29
4.4.3 Experiment 3 - Test for Invariance . . . 29
4.4.4 Experiment 4 - Value Function . . . 30

5 Results and Discussion **31**
5.1 Experiment 1 - Performance Upper Bound . . . 31
5.2 Experiment 2 - Environment Model and Data Augmentation . . 32
5.2.1 Model Predictions . . . 32
5.2.2 Evaluation Results . . . 33
5.2.3 Statistics and Interpretation . . . 35
5.3 Experiment 3 - Test for Invariance . . . 37
5.4 Experiment 4 - Value Function . . . 39

6 Conclusion **40**

7 Appendix - Notation Overview **48**

8 Appendix - Implementation **49**

9 Appendix - Additional Hyperparameters & Implementation Details **50**

9.1 CEM . . . 50
9.2 Recurrent State Space Model . . . 50
9.2.1 State model . . . 50
9.2.2 Reward model . . . 50
9.2.3 Observation model . . . 50
9.2.4 Encoder . . . 51
9.2.5 Training procedure . . . 51
9.3 RSSM (Downscaled) . . . 51
9.4 Action-Value Model . . . 51

10 Appendix - Pseudocode **52**

10.1 Cross-Entropy Method . . . 52
10.2 PlaNet . . . 53

1 Introduction

Despite achieving impressive results [4, 5], deep reinforcement learning (RL) algorithms are still considered to have poor data efficiency and generalization abilities [6, 7]. That is, a difference is observed in train performance and test performance. Minor perturbations in the input data are able to drastically influence model predictions [8], which is undesirable if such algorithms are deployed in a real-world setting. To close the so called generalization gap between train and test performance it is desirable to make the model invariant to these perturbations. This is commonly illustrated as a bias-variance trade-off in the model prediction error. Features that are irrelevant to the objective should be ignored during prediction. However, this introduces a bias in the model and features that are erroneously recognized as irrelevant increase the expected prediction error instead. To create a proper model, the question is which features are most informative for making the right predictions.
In deep reinforcement learning, the algorithm (or agent) is given the opportunity to perform actions upon which it receives feedback from its environment, i.e. it is allowed to query new data. This opportunity also brings a new challenge. Namely, the data distributions shifts based on the agent's beliefs and actions. The currently observed data might not be sufficient to solve the problem and active exploration is required. To increase the challenge even further, RL agents are typically quite limited in their data collection, since this process is time consuming and possibly expensive. Without introducing bias into the model, there are simply too many possibilities to explore from the limited data that is available. Luckily, useful biases are often known beforehand and can be incorporated into the model. With these biases, agents might be able to more robustly deal with "unknown unknowns". That is, variability in the data which was not foreseen.
There are several ways of enforcing known biases into a model, and one of these is data augmentation. Knowing the invariance of certain properties towards the objective, these properties can be modified in existing data to create artificial data points. These artificial data points are used to train the model, which is then better able to recognize this invariance by making more effective use of the data.
Model based reinforcement learning algorithms improve on data efficiency by using a model of their surroundings. Decisions can then be based on simulated data obtained from the environment model. A limitation, however, is that such an environment model is not always available and difficult to build, since everything the agent can encounter should be known and incorporated beforehand. A more robust agent would be able to deal with challenges that were not anticipated. A current challenge in reinforcement learning is to learn/build an environment model from experience obtained from the real world [9, 10, 11, 12]. This is a difficult problem, since the object dynamics are often non-linear and should be modelled from partial information of the environment. Current methods have been identified to overfit on the limited data that is available, and fail to generalize to unseen data [13, 14, 15, 16]. This has led to a focus on finding

ways to embed useful prior information about the world into models, so agents are better able to recognize the structure of their surroundings. With the right abstractions, many RL objectives can be simplified. Consider, for example, a robot arm tasked with moving objects between arbitrary locations. In most circumstances, it is reasonable to assume that the color of the object should not affect this procedure. When such redundancies are known, data augmentation can be used to generate artificial simulations based on existing data. This artificial data enforces a consistency in the model predictions when confronted with such redundancies and is therefore considered a regularization procedure.
Given an environment model, planning was found to be an effective method for action selection, in terms of data efficiency and asymptotic performance [17, 7, 18]. Currently, an algorithm called PlaNet [19] is among the state of the art approaches for planning using a learned environment model. In PlaNet, a recurrent neural network architecture is used to construct an approximate model of the environment. The PlaNet algorithm then uses the Cross-Entropy Method (CEM) [20] to plan a trajectory in latent space, based on the predicted rewards from a learned environment model. This model, named the Recurrent State Space Model (RSSM), has been the basis of multiple state-of-the-art Reinforcement Learning algorithms [19, 21, 22] and uses an encoder to update its internal state. Recent work, however, suggests that these encoders overfit on the limited data that is available [3]. Data augmentation was applied to mitigate this and was found to be surprisingly effective. Pre-existing model-free algorithms, such as Soft Actor-Critic (SAC) and PPO were able to exceed model based algorithms in terms of data efficiency and performance when combined with data augmentation. This raises the question if data augmentation, when applied to model-based RL algorithms, can be used to achieve the same increase of performance. This forms the main focus of this research, namely
RQ 1: How does regularizing the Recurrent State Space Model used in PlaNet through data augmentation affect its performance in terms of data efficiency and asymptotic performance?
The following preliminary questions help to answer this, and are answered in the Background and Methodology sections.

1.1 What is PlaNet?
1.2 What type of data augmentation should be used?
1.3 How can data augmentation be incorporated in the algorithm?
1.4 How can the model performance be evaluated?

The authors of PlaNet hypothesize that the algorithm can be improved by using a function that can estimate the utility of environment states/transitions (called a value function), since it can be used during planning to take into account states beyond the planning horizon. Given the success of Kostrikov et al. [3] in regularizing a value function through data augmentation, the secondary research focus is
RQ 2: How does the regularization procedure implemented by Kostrikov et al. affect the PlaNet algorithm when it is augmented with a value function?

2 Background

This section gives an extensive overview of the concepts relevant to this work. This makes the document relatively self-contained, but its purpose is also to provide a good exercise for the author to gain a thorough understanding of the topics. Readers that are already familiar with these concepts can safely skip it. In control theory and reinforcement learning different notations are used for the same concepts. In this document the reinforcement learning conventions are followed.

2.1 Markov Decision Process

Decision problems where actions influence not just immediate rewards, but also future rewards, are often framed as Markov Decision Processes (MDPs). An MDP can be thought of of as an environment which is in some state. An agent can select actions to perform in this environment. Executing these actions allows the agent to change the environment state, as well as obtain a reward for every action taken. The goal of the agent is to build a strategy that maximizes the total reward obtained when executing actions in the environment. More formally, an MDP $\mathcal{M}$ is defined as a $4-$tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \Psi, \mathcal{P} \rangle$, where

- $\mathcal{S}$ - a set of possible states
- $\mathcal{A}$ - a set of possible actions
- $\Psi \subseteq \mathcal{S} \times \mathcal{A}$ - a set containing permissible state-action pairs
- $\mathcal{P}(s', r|s, a)$ - a transition function $\mathcal{P} : \mathcal{S} \times \mathbb{R} \times \Psi \rightarrow [0, 1]$ giving the probability of transitioning to state $s' \in \mathcal{S}$ and obtaining reward $r \in \mathbb{R}$, given that the current state is s, action a is executed and $(s, a) \in \Psi$.

The agent thus interacts with the environment in discrete time steps. At each step t the agent receives an environment state s_t based on which it selects an action a_t. The agent executes a_t in the environment upon which it receives environment state s_{t+1} and a reward r_{t+1}. The transitions satisfy the Markov property, meaning that the transition probabilities from state s_t are not dependent on previously encountered states. By sequentially selecting actions the agent traverses a trajectory in the environment, which is a sequence

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, ...$$

MDPs are stochastic processes, and let S_t, A_t, R_t denote the random variables corresponding to the state, action and reward at time t, respectively. Then

$$\mathcal{P}(s', r|s, a) \doteq p(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

Also, let the following function denote the expected reward for some state transition

$$
\begin{aligned}
\mathcal{R}(s,a) &\doteq \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] \\
&= \int\limits_{r\in\mathbb{R}} \int\limits_{s'\in\mathcal{S}} r\mathcal{P}(s', r|s, a) ds' dr
\end{aligned}
$$

and let

$$
\mathcal{P}(s'|s,a) \doteq \int\limits_{r\in\mathbb{R}} \mathcal{P}(s', r|s, a) dr
$$

denote the transition probabilities.

In the MDP framework, there are two sources of uncertainty when computing a trajectory. When performing an action a in state s, the subsequent state s' as well as the reward obtained are sampled from the transition function (i.e. $(s', r) \sim \mathcal{P}(\cdot, \cdot|s, a)$). The agent has control over which action is selected at each time step. However, it is often still in the agent's best interest to do this in a stochastic manner, making it the second source of uncertainty. To know which action should be selected, the agent builds a policy $\pi : \Psi \to [0, 1]$. For any state $s \in \mathcal{S}$, $\pi(\cdot|s)$ defines a probability distribution over all permissible actions, which the agent can sample to perform action selection. Let S_t, A_t, R_t denote the random variables corresponding to the trajectory's states, actions and rewards of step t, respectively.

The sum of rewards obtained from following a trajectory until time step T starting from state s_t is referred to as the 'return' G_t, where[1]

$$
G_t \doteq \sum_{\tau=t+1}^{T} R_\tau
$$

The goal of the agent is to find a policy π such that the expected return is maximized. The expected return under some policy is an important concept in reinforcement learning and is referred to as the value function $v_\pi : \mathcal{S} \to \mathbb{R}$,

[1] When considering infinite trajectories a discount factor γ is used, such that the total reward of infinite trajectories remains finite. The return is then defined as $G_t \doteq \sum_{\tau=0}^{\infty} \gamma^\tau R_{t+\tau+1}$. In this work, all environments are bounded with a maximal number of iterations. Thus only the finite-horizon case is considered and γ is omitted.

where[2]

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t | S_t = s\right] && (1)\\ &= \mathbb{E}_\pi\left[R_{t+1} + G_{t+1} | S_t = s\right] && (2)\\ &= \int\limits_a \pi(a|s) \int\limits_{s'} \mathcal{P}(s', r|s, a)\left[r + \mathbb{E}_\pi\left[G_{t+1}|S_{t+1} = s'\right]\right] ds' da && (3)\\ &= \int\limits_a \pi(a|s) \int\limits_{s'} \mathcal{P}(s', r|s, a)\left[r + v_\pi(s')\right] ds' da && (4) \end{aligned}$$

$v_\pi(s_t)$ is here defined recursively in terms of $v_\pi(s_{t+1})$. This is referred to as the Bellman equation for v_π. The value function conditioned on some action is referred to as the action-value function $q_\pi(s, a)$

$$q_\pi(s, a) \doteq \int\limits_{s'} \mathcal{P}(s', r|s, a)\left[r + v_\pi(s')\right] ds' \quad (5)$$

s.t.

$$v_\pi(s) = \int\limits_a \pi(a|s) q_\pi(s, a) da$$

2.2 Policy optimization in MDPs

The goal of an agent is to maximize the return, given the state that it is currently in. An optimal policy π^* should be found s.t.

$$\pi^* \doteq \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}_\pi\left[G_0\right] \quad (6)$$

for any $s_0 \in \mathcal{S}$, $a_t \sim \pi(\cdot|s_t)$ and $s_{t+1}, r_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$. The value function under the optimal policy is denoted v_* (i.e. $v_* \doteq v_{\pi^*}$). When the state space is discrete and the environment model is known explicitly, Value Iteration is guaranteed to converge to the optimal policy π^*. However, these requirements rarely hold and convergence might be slow. When it is possible to sample experience from an environment model, but the state transition distributions are not known explicitly, it is possible to use Monte Carlo methods to optimize policies. This involves simulating a number of finite-length trajectories in the environment model (with action selection under policy π) and using the observed cumulative rewards to form an estimate of v_π. Let D be the length of these trajectories and let $r_{t:t+D}$ denote the sequence of observed rewards for some trajectory. The value function is then estimated by[3]

$$\hat{v}_\pi(s_t) = \sum_{\tau=t+1}^{t+D+1} r_\tau \quad (7)$$

[2]The integral is over all permittable $(s, a) \in \Psi$, rather than $\mathcal{S} \times \mathcal{A}$.

[3]A value function estimate is also commonly denoted by V in RL literature.

where s_t is the starting state, $a_t \sim \pi(\cdot|s_t)$ and $s_{t+1}, r_{t+1} \sim p(\cdot|s_t, a_t)$. In this estimate only a finite sequence of rewards is considered. Also, simulations have to complete in order to be able to form an estimate. If we already have an existing estimate of the value function, we can use this to bootstrap our new estimate that does take into account all future states. That is, a D-step temporal difference estimator can be used:

$$\hat{v}_\pi(s_t) = \sum_{\tau=t+1}^{t+D+1} r_\tau + \hat{v}_\pi(s_{t+D+1}) \tag{8}$$

The same temporal distance (TD) principle can be used to construct an estimator for the action-value function $q(s, a)$ (defined in Eq. 5).

$$\hat{q}_\pi(s_t, a_t) = \sum_{\tau=t+1}^{t+D} r_\tau + \hat{q}_\pi(s_{t+D}, a_{t+D}) \tag{9}$$

In practice, optimization using TD estimators was often observed to converge faster than using Monte Carlo estimators, since it is not always required to execute complete episodes until an estimate can be made [23].

2.3 MDP Homomorphisms

Many reinforcement learning objectives can be simplified by considering the right abstractions. MDP homomorphisms [1] are a way to formalize this, where MDPs with smaller state or action spaces are constructed using transformations that preserve the original MDP dynamics. That is, given an MDP

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \Psi, \mathcal{P} \rangle$$

and some state-action transformation function

$$\psi : \mathcal{S} \times \mathcal{A} \to \tilde{\mathcal{S}} \times \tilde{\mathcal{A}}$$

a reduced MDP $\tilde{\mathcal{M}} = \langle \tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \tilde{\Psi}, \tilde{\mathcal{P}} \rangle$ can be constructed as long as the transition reward expectations for both MDPs are equal, as well as the transition probabilities between states $\tilde{s}$ and equivalent state sets in the original MDP. More formally,

$$\forall_{s \in \mathcal{S}, a \in \mathcal{A}} \quad \tilde{\mathcal{P}}(\tilde{s}'|\tilde{s}, \tilde{a}) = \sum_{s' \in \psi^*(\tilde{s}', \tilde{a})} \mathcal{P}(s'|s, a) \tag{10}$$

$$\forall_{s \in \mathcal{S}, a \in \mathcal{A}} \quad \tilde{\mathcal{R}}(\tilde{s}, \tilde{a}) = \mathcal{R}(s, a) \tag{11}$$

where $(\tilde{s}, \tilde{a}) = \psi(s, a)$ and $\psi^*(\tilde{s}, \tilde{a})$ denotes the set of states that ψ maps to $\tilde{s}$. Since the state-action transformation is invariant w.r.t. the reward function and the transition dynamics are preserved, a policy can be constructed in the simpler MDP and can then be used to construct a policy that performs equally

well in the original MDP [1]. To give a concrete example, consider a gridworld with symmetry across a vertical axis where some goal state G should be reached with as little steps as possible.

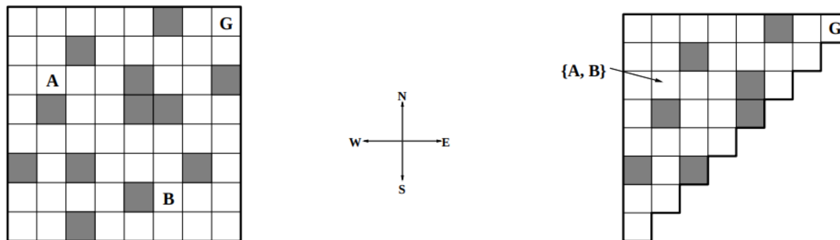


Figure 1: Gridworld example illustrating the reducibility of Reinforcement Learning objectives. (Ravindran et al. [1])

Finding action-value estimations in the reduced MDP (right) is sufficient for estimations in all states of the original MDP (left) if states and actions are mirrored across the diagonal.

2.4 Partial Observability

In the real world, agents can rarely obtain all information about the environment state. Therefore, a generalization of MDPs was made, in which the 4-tuple is extended with a function $\mathcal{O}(o|s, a)$ that gives the probability of receiving some observation o when taking an action a in state s. At each time step, the agent receives such an observation instead of all information about the environment state. This generalization was therefore named 'Partially Observable Markov Decision Process' (POMDP). In a POMDP the agent generally cannot completely derive the environment state from observations alone, even with the Markov assumption. The environment state cannot be measured directly and some information remains hidden. The agent therefore has to use a probability distribution of being in any possible state, given the history leading up to time step t. This distribution reflects the agent's belief about its surroundings and is therefore often referred to as the belief of being in state s_t or

$$bel(s_t) \doteq p(s_t|o_{0:t}, a_{0:t}) \tag{12}$$

where $o_{0:t}$ and $a_{0:t}$ are the sequences of observations and actions up to time t, respectively. This is the belief of being in state s_t after making the observation o_t. This observation is obtained after executing action a_t so often it is useful when planning actions to predict future states (prior to taking action a_t). This is denoted as

$$\overline{bel}(s_t) \doteq p(s_t|o_{0:t-1}, a_{0:t}) \tag{13}$$

2.5 Bayes Filter

The Bayes Filter is a general algorithm for computing the belief over possible states. It does this by recursively computing $bel(s_t)$ from $bel(s_{t-1})$ given the applied action a_t and resulting observation o_t. The filter is derived by induction, where we assume prior belief $bel(s_0)$ is correctly initialized.
Suppose we'd like to know $bel(s_t)$ for some s_t. We can apply Bayes Rule to obtain[4]

$$\begin{aligned} p\left(s_t|o_{0:t}, a_{0:t}\right) &= \frac{p\left(o_t|s_t, o_{0:t-1}, a_{0:t}\right) p\left(s_t|o_{0:t-1}, a_{0:t}\right)}{p\left(o_t|o_{0:t-1}, a_{0:t}\right)} \\ &= \eta p\left(o_t|s_t, o_{0:t-1}, a_{0:t}\right) p\left(s_t|o_{0:t-1}, a_{0:t}\right) \end{aligned}$$

The Markov assumption tells us that if we know state s_t, the history leading up to this state tells us nothing about the future. Therefore,

$$p\left(o_t|s_t, o_{0:t-1}, a_{0:t}\right) = p\left(o_t|s_t\right)$$

and

$$\begin{aligned} p\left(s_t|o_{0:t}, a_{0:t}\right) &= \eta p\left(o_t|s_t\right) p\left(s_t|o_{0:t-1}, a_{0:t}\right) \\ &= \eta p\left(o_t|s_t\right) \overline{bel}\left(s_t\right) \end{aligned}$$

The $\overline{bel}(s_t)$ term can be expanded by applying the rule of total probability and conditioning on possible previous states s_{t-1}

$$\begin{aligned} \overline{bel}\left(s_t\right) &= p\left(s_t|o_{0:t-1}, a_{0:t}\right) \\ &= \int p\left(s_t|s_{t-1}, o_{0:t-1}, a_{0:t}\right) p\left(s_{t-1}|o_{0:t-1}, a_{0:t}\right) ds_{t-1} \end{aligned}$$

We can again simplify this expression by applying the Markov assumption, namely that

$$p\left(s_t|s_{t-1}, o_{0:t-1}, a_{0:t}\right) = p\left(s_t|s_{t-1}, a_t\right)$$

Action a_t cannot be removed from the expression as it occurs after moving to state s_{t-1} and therefore still gives information about state s_t. The Bayes Filter can now be expressed as the following update equations:

$$bel(s_t) = \eta p\left(o_t|s_t\right) \overline{bel}\left(s_t\right) \tag{14}$$

where

$$\overline{bel}\left(s_t\right) = \int p\left(s_t|s_{t-1}, a_t\right) bel(s_{t-1}) ds_{t-1} \tag{15}$$

Any concrete implementation of the Bayes Filter requires additional assumptions about how to compute these probabilities. For example, a Kalman Filter is a type of Bayes Filter for problems with linear dynamics.

[4]Since $p(o_t|o_{0:t}, a_{0:t})$ is constant for all states s_t it does not affect the problem solution and is denoted as η.

2.6 Recurrent State Space Model

Generally, the environment dynamics will be highly non-linear and Kalman Filters cannot be used. A common approach is to use auto-encoders to update the state belief distribution [24, 10, 12]. Auto-encoders are a type of neural network architecture that are trained to represent their input as a low-dimensional representation. That is, it attempts to capture the features that are most informative about its input. The Recurrent State Space Model aims to learn a model of the environment and uses an encoder to update its internal state. This internal state cannot be observed from the environment directly and is thus inferred from the observations. It is therefore called a latent state representation. The RSSM has repeatedly formed the basis of state-of-the-art models in reinforcement learning [19, 21, 22]. For some POMDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \Psi, \mathcal{P}, \mathcal{O} \rangle$ an approximate model $\tilde{\mathcal{M}} = \langle \tilde{\mathcal{S}}, \tilde{\mathcal{A}}, \tilde{\Psi}, \tilde{\mathcal{P}}, \tilde{\mathcal{O}} \rangle$ is learned, where

- $\tilde{\mathcal{S}}$ - is the set of possible latent states of the model. Since $\mathcal{S}$ is generally not known, the number of dimensions of the latent states must be chosen.
- $\tilde{\mathcal{A}}$ - is the set of possible actions that can be applied to the model approximation and is chosen to be equal to $\mathcal{A}$.
- $\tilde{\Psi}$ - No restrictions are made for which actions can be applied to which latent state, so $\tilde{\Psi} = \tilde{\mathcal{S}} \times \tilde{\mathcal{A}}$.
- $\tilde{\mathcal{P}}$ - controls the transitions between the latent states, given the chosen actions. In the RSSM, the latent state $\tilde{s}_t \in \tilde{\mathcal{S}}$ consists of two components (h_t, z_t). h_t is the internal state of a recurrent neural network $\tilde{\mathcal{P}}_{det}$ and forms a deterministic component of $\tilde{s}_t$. Based on h_t, a stochastic state model $\tilde{\mathcal{P}}_{sto}(\cdot|h_t)$ generates the z_t state component. That is, every time step t, $\tilde{s}_t = (h_t, z_t) = (\tilde{\mathcal{P}}_{det}(h_{t-1}, z_{t-1}, a_{t-1}), \tilde{\mathcal{P}}_{sto}(h_t))$, corresponding to Eq. 15 in the Bayes Filter. A separate reward model then computes the reward r_t based on $\tilde{s}_t$.
- $\tilde{\mathcal{O}}$ - a decoder that generates an observation based on the latent state $\tilde{s}_t$.

Additionally, to update the state belief distribution prior to obtaining the observation from the environment (Eq. 14 of the Bayes Filter), an encoder model $q(\tilde{s}_t|h_t, o_t)$ is used to reparameterize the distribution.

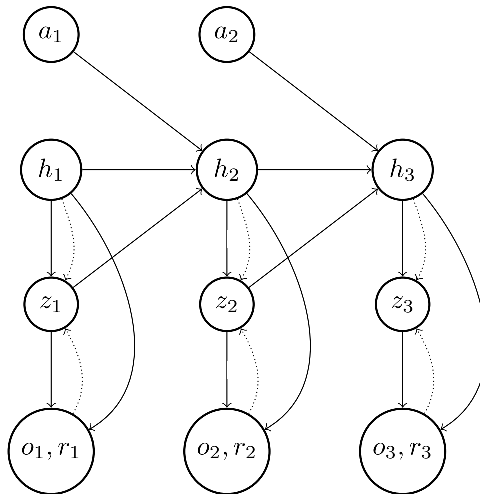


Figure 2: Dependency graph of the RSSM. The dotted arrows indicate the information the encoder uses to update the prior belief distribution upon receiving an observation.

The RSSM aims to mimic the environment and thus follows the POMDP structure. That is, at each time step an action a_t is provided by the agent and an observation o_{t+1} and reward r_{t+1} are returned. The RSSM has an internal state $\tilde{s}_t$ on which it bases the observations and rewards. The authors chose to split the internal state into a deterministic part h_t and stochastic part z_t. The deterministic part is a hidden state of a recurrent neural network giving the state transition model $h_t = \tilde{\mathcal{P}}_{det}(h_{t-1}, z_{t-1}, a_{t-1})$. The stochastic state model then gives $z_t \sim \tilde{\mathcal{P}}_{sto}(\cdot|h_t)$. Based on this internal state $\tilde{s}_t = (h_t, z_t)$ an observation model and reward model give $o_t \sim \tilde{\mathcal{O}}(\cdot|\tilde{s}_t)$ and $r_t \sim \tilde{\mathcal{R}}(\cdot|\tilde{s}_t)$, respectively.
The reward, observation and stochastic state model are all Gaussians parameterized by a neural network (see section 9). The former two have a fixed identity covariance.

2.6.1 Training objective

To evaluate the environment model's correctness, the log-likelihood of the data given the current parameterization of the model is computed. This, however, cannot be optimized directly, so it is common practice to consider a lower bound (referred to as the evidence lower bound or ELBO) obtained through Jensen's inequality (see derivation below). Also, the importance sampling trick used in Section 2.8 is used to express the bound in terms of our variational belief

distribution $q(s_{0:T}|o_{0:T}, a_{0:T}) = \prod_{t=1}^{T} q(s_t|s_{t-1}, a_{t-1}, o_t)$:

$$
\begin{aligned}
\ln p(o_{0:T}, r_{1:T}|a_{0:T}) &= \ln \int \prod_{t=0}^{T} p(o_t|s_t)p(r_t|s_t)p(s_t|s_{t-1}, a_{t-1})ds_{0:T} \\
&= \ln \mathbb{E}_{p(s_{0:T}|a_{0:T})}[\prod_{t=0}^{T} p(o_t|s_t)p(r_t|s_t)] \\
&= \underbrace{\ln \mathbb{E}_{q(s_{0:T}|o_{0:T},a_{0:T})}[\prod_{t=0}^{T} p(o_t|s_t)p(r_t|s_t)\frac{p(s_t|s_{t-1}, a_{t-1})}{q(s_t|o_{0:t}, a_{0:t-1})}]}_{\text{Importance sampling}} \\
&\geq \underbrace{\mathbb{E}_{q(s_{1:T}|o_{0:T},a_{0:T})}[\sum_{t=0}^{T} \ln p(o_t|s_t) + \ln p(r_t|s_t) + \ln \frac{p(s_t|s_{t-1}, a_{t-1})}{q(s_t|o_{0:t}, a_{0:t-1})}]}_{\text{Jensen's inequality (since the natural logarithm is concave)}} \\
&= \sum_{t=0}^{T} \mathbb{E}_{q(s_t|o_{0:t},a_{0:t-1})}[\ln p(o_t|s_t)] + \mathbb{E}_{q(s_t|o_{0:t},a_{0:t-1})}[\ln p(r_t|s_t)] \\
&\quad + \mathbb{E}_{q(s_t|o_{0:t},a_{0:t-1})}[\ln \frac{p(s_t|s_{t-1}, a_{t-1})}{q(s_t|o_{0:t}, a_{0:t-1})}] \\
&= \sum_{t=0}^{T} \underbrace{\mathbb{E}_{q(s_t|o_{0:t},a_{0:t-1})}[\ln p(o_t|s_t)]}_{-\mathcal{L}_o} + \underbrace{\mathbb{E}_{q(s_t|o_{0:t},a_{0:t-1})}[\ln p(r_t|s_t)]}_{-\mathcal{L}_r} \\
&\quad - \underbrace{\mathrm{KL}(q(s_t|o_{0:t}, a_{0:t-1})||p(s_t|s_{t-1}, a_{t-1}))}_{\mathcal{L}_{\mathrm{KL}}}
\end{aligned}
$$

Since the observation model of the RSSM is modeled as a Gaussian with identity covariance, maximizing the log-likelihood is equivalent to minimizing the mean squared error. The same holds for the reward model (see derivation below). Together with the KL-divergence, these make up three loss components $\mathcal{L}_o, \mathcal{L}_r, \mathcal{L}_{\mathrm{KL}}$ which are all weighted equally.

$$
\begin{aligned}
\ln(p(r_t|s_t)) &= \ln \mathcal{N}(r_t; \mu_t, \sigma) \\
&= \ln \left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{r_t-\mu_t}{\sigma}\right)^2} \right) \\
&= \ln(\frac{1}{\sigma\sqrt{2\pi}}) + \ln(e^{-\frac{1}{2}\left(\frac{r_t-\mu_t}{\sigma}\right)^2}) \\
&= \ln(\frac{1}{\sigma\sqrt{2\pi}}) - \frac{1}{2}\left(\frac{r_t-\mu_t}{\sigma}\right)^2
\end{aligned}
$$

where μ_t is chosen by the RSSM based on state belief s_t and σ is set to 1. Minimizing this expression is equivalent to minimizing the squared error up to a constant.

2.7 Bias-Variance Trade-off

The mean squared error (MSE) loss is often used to describe the bias-variance trade-off also discussed in the introduction. By introducing biases in the model, it can be more robust against variability in the data and decrease the expected prediction error. First, we can rewrite the expected squared error given that we are in some state s_t as follows:

$$\begin{aligned}
\mathbb{E}_{r_t}[(r_t - \mu_t)^2|s_t] &= \mathbb{E}_{r_t}[r_t^2 - 2r_t\mu_2 + \mu_t^2|s_t] \\
&= \mu_t^2 - 2\mu_t\mathbb{E}_{r_t}[r_t|s_t] + \mathbb{E}_{r_t}[r_t^2|s_t] \\
&= \mu_t^2 - 2\mu_t\mathbb{E}_{r_t}[r_t|s_t] + \mathbb{E}_{r_t}[r_t|s_t]^2 + \mathrm{Var}[r_t|s_t] \\
&= (\mu_t - \mathbb{E}_{r_t}[r_t|s_t])^2 + \mathrm{Var}[r_t|s_t]
\end{aligned}$$

It now becomes apparent that the best prediction we can hope to achieve is $\bar{\mu}_t \doteq \mathbb{E}_{r_t}[r_t|s_t]$, minimizing the first term. The second term is independent of the model prediction and is therefore irreducible. This is referred to as the Bayes Error. If we now take the expectation of the loss expression w.r.t. the state s_t we obtain

$$\begin{aligned}
\mathbb{E}_{s_t}[(r_t - \mu_t)^2] &= \mathbb{E}_{s_t}[(\mu_t - \bar{\mu}_t)^2 + \mathrm{Var}[r_t|s_t]] \\
&= \mathbb{E}_{s_t}[\mu_t^2 - 2\mu_t\bar{\mu}_t + \bar{\mu}_t^2] + \mathbb{E}_{s_t}[\mathrm{Var}[r_t|s_t]] \\
&= \bar{\mu}_t^2 - 2\bar{\mu}_t\mathbb{E}_{s_t}[\mu_t] + \mathbb{E}_{s_t}[\mu_t^2] + \mathbb{E}_{s_t}[\mathrm{Var}[r_t|s_t]] \\
&= \bar{\mu}_t^2 - 2\bar{\mu}_t\mathbb{E}_{s_t}[\mu_t] + \mathbb{E}_{s_t}[\mu_t]^2 + \mathrm{Var}[\mu_t] + \mathbb{E}_{s_t}[\mathrm{Var}[r_t|s_t]] \\
&= \underbrace{(\bar{\mu}_t - \mathbb{E}_{s_t}[\mu_t])^2}_{\text{Bias}^2} + \underbrace{\mathrm{Var}[\mu_t]}_{\text{Variance}} + \underbrace{\mathbb{E}_{s_t}[\mathrm{Var}[r_t|s_t]]}_{\text{Bayes Error}}
\end{aligned}$$

The expected loss is composed of three terms which each have their interpretation:

1. Bias - Difference between the expected model prediction and the target. Measures how well the model captures relevant relations between features in the dataset.
2. Variance - Measures how the choice of s_t affects the deviation from the average model prediction and thus how sensitive the model is to some selection of s_t.
3. Bayes Error - irreducible error component independent of the model predictions.

Choosing the right model amounts to a trade-off in expressivity (low bias), while being robust against variation in the data (low variance). The derivation is specific to the squared error loss, but the intuition remains useful for other loss functions.

2.8 Cross-Entropy Method

This section is a summary of an extensive tutorial on the cross-entropy method, provided by De Boer et al. [25]. Its relation to finding optimal policies in MDPs is described in subsection 2.8.1.

The cross-entropy method is an optimization method for estimating a quantity

$$\ell = \mathbb{E}[H(X)] = \int H(x) f(x; \mathbf{u}) dx \tag{16}$$

where X is some random variable with known pdf $f(\cdot; \mathbf{u})$ parameterized by parameter vector $\mathbf{u}$. H(x) is some performance function (e.g. the total reward obtained for performing some action x, in the reinforcement learning setting). The most straightforward way of estimating ℓ is to take a crude Monte-Carlo approach. That is, we take N random samples $X_1, ..., X_N$ from $f(\cdot; \mathbf{u})$ and then compute

$$\hat{\ell} = \frac{1}{N} \sum_{i=1}^{N} H(X_i) \tag{17}$$

A problem arises, however, if there are rare events that are influential to the value of ℓ. Then many simulations (samples from $f(\cdot; \mathbf{u})$) are required to estimate ℓ accurately. An alternative approach is to use importance sampling. That is, $X_1, ..., X_N$ are sampled from a different pdf g on the same space and ℓ is evaluated using the (unbiased) likelihood estimator

$$\hat{\ell} = \frac{1}{N} \sum_{i=1}^{N} H(X_i) \frac{f(X_i; \mathbf{u})}{g(X_i)} \tag{18}$$

Under this different distribution g the probability of obtaining informative samples can be much higher and thus ℓ can be estimated more accurately with a limited number of samples drawn. Now the question remains which distribution g should be used. The Cross-Entropy Method is a way to iteratively select such distributions. The optimal distribution g^* would be

$$g^*(x) := \frac{H(x) f(x; \mathbf{u})}{\ell} \tag{19}$$

as this would just result in using ℓ as its own estimator. The problem is that this value is unknown. What we can do, however, is to try to find some distribution closest to g^*. The idea is to select some $f(\cdot; \mathbf{v})$ such that the distance between $f(\cdot; \mathbf{v})$ and g^* is minimized. The distance measure used is the Kullback-Leibler divergence $\mathcal{D}(g^*, f(\cdot; \mathbf{v}))$, which is defined as

$$\mathcal{D}(g^*, f(\cdot; \mathbf{v})) = \mathbb{E}_{g^*} \ln \frac{g^*(X)}{f(X; \mathbf{v})} = \int g^*(x) \ln g^*(x) dx - \int g^*(x) \ln f(x; \mathbf{v}) dx \tag{20}$$

Since $\mathbf{v}$ does not influence the left term, minimizing $\mathcal{D}(g^*, f(\cdot; \mathbf{v}))$ is equivalent to minimizing the right term $(-\int g^*(x) \ln f(x; \mathbf{v}) dx)$, which is also referred to as the cross-entropy between g^* and $f(\cdot; \mathbf{v})$. This is then equivalent to the following maximization problem

$$\begin{aligned}\max_{\mathbf{v}} D(\mathbf{v}) &= \max_{\mathbf{v}} \int g^*(x) \ln f(x; \mathbf{v}) dx \\ &= \max_{\mathbf{v}} \int \frac{H(x) f(x; \mathbf{u})}{\ell} \ln f(x; \mathbf{v}) dx \\ &= \max_{\mathbf{v}} \int H(x) f(x; \mathbf{u}) \ln f(x; \mathbf{v}) dx\end{aligned}$$

Note that this does not depend on the unknown value ℓ. The optimal set of parameters $\mathbf{v}^* = \text{argmax}_{\mathbf{v}} D(\mathbf{v})$ can be estimated by taking random samples $X_1, ..., X_N$ from $f(\cdot; \mathbf{u})$ and solving

$$\max_{\mathbf{v}} \hat{D}(\mathbf{v}) = \max_{\mathbf{v}} \frac{1}{N} \sum_{i=1}^{N} H(X_i) f(X_i; \mathbf{u}) \ln f(X_i; \mathbf{v}) \tag{21}$$

However, we run into the same problem. Namely, that the probability of informative events occurring can be very small and that we therefore need many samples to accurately estimate $\mathbf{v}^*$. We can use the importance sampling trick again, and introduce a different distribution $f(\cdot; \mathbf{w})$ which can be sampled from and estimate $\mathbf{v}^*$ using

$$\max_{\mathbf{v}} \hat{D}(\mathbf{v}) = \max_{\mathbf{v}} \frac{1}{N} \sum_{i=1}^{N} H(X_i) \frac{f(X_i; \mathbf{u})}{f(X_i; \mathbf{w})} \ln f(X_i; \mathbf{v}) \tag{22}$$

where $X_1, ..., X_N$ are sampled from $f(\cdot; \mathbf{w})$. The parameters $\mathbf{w}$ are updated in an iterative manner to be the current best estimate of $\mathbf{v}$. First, we sample $X_1, ..., X_N$ from initial configuration $f(\cdot; \mathbf{w}_0) = f(\cdot; \mathbf{u})$. Then we create an ordering of the samples, based on how well they score on the performance function H. We then select the top K samples with $0 < K < N$ and let $\mathbf{w}_1$ be the set of parameters which is optimal for estimating that X scores as well as the closest samples. That is, if $X^{(k)}$ for $1 \leq k \leq K$ denote top K samples, then w_1 is selected by trying to maximize

$$\mathbf{w}_1 = \underset{\mathbf{v}}{\text{argmax}} \frac{1}{K} \sum_{k=1}^{K} H(X^{(k)}) \frac{f(X^{(k)}; \mathbf{u})}{f(X^{(k)}; \mathbf{w}_0)} \ln f(X^{(k)}; \mathbf{v}) \tag{23}$$

Now, hopefully, under $f(\cdot; \mathbf{w}_1)$ the quantity ℓ can be estimated more accurately. If we then sample $f(\cdot; \mathbf{w}_1)$ N times and select the K closest samples, we can follow the same process to obtain an even better set of parameters $\mathbf{w}_2$. This process continues for a fixed number of iterations.

2.8.1 Relation to policy optimization in MDPs

The Cross-Entropy Method can be used to learn policies for maximizing rewards in MDPs. A randomly initialized policy is constructed and iteratively updated to obtain higher rewards from the trajectories that are sampled. Let D be the number of steps simulated in a trajectory and let X be a random variable over possible action sequences $a_{t:t+D}$ in the trajectories. Then $f(X;\mathbf{w}_0)$ is randomly initialized as some pdf over the action sequences, making it the initial policy.
$f(X;\mathbf{w}_0)$ is sampled for N action sequences $a^{(n)}_{t:t+D}$. Each of the action sequences is executed in the environment model to obtain reward sequences $r^{(n)}_{t+1:t+D+1}$. Each action sequence is evaluated by the performance function, which in this case is an estimate of the value function: $H(a_{t:t+D}) = \hat{v}_\pi(s_t) = \sum_{\tau=t+1}^{t+D+1} r_\tau$.
The K action sequences that perform best are used to update the policy parameters through Eq. 23. Under this policy the same procedure can be repeated until convergence.

2.9 Data Augmentation

Data augmentation is the process of creating artificial data samples based on existing samples obtained from the true data distribution. These samples can be created by modifying the original samples in ways that do not change its corresponding model target output. For example, in image classification problems, sample images can be augmented by shifting the image for some number of pixels. For most classification problems, the target label will remain the same for the newly created image, therefore extending the available dataset. In reinforcement learning, the objective (Eq. 6) is to find a policy that maximizes the value function $v_\pi(s)$. Let $\phi : \mathcal{S} \times \mathcal{T} \rightarrow \mathcal{S}$ denote a function that creates an augmented environment state s.t.

$$v_\pi(s) = v_\pi(\phi(s,\nu)), \text{ for all } s \in \mathcal{S} \text{ and } \nu \in \mathcal{T}. \tag{24}$$

Here, ν denotes a parameterization of ϕ, taken from the set of possible parameterizations $\mathcal{T}$. ϕ can be used to generate additional data samples for training a value function estimator, assuming that it is invariant w.r.t. the value function or similarly, the action-value function:

$$q_\pi(s,a) = q_\pi(\phi(s,\nu),a), \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}, \text{ and } \nu \in \mathcal{T}, \tag{25}$$

Alternatively, data augmentations can be defined over state-action pairs rather than single states:

$$q_\pi(s,a) = q_\pi(\psi(s,a,\nu)), \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}, \text{ and } \nu \in \mathcal{T}, \tag{26}$$

where $\psi : \Psi \times \mathcal{T} \rightarrow \Psi$ augments state-action pairs, rather than states. This broadens the space of possible data augmentations, since it is a generalization of Eq. 25. The chosen augmentations π or ψ can then be used to generate

value estimates with lower variance, by averaging over multiple states that are assumed to have equal expected return.

$$\hat{q}_\pi(s,a) \approx \frac{1}{M}\sum_{m=1}^{M} \hat{q}_\pi\left(\phi\left(s,\nu_m\right),a\right) \text{ where } \nu_m \in \mathcal{T} \tag{27}$$

2.9.1 Regularization

The PlaNet algorithm plans in the RSSM using the cross-entropy method to select actions for data collection. Given any data augmentation ψ with invariance w.r.t. the state-action value (i.e. Eq. 26), an estimator for $q(s,a)$ can be regularized by averaging over the newly generated samples, resulting in a lower variance estimate:

$$\hat{q}_\pi(s,a) \approx \frac{1}{M}\sum_{m=1}^{M} \hat{q}_\pi\left(\psi\left(s,a,\nu_m\right)\right) \text{ where } \nu_m \in \mathcal{T} \tag{28}$$

There are several ways in which this estimate can be used in the PlaNet algorithm:

1. The RSSM is trained using stochastic gradient descent, requiring sample episodes obtained from the true environment. The most obvious use of data augmentation is to generate artificial sample episodes from existing ones, effectively increasing the amount of data available. Important to note, however, is that the same parameterization ν should be used for all time steps in the trajectory, since the transition function is otherwise not accurately represented.

2. A TD estimator (Eq. 9) can regularized by substituting Eq. 28 to obtain

$$\hat{q}_\pi(s_t,a_t) = \sum_{\tau=t+1}^{t+D} r_\tau + \frac{1}{M}\sum_{m=1}^{M} \hat{q}_\pi\left(\psi\left(s_{t+D},a,\nu_m\right)\right) \tag{29}$$

   When this estimator is used in the CEM, additional states are generated at the end of the trajectory, which are equivariant w.r.t. the q-function according to the definition of ψ. This estimate is less sensitive to inaccuracies in the action-value function model.

3. The learned action-value function $\hat{q}_\pi(s,a;\theta)$ parameterized by θ is trained using SGD as well. Given q-value targets y_i for state action pair (s_i,a_i), the model parameters are updated using

$$\theta \leftarrow \theta - \alpha\nabla_\theta \mathcal{L}(\hat{q}(s_i,a_i;\theta),y_i) \tag{30}$$

   where α is a learning rate parameter. The state action pair (s_i,a_i) and target y_i are obtained from a dataset $\mathcal{D}$. Namely, a tuple (s_i,a_i,r_i,s_i',a_i')

from $\mathcal{D}$ contains state action tuple (s_i, a_i), the resulting reward r_i and subsequent state action tuple (s'_i, a'_i). The target value y_i is usually computed using

$$y_i = r_i + \hat{q}_\pi(s'_i, a'_i; \theta) \tag{31}$$

This can, however, also be regularized by substituting Eq. 27, to obtain

$$y_i = r_i + \sum_{m=1}^{M} \hat{q}_\pi \left(\phi\left(s'_i, \nu_m\right), a'_i\right) \tag{32}$$

3 Related Work

Current model free reinforcement learning algorithms, such as PPO [26], TRPO [27], DDPG [28] and SAC [29], are able to build policies for complex tasks, but require many samples from the environment [7, 30, 6]. Model based reinforcement learning improves sample efficiency by using a model of the environment to obtain simulated data [31, 17, 32, 33]. Current work focuses on building algorithms that learn an environment model from the observations that they obtain from their surroundings. Many different approaches are taken, such as building locally linear approximations and using Kalman filters or LQR for optimal control in these approximations [34, 35, 36, 37, 38]. Others use Gaussian processes to model the environment dynamics [17, 39, 40]. Advantages of these methods is that they are simple and data efficient, because of the strong assumptions they make of their surroundings [41, 32]. They are also less expressive, however, when compared to models based on neural networks. These models are able to represent the environment more accurately, but require much more data and training time to do so. The models were observed to overfit on the limited data available in the RL setting [32, 3]. What the methods have in common is that they all base action selection on an approximation of the environment. This introduces a bias toward the environment model, rather than the real environment [17, 42]. Therefore, model based RL often has lower asymptotic performance when compared to model free RL with sufficient data [32, 18]. Chua et al. [32] observe that the quality of the learned environment model is crucial for the performance of model based RL methods.

3.1 Planning in learned environment models

In terms of data efficiency and asymptotic performance, model predictive control (MPC) in the learned environment models has shown to be a promising method for control [43, 17, 7, 18]. The authors of PETS [32] claim their model requires 8 times less data than SAC, and 125 times less than PPO (which are both model free methods). Using uncertainty estimates in their environment model, they were able to deal with model approximation errors while planning. POPLIN [7] was shown to be even more data efficient, by using policy networks for planning, rather than sampling random sequences. Currently, PlaNet [19] is considered among the state-of-the-art model based RL algorithms for control. PlaNet learns a non-linear recurrent state-space model (RSSM) and uses model-predictive control to do planning in latent space. More specifically, they use the Cross-Entropy method (CEM) to simulate trajectories in the latent space model and select an optimal one. A downside of this is that, since the planning relies on rollouts with a limited number of steps, the computed policy is only suitable in a finite horizon. This limitation has been observed by multiple people [42, 44, 18]. The authors of PlaNet claim this could be mitigated by training a neural network to learn the value function, as it tries to predict the expected cumulative reward obtained after reaching each state. To our knowledge this has not been tried.

3.2 MDP homomorphisms

Learned environment models aim to find a simplified and lower-dimensional representation that is equally useful for policy optimization in the original environment. Therefore, it omits information that is not influential to the performance of the policy (i.e. the value function). 'Naive' approaches in which encoders attempt to learn useful abstractions were shown to overfit on the data [3]. A more explicit definition of what constitutes a useful abstraction was made through MDP homomorphisms [1, 45], in which the invariance of certain state-action pairs w.r.t. the value function is used to minimize the MDP[5]. This invariance property is also used in data augmentation to generate new samples [3, 2]. Constructing algorithms to find MDP homomorphisms is a difficult problem and remains an active focus of research [46, 47, 48], as well as ways of enforcing the invariance properties in deep RL [49].

3.3 Data augmentation

Data augmentation is a commonly used technique in supervised learning for regularization of neural networks [50, 51, 52]. The early successes of convolutional neural networks (such as AlexNet [53]) already showed the beneficial effects of using augmented images during training. Augmentation can be used to generate synthetic data that can resolve class imbalance in datasets or just increase the dataset size [54]. A wide variety of augmentations have been used, such as flipping, cropping, translation, rotation, recoloring and noise injection [55]. Compositions of augmentation functions are often valid augmentations as well, making the collection of possible augmentations extensive. Since the applicability of augmentations is also dependent on the problem being solved, choosing the right augmentations is a complex task. This motivates the use of adversarial training, such as GANs, in which a model is tasked specifically with finding effective augmentations [56, 57]

3.3.1 Data augmentation in reinforcement learning

Many reinforcement learning algorithms successfully apply convolutional neural networks for identifying features in state observations and use these to construct policies [4, 5, 28]. These methods, however, still do not perform well in terms of data efficiency and their ability to generalize to new environments [2]. An extensive period of data collection is needed to obtain good policies. Data augmentation has traditionally been recognized as a method of improving data efficiency of CNNs. Early successful applications of CNNs (LeNet [51], AlexNet [53]) already made use of augmented data to force the network to be invariant to certain transformations (e.g. rotation, reflection). Also, the image's invariance

[5]For example, consider a grid maze that is symmetrical along some axis. Solving the maze on one side also gives the solution on the other side, since the solution can be mirrored. This means the state-action pairs of these solutions are invariant to the mirroring procedure. Knowing this invariance, policies in a MDP homomorphism can be 'lifted' to policies in the original MDP with equal performance.

to spatial shifts is the main motivation behind CNN architectures. Recent work suggests that data augmentation is undervalued in the reinforcement learning setting. Kostrikov et al. [3] surpass the state of the art algorithms by simply regularizing a soft actor-critic's action value model through data augmentation. Laskin et al. [2] introduce RAD, a module for data augmentation in reinforcement learning, with which they show that RL significantly benefits from augmented data. Another recently introduced model called CURL (Srinivas et al. [58]) achieves better data efficiency by making use of data augmentation. Earlier work by Cobbe et al. [13] tries to quantify the generalization ability of RL algorithms using their CoinRun environment. They show several regularization methods, including data augmentation, improve generalization in RL. However, why certain data augmentation procedures are effective remains unclear.

4 Methodology

This section covers the approach taken in this research to investigate the research questions, with the primary focus being:
RQ 1: How does regularizing the Recurrent State Space Model used in PlaNet through data augmentation affect its performance in terms of data efficiency and asymptotic performance?
To address this, some preliminary questions are discussed, namely

1.1 What is PlaNet? (Section 4.1)
1.2 What type of data augmentation should be used? (Section 4.2)
1.3 How can data augmentation be incorporated in the algorithm? (Section 4.2.3)
1.4 How can the model performance be evaluated? (Section 4.3)

The secondary focus of this research is as follows:
RQ 2: How does the regularization procedure implemented by Kostrikov et al. affect the PlaNet algorithm when it is augmented with a value function?
If the algorithm can learn a value function model to optimize over the expected return rather than a finite sequence of rewards the value function can be regularized through data augmentation in the same way that proved successful for Kostrikov et al. [3].
Section 4.4 describes the experiments that are done to investigate these questions. First, the general setup of the algorithm is covered.

4.1 PlaNet

The PlaNet algorithm uses the Cross-Entropy Method to plan in a learned environment model called the Recurrent State Space Model (RSSM). The actions that obtained the highest rewards in the RSSM are selected to be executed in the real environment. At every time step, the agent plans ahead to select an action. The trajectories that follow from this are used as data to improve the RSSM. The improved RSSM can then be used to collect better data samples through more informative planning. The PlaNet pseudocode can be found in Appendix 10. The Cross-Entropy method is used to plan in the environment models and is explained in Section 2.8. The policy distribution is chosen to be Gaussian and iteratively reparameterized accordingly. Pseudocode can be found in Appendix 10. The RSSM is summarized in section 2.6 with appendix 9 provides additional details regarding the design choices and implementation of the RSSM.

4.2 Data Augmentation

This section covers how data augmentation is applied in the PlaNet algorithm. The procedures used for augmentation are described, as well as how the augmented data is used.
First, it should be noted that what constitutes as a valid data augmentation

procedure depends on the agent objective. In this work the problem of balancing an inverted pendulum is considered. At each time step, the agent is allowed to apply a torque at the base of the pendulum and receives a reward based on the angle the pendulum makes with the upright position. Smaller angles result in larger rewards. The agent also obtains an observation giving information about the environment state. This environment state is described by the angle of the pendulum, as well as its velocity and acceleration. However, these are not observed directly and should thus be inferred from the observations.

4.2.1 Mirror Augmentation

The balancing objective is invariant to mirroring across the vertical axis (Figure 3). The angle that the pendulum makes with the upright position does not change, and therefore also the reward at each time step. When actions are mirrored as well, complete episodes of data can be augmented this way. The "mirror" data augmentation is therefore one of the procedures used in this research. That is, when data is sampled from the replay memory, there is a 0.5 probability that it will be mirrored.

Figure 3: Example of state-action based data augmentation on a pendulum environment. Despite being two different states, the rewards obtained are equivalent if the actions are mirrored, effectively doubling the data that is available.

4.2.2 Translate Augmentation

One of the papers introducing the spike of interest in data augmentation evaluated several data augmentation procedures and provided a module (RAD) with implementations of these procedures [2]. The observation-based augmentations tested included rotation, translation, flipping, cropping, cutouts and various coloring schemes. The environment states were augmented by addition of Gaussian noise and random scaling. From their experiments, random translation (Figure 4) was found to be most effective by a large margin. It is, however, not clear why this augmentation procedure was more effective. The random translate procedure is therefore chosen as the second data augmentation used in this research.

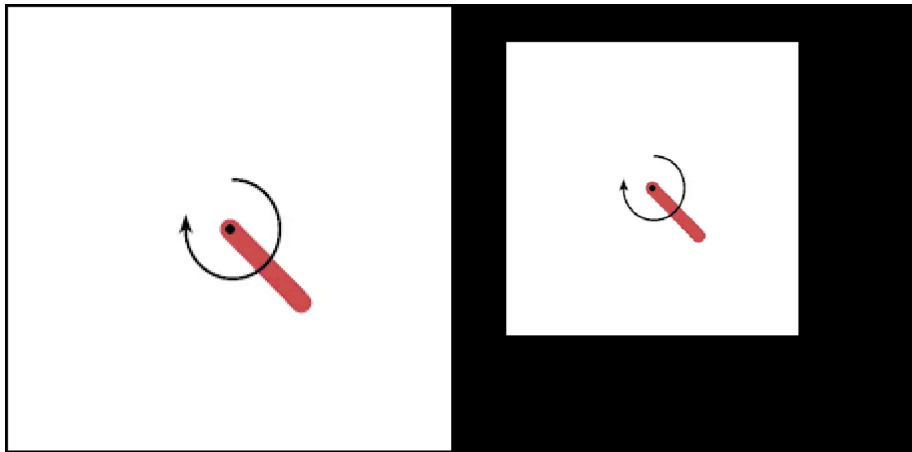


Figure 4: Random translate data augmentation procedure. The original image (left) is placed randomly in a black square (right). The original image retains its size so the augmented image is larger. The encoder architecture is chosen such that applying the convolutional layers on both image sizes gives 256 2x2 feature maps for encodings of the same size.

4.2.3 Regularization

As described in Section 2.9.1, there are various ways in which data augmentation can be incorporated into the PlaNet algorithm. The primary distinction is that data augmentation can be used during train time for generating additional data, or during test time for variance reduction of the CEM rollouts. In this work, we use data augmentation during train time. The model is trained using gradient descent while averaging gradients over augmented sample trajectories.

4.3 Evaluation

To evaluate the performance of reinforcement learning algorithms, several benchmarks were introduced. These typically consist of various toy problems that the agent needs to solve, each with different challenges. The PlaNet algorithm was originally tested on six control tasks of the DeepMind control suite [59]. Recently, this benchmark was extended to 20 environments with the introduction of the Dreamer algorithm [21], for a more thorough evaluation. All environments in these benchmarks are based on the MuJoCo physics engine. Being unable to obtain a license for MuJoCo, this research was restricted to the use of other environments in the commonly used OpenAI Gym [60]. More specifically, we evaluate the RSSM on the pendulum environment, for its simplicity and clear invariance towards mirroring over a vertical axis (see Figure 3).

4.4 Experiment Setup

This section provides a description of the setup of the experiments that were run during this research. Additional settings and hyperparameters can be found in Appendix 9.
Due to practical considerations the experiment setup has been limited in some aspects. With the primary limitation being the computation power available, the experiments were designed accordingly. Data collection was chosen to happen under a uniformly random policy, rather than the CEM. The CEM is only used to evaluate the model performance. This does mean that the agent is not able to explicitly search for data with a high reward expectation, and limits the agent's ability to deal with sparsity in the environment's reward function. However, with planning being the most demanding procedure in terms of computation, this does save a lot of time. Furthermore, the effectiveness of data augmentation when learning environment models can still be investigated.

4.4.1 Experiment 1 - Performance Upper Bound

To obtain an upper bound of the performance of the algorithm the CEM is used to plan in the true pendulum environment model. No environment approximation is learned, but a baseline is formed for what could be achieved in a perfect model.

4.4.2 Experiment 2 - Environment Model and Data Augmentation

In this experiment, the environment model approximations (RSSMs) are learned with and without the use of data augmentations. The "mirror" and "translate" augmentations are considered separately. The data used to train the RSSM consists of 1000 episodes which are sampled from the true environment model under a uniformly random policy and stored in a dataset. Each model is trained using 200 iterations of gradient descent (see Section 2.6.1). The model is then evaluated by using the CEM to plan in the model approximation, to select actions in the true environment. The sum of rewards (or return) obtained from the true environment shows the performance of the algorithm.
Hafner et al. assume an initial state when collecting the data for training, such that the initialization of the latent state of the RSSM corresponds to this initial state. Since this assumption is not compatible with some of the data augmentations used, random initial states are used. Instead, the encoder sets the state using the initial observation.

4.4.3 Experiment 3 - Test for Invariance

To evaluate how well the models respect the invariance property used to perform data augmentation, the difference between reward predictions of invariant state-action pairs is considered. That is, the learned environment model is presented with a trajectory of (s_t, a_t) pairs to give reward predictions $\tilde{r}_{t+1}^{(1)}$. Subsequently, the model is given a trajectory that consists of $\psi(s_t, a_t)$, which are state-action

pairs that have an equal reward expectation. For this trajectory, the model will give predictions $\tilde{r}^{(2)}_{t+1}$, from which a difference is computed: $\Delta\tilde{r}_t = \tilde{r}^{(1)}_t - \tilde{r}^{(2)}_t$. A trained model in which the invariance property ψ is embedded will give a reward difference $\Delta\tilde{r} = 0$. The question this experiment tries to answer is whether there is any difference between the $\Delta\tilde{r}$ of models trained with or without data augmentation.
To mitigate the effects of randomness in weight initialization and the optimization procedure, six models are trained and evaluated both with and without data augmentation. Due to constraints in computation power, the models were downscaled in comparison to experiment 2. The hyperparameters of the downscaled models can be found in appendix 9.3.

4.4.4 Experiment 4 - Value Function

The algorithm is tasked with training the RSSM as well as a value function model. This model is used during planning using the CEM. The value function model is trained both with and without data augmentation.

5 Results and Discussion

This section presents the results obtained from the experiments run during this research. All run log files remain existent and consist of the rewards and observations obtained and predicted during evaluation, as well as training[6]. Additional information regarding hyperparameters can be found in Appendix 9.

5.1 Experiment 1 - Performance Upper Bound

To form a baseline, the Cross-Entropy Method is used to plan in the true environment model. At each environment step, 1000 simulations in the true pendulum environment model are executed. The cumulative rewards of all trajectories is used to reparameterize the policy distribution. Since the objective of the learned environment model is to accurately represent the true model, any deviations will generally decrease performance during planning.

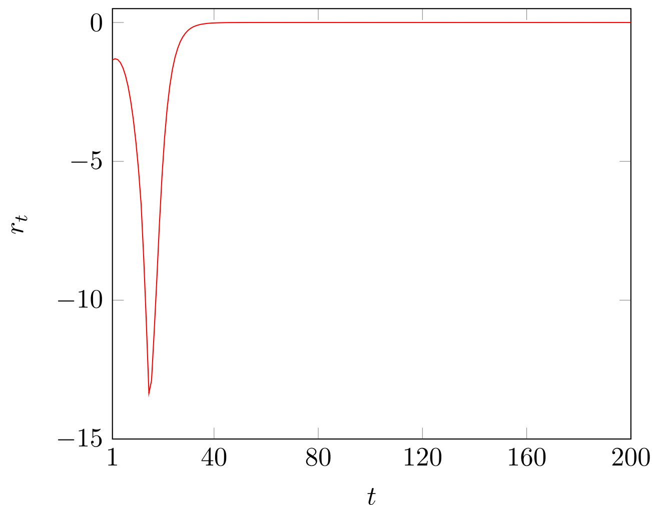


Figure 5: Rewards over time for an agent using the Cross-Entropy Method to solve the Pendulum environment. Actions are planned using simulations in the true environment model. Results are averaged over 10 episodes. In each of the episodes, the CEM requires one swing (causing the initial decrease of rewards) from the initial state to balancing the pendulum while obtaining rewards around 0.

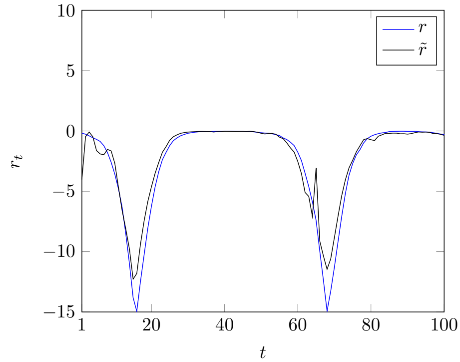
[6]Model weights and datasets have been periodically saved, and can be obtained by contacting the author.

5.2 Experiment 2 - Environment Model and Data Augmentation

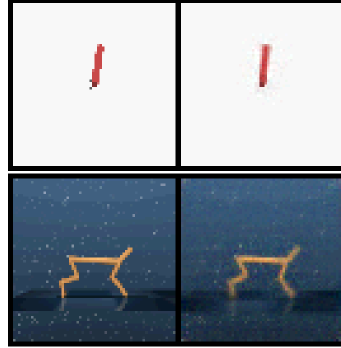
The RSSM has been trained on a simple but commonly used OpenAI benchmark for reinforcement learning, where the goal is to balance a pendulum. The PlaNet agent selects actions by planning in the learned environment model, that is trained to mimic the pendulum. The environment reward function is based on the angle the pendulum makes with the upright goal state, meaning there is an invariance towards mirroring the pendulum over a vertical axis. Trajectories sampled from the true environment model could therefore be augmented/mirrored to create artificial trajectories with equal reward expectation. Multiple RSSMs were trained both with and without the use of these artificial trajectories, to observe their effects. A random translation procedure has been tried as well, given its success in related work.

5.2.1 Model Predictions

To get a general impression of the RSSM prediction quality some examples are shown here, before presenting the evaluation results in Section 5.2.2. The trained environment model is able to accurately predict the rewards and observations of the true pendulum environment. Figure 6a shows the predicted reward signal $\tilde{r}$ compared to true reward r for some evaluation episode. Figure 6b shows the RSSM is able to reconstruct the observations based on its latent state $\tilde{s}_t$. This ability was verified to be still present when the agent is confronted with more complex observations (such as the Cheetah environment). It is important to note that the encoder/decoder structure of the RSSM forces a reconstruction from $\tilde{s}_t$.



(a) True rewards r and predicted rewards $\tilde{r}$ during one episode when evaluating the RSSM trained using data augmentation.



(b) True observation (left) and predicted observation (right) of the RSSM in both the OpenAI pendulum environment and the Cheetah environment of the Control Suite benchmark.

Figure 6: Examples of observation and reward predictions given by the RSSM.

5.2.2 Evaluation Results

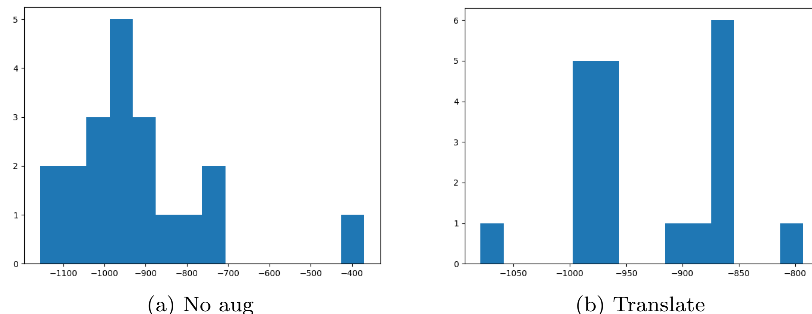
Table 1 shows the cumulative reward obtained during 20 evaluation runs for each of the trained models. The model trained without data augmentation is labeled ”no aug”. ”translate” and ”mirror” refer to the random translate and mirror procedure described in Section 4.2, respectively. All three models were trained on the same number of (possibly augmented) sample episodes. Data augmentation does, however, increase the number of data samples that can be used for training. Since the mirror augmentation procedure effectively doubles the available data, another model was trained using all samples and was labeled ”mirror(2x)”.

	$G_0^{(1)}$	$G_0^{(2)}$	$G_0^{(3)}$	$G_0^{(4)}$	$G_0^{(5)}$	$G_0^{(6)}$	$G_0^{(7)}$	$G_0^{(8)}$
no aug	-923.5	-1008.9	-1008.8	-1042.6	-885.9	-870.8	-1058.1	-1155.6
translate	-860.0	-976.1	-981.4	-856.7	-792.9	-990.5	-965.6	-855.5
mirror	-994.6	-1176.3	-853.4	-1055.8	-987.8	-780.1	-917.9	-1192.2
mirror(2x)	-1073.2	-1088.5	-1186.2	-1295.8	-1084.4	-1186.2	-1191.3	-1213.6
	$G_0^{(9)}$	$G_0^{(10)}$	$G_0^{(11)}$	$G_0^{(12)}$	$G_0^{(13)}$	$G_0^{(14)}$	$G_0^{(15)}$	$G_0^{(16)}$
no aug	-952.6	-978.5	-978.7	-735.4	-956.3	-778.0	-369.8	-1152.2
translate	-860.7	-975.8	-885.5	-899.9	-980.5	-995.4	-960.6	-1079.1
mirror	-987.1	-1090.7	-1073.0	-907.1	-963.8	-1196.5	-969.4	-968.3
mirror(2x)	-1260.2	-1179.1	-1082.0	-1185.2	-1259.6	-1082.4	-1301.4	-1022.9
	$G_0^{(17)}$	$G_0^{(18)}$	$G_0^{(19)}$	$G_0^{(20)}$	mean	stdev		
no aug	-976.4	-902.7	-748.6	-1092.7	-928.8	176.2		
translate	-973.9	-858.1	-869.9	-980.6	-929.9	71.5		
mirror	-1042.4	-979.9	-1069.3	-991.0	-1020.2	92.5		
mirror(2x)	-1169.1	-1183.4	-1307.9	-1190.2	-1177.1	83.1		

Table 1: Sum of obtained rewards (i.e. return G_t) starting from initial state s_0 for multiple learned environment models, each trained with different data augmentation procedures (no aug, translate, mirror). Results are shown for 20 evaluation episodes for each model.

Given these results, we would like to draw conclusions which model performed better. In related work on Reinforcement Learning, these are typically drawn based on evaluation runs of various benchmarks and there is no consensus about how many runs are required for these conclusions to be meaningful. Colas et al. [61] empirically investigate statistical tests for their applicability and statistical power (i.e. true positive rate of rejecting the null hypothesis) in RL and recommend the Welch t-test based on their findings. A sample size of 20 was required to reach a statistical power of 0.8 with effect size[7] of 1. It should be noted however, that the Welch t-test does assume that the return G_t of both algorithms is normally distributed, but does not assume equal variance. This assumption generally *cannot* be made in the RL setting, but Colas et al. observed robustness against the violation of this assumption in a similar setting (see Figure 7). Evaluation of this claim is outside the scope of this research, which is why we present an interpretation based on their findings.

[7] where effect size $\epsilon = \frac{|\Delta\mu|}{\sqrt{(\sigma_1^2+\sigma_2^2)/2}}$



(a) No aug (b) Translate

Figure 7: Histograms of the 20 return values obtained for the models trained without data augmentation and with the "translate" data augmentation (as reported in Table 1). A visual inspection of the returns does not indicate normality of the distributions, as assumed in the Welch t-test. Figure (b) suggests a multi-modal distribution which likely corresponds to the number of times the agent drops the pendulum. Colas et al. did, however, also evaluate the Welch t-test with multi-modal (mixture of normal) distributions.

5.2.3 Statistics and Interpretation

The trained environment models were compared using the Welch t-test and the results are presented in this section. In each of the comparisons our null hypothesis states that the compared algorithms perform equally (i.e. give equal expected return in the evaluation runs). Our alternative hypothesis states there is a difference in performance, which could be better or worse, which is why a two-sided test is used. Based on the (N=20) samples in Table 1,

- Comparing "no aug" and "translate" gives a t-statistic of 0.026 and p-value (i.e. probability of observing the data given our null hypothesis) of 0.979, meaning the test confidently states there is no difference in the expected return obtained from both algorithms. When looking at Table 1, some interesting observations can be made. Comparing the first 9 samples does indicate a preference towards the model trained with the "translate" data augmentation. In fact, had we only considered these samples, the Welch t-test would give a p-value of 0.045, and would result in rejecting the null hypothesis (with the commonly used threshold value of 0.05[8]). This clearly demonstrates the importance of a large sample size when comparing these algorithms, but also suggests larger sample sizes might be required than reported by Colas et al. [61]. Evaluation run 15 of the "no aug" model is a strong influence in the contrasting conclusions. It is the only run (of 80 runs) in which the agent balances the pendulum upright throughout the entire episode, resulting in a high cumulative reward.

[8] Multiple hypotheses are tested using the same data and thus the α-value should be adapted. This is reflected on after presenting the test results.

- Comparing the "no aug" and "mirror" models gives t= 2.053 and a p-value of 0.049, meaning the null hypothesis could be rejected and the alternative hypothesis is favored. It should be noted, however, that this alternative hypothesis states that the two environment models do not perform equally w.r.t. expected return. With a one sided test (with alternative hypothesis that "mirror" is worse than "no aug") the rejection threshold would have to be adjusted, meaning the null hypothesis would not be rejected. The observed means do indicate that the "mirror" model performs worse than the "no aug" model.

- Comparing the "no aug" and "mirror(2x)" models gives t= 5.698 and a p-value of 0.000005, meaning for both the one-sided and two-sided test the null hypothesis can be rejected. The "mirror(2x)" model performs worse than the model trained without data augmentation.

An alternative test that does not assume normality of the distributions is a permutation test. This test was also included in the comparison made by Colas et al, but their results favored the Welch t-test. However, to validate the results a permutation test was performed as well, giving practically equivalent p-values of $0.979, 0.041, 0.000005$ for the three comparisons, respectively.
Each of the models trained using data augmentation does show a decreased variance compared to the non-augmented model. The Levene's test for equal variances with non-normal distributions obtained the following:

- The return values obtained for the "no aug" and "translate" models result in a w-statistic of 3.071 with p-value 0.088.

- Comparing the "no aug" and "mirror" models give a w-statistic of 2.045 with p-value 0.161.

- Lastly, "no aug" and "mirror(2x)" give a w-statistic of 2.782 with p-value 0.104.

Despite having a reduction in variance for each of the data augmentation procedures, none of the p-values pass the 0.05 threshold commonly used for rejecting the hypothesis that the models have equal variance. With more evaluation runs this would likely be possible, since data augmentation is already established as a variance reduction technique [55].
Furthermore, since multiple hypotheses are tested on the same data, the probability of obtaining a false positive result increases and the α threshold should be changed to compensate for this. Using Bonferroni correction, the threshold is divided by the number of hypotheses that are tested, which is 9. This gives $\alpha = 0.05/9 \approx 0.0055$, meaning that only the hypothesis that "no aug" and "mirror(2x)" perform equally well can be rejected.
Given these results, it should be noted that there are some factors that could not be accounted for in the experiments due to constraints in the computation available. Each of the trained models was evaluated using 20 episodes. Variability in the weight initializations, data collection or optimization procedure

affects the quality of the model. Ideally, the model would be evaluated using newly trained models for each run.
One explanation for the difference in results between various data augmentation schemes could therefore be the difference in model initializations. Experiment 3 was designed to investigate this claim, where reducing the model size allows more models to be trained. The worse performance during extended training with data augmentation could also be caused by overfitting on the limited data in replay memory. This could be verified by training models without data augmentation for an extended number of iterations as well, and see if they show similar worse performance. However, the fact that the models trained with an equal number of data points showed worse performance indicates that the data augmentation procedure is the culprit. It should also be noted that the "mirror(2x)" models are "mirror" models that are trained for an extended time period. That is, they are the same instance with the same (possibly bad) weight initialization.
Laskin et al. [2] compared different data augmentation procedures and show the "translate" procedure to be most effective. In our experiments, it did not produce a beneficial effect. This could be due to the augmentation being less effective on this particular model or problem that is solved. Laskin et al. compare the different augmentations on one environment using one algorithm and it is not clear why the random translate augmentation *should* be effective. The pendulum environment is simple compared to many RL benchmarks and it could be that augmenting existing data samples does not provide extra information about the solution space. Data augmentation removes the independence between samples in the dataset, changing the data distribution. The goal of this dependence is to make it more apparent to the model during training, if the relation is otherwise not well represented in the dataset. It could be, however, that in the pendulum problem the data collected under a uniformly random policy was already informative enough for data augmentation to have any effect. As noted before, the first 9 evaluation runs of the model trained with the "translate" procedure did indicate a beneficial effect, which disappeared when extending the number of evaluation runs. This was, however, mainly caused by one "outlier", in which the model trained without data augmentation managed to balanced the pendulum perfectly throughout the entire episode. This further illustrates the variance between evaluation runs and their effects on the conclusions that are drawn. It also raises the question whether this outlier was coincidental or does reflect the performance of the environment model. It also indicates that, for these questions to be answered, a sample size of 20 runs is not enough. All models trained with data augmentation did show a reduction in the variance between evaluation runs.

5.3 Experiment 3 - Test for Invariance

To test how well the learned environment models respect the invariance property of some augmentation procedure, the difference in predictions ($\Delta \tilde{r}_t$) for trajectories with equal reward expectation was considered. Six models were

trained without using data augmentation on the pendulum environment. Another six models were trained using the "mirror" augmentation procedure, in which some trajectories were mirrored when sampled from the dataset. The question is whether explicitly using the invariance property to generate data samples is reflected in the learned environment model. Each of the trained models is evaluated for five episodes, for a total of 30 evaluation runs for each model type.
The mean difference in rewards for all model predictions were 0.10324 and 0.10319 for the models trained without and with data augmentation, respectively. They are practically equal, which is reflected by the following tests that were performed:

- Welch t-test: Gives $t = 0.0316$ with p-value 0.975 when testing whether there is a difference between $\Delta\tilde{r}$ for both models.
- Permutation testing gave $p = 0.975$, meaning both tests confidently show the null hypothesis should not be rejected and both models respect the mirroring invariance equally well, despite the use of data augmentation.

To evaluate the effects of initializations on the model, each of the 6×2 downscaled models were evaluated for three episodes using the CEM. The obtained returns are presented in table 2.

	$G_0^{(1)}$	$G_0^{(2)}$	$G_0^{(3)}$	$G_0^{(4)}$	$G_0^{(5)}$	$G_0^{(6)}$
no aug	-1336.1	-1369.8	-1006.6	-885.7	-1110.3	-1112.9
mirror	-1113.1	-1113.4	-1573.6	-1640.9	-1140.9	-1197.3
	$G_0^{(7)}$	$G_0^{(8)}$	$G_0^{(9)}$	$G_0^{(10)}$	$G_0^{(11)}$	$G_0^{(12)}$
no aug	-1368.4	-1451	-1042.7	-1027.1	-1472.3	-1406
mirror	-1100.7	-1105.2	-1103	-1120.6	-1102.7	-1104.3
	$G_0^{(13)}$	$G_0^{(14)}$	$G_0^{(15)}$	$G_0^{(16)}$	$G_0^{(17)}$	$G_0^{(18)}$
no aug	-1349.7	-1004.5	-1099.9	-1349.1	-1082.1	-1416
mirror	-1147.9	-1627.3	-1188.1	-1131.6	-1099.6	-1158.4

Table 2: Return values G_0 obtained when evaluating the downscaled RSSM for 18 runs.

Again, there is a high variance between evaluation runs. The sample means are -1216.2 and -1209.4 for "no aug" and "mirror", respectively. Testing for the inequality of the expected return gives

- Welch t-test: $t = -0.107$ with p-value of 0.915
- Permutation test: p-value of 0.915

These values contrast the results of experiment 2, since they give an indication that there is no difference in expected return. This also suggests that the difference observed in experiment 2 is not due to the effect of data augmentation.

5.4 Experiment 4 - Value Function

The experiment showed that training a value function in conjunction with the RSSM in the way presented in this work proved to be unstable. Value function predictions quickly diverged to become unrealistic and not applicable for planning. The PlaNet algorithm thus is not compatible with using a learned value function in the way that is presented in this work, regardless of the use of data augmentation.

6 Conclusion

The primary focus of this work has been to investigate the effects of data augmentation for regularizing learned environment models in reinforcement learning. More specifically,
RQ 1: How does regularizing the Recurrent State Space Model used in PlaNet through data augmentation affect its performance in terms of data efficiency and asymptotic performance?
Recurrent State Space Models were trained with and without data augmentation on a simple and commonly used RL benchmark, in which an inverted pendulum is balanced. From our experiments, no significant difference was found between the expected sum of rewards obtained when planning with models trained with or without data augmentation. This applied to all augmentation procedures that were evaluated and stands in contrast with the results of related work [3, 2]. Interestingly, the first 9 runs gave an indication that data augmentation performed better, and the Welch t-test confirmed this. However, with a larger sample size (of 20, as recommended by Colas et al. [61],) these indications were shown to be false. This illustrated the importance of a large sample size when comparing RL algorithms, since the variance between different runs can be large. Care should be taken to perform enough runs to draw conclusions with enough statistical power, in order not to overfit our findings on coincidental results. When looking at the differences between predictions for reward-invariant state-action pairs, there was no difference between the models trained with or without data augmentation. From this we conclude they both respect the invariance property that data augmentation aims to enforce equally well. Each of the models trained with data augmentation did show a reduced variance in their evaluation runs. It is surprising that this did not result in an improved performance of the CEM when planning actions, since high variance (w.r.t. properties that are invariant w.r.t. the reward function) in simulated trajectories is a general issue of Monte-Carlo type algorithms. This raises the question if the regularization of the environment model resulted in variance reduction w.r.t. properties that are relevant to the planning procedure and that test-time data augmentation might be more effective.

Concurrently with Laskin et al, Kostrikov et al. successfully applied data augmentation in reinforcement learning by regularization of a learned action-value function. The authors of PlaNet hypothesized that using a value function during planning could improve its performance. In this work, we investigate this claim as a secondary research focus, namely
RQ 2: How does the regularization procedure implemented by Kostrikov et al. affect the PlaNet algorithm when it is augmented with a value function?
Adding a value function to the RSSM training objective to be used during planning was found to be unstable in the approach taken in this work. The PlaNet algorithm therefore cannot be regularized in the same manner implemented by Kostrikov et al. A more successful approach was taken by Hafner et al. when applying an actor-critic agent on the RSSM [21]. Their agent, called Dreamer, is

better able to deal with long-horizon challenges in RL benchmarks. Therefore, applying data augmentation on Dreamer might prove more effective.

Further recommendations for future work include the use of various data augmentation procedures on different RL benchmarks, since it is not clear which augmentations are effective for a given problem. This could also give insight to what is required to develop algorithms that have better generalization performance. Related work already explores the possibility of automatically searching augmentations that are useful when solving a problem by modeling it as a multi-armed bandit problem [62]. This does still require the manual selection of a number of data augmentations. To automate the process of finding useful augmentations an approach based on GANs might prove useful, such as used by Zhu et al. [57]. The idea of learning world models to generate artificial episodes of data could also be seen as a form of data augmentation.
Furthermore, there is no consensus on how to properly evaluate RL algorithms. In this work we observed that with 20 runs for each model, which is extensive by RL standards, the effects of single runs can be quite influential in the outcome of statistical tests. To not be mislead by coincidental results, we argue that work on how to compare RL algorithms with sufficient statistical power would be valuable.

References

[1] Balaraman Ravindran and Andrew G Barto. Model minimization in hierarchical reinforcement learning. In *International Symposium on Abstraction, Reformulation, and Approximation*, pages 196–211. Springer, 2002.

[2] Michael Laskin, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990*, 2020.

[3] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[6] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Tingwu Wang and Jimmy Ba. Exploring model-based planning with policy networks. *arXiv preprint arXiv:1906.08649*, 2019.

[8] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[9] Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C Courville, and Yoshua Bengio. A recurrent latent variable model for sequential data. In *Advances in neural information processing systems*, pages 2980–2988, 2015.

[10] Maximilian Karl, Maximilian Soelch, Justin Bayer, and Patrick Van der Smagt. Deep variational bayes filters: Unsupervised learning of state space models from raw data. *arXiv preprint arXiv:1605.06432*, 2016.

[11] Andreas Doerr, Christian Daniel, Martin Schiegg, Duy Nguyen-Tuong, Stefan Schaal, Marc Toussaint, and Sebastian Trimpe. Probabilistic recurrent state-space models. *arXiv preprint arXiv:1801.10395*, 2018.

[12] Lars Buesing, Theophane Weber, Sébastien Racaniere, SM Eslami, Danilo Rezende, David P Reichert, Fabio Viola, Frederic Besse, Karol Gregor, Demis Hassabis, et al. Learning and querying fast generative models for reinforcement learning. *arXiv preprint arXiv:1802.03006*, 2018.

[13] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1282–1289, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[14] Amy Zhang, Nicolas Ballas, and Joelle Pineau. A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*, 2018.

[15] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

[16] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

[17] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[18] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[19] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

[20] Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

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

[22] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. *arXiv preprint arXiv:2005.05960*, 2020.

[23] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[24] Manuel Watter, Jost Springenberg, Joschka Boedecker, and Martin Riedmiller. Embed to control: A locally linear latent dynamics model for control from raw images. In *Advances in neural information processing systems*, pages 2746–2754, 2015.

[25] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[27] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[28] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[29] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[30] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[31] Christopher G Atkeson and Juan Carlos Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of international conference on robotics and automation*, volume 4, pages 3557–3564. IEEE, 1997.

[32] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

[33] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. *A survey on policy search for robotics*. now publishers, 2013.

[34] Evan Archer, Il Memming Park, Lars Buesing, John Cunningham, and Liam Paninski. Black box variational inference for state space models. *arXiv preprint arXiv:1511.07367*, 2015.

[35] Tuomas Haarnoja, Anurag Ajay, Sergey Levine, and Pieter Abbeel. Backprop kf: Learning discriminative deterministic state estimators. In *Advances in Neural Information Processing Systems*, pages 4376–4384, 2016.

[36] Rahul G Krishnan, Uri Shalit, and David Sontag. Structured inference networks for nonlinear state space models. In *Thirty-first aaai conference on artificial intelligence*, 2017.

[37] Marco Fraccaro, Simon Kamronn, Ulrich Paquet, and Ole Winther. A disentangled recognition and nonlinear dynamics model for unsupervised learning. *Advances in Neural Information Processing Systems 30, NIPS*, 2017.

[38] Philipp Becker, Harit Pandya, Gregor Gebhardt, Cheng Zhao, James Taylor, and Gerhard Neumann. Recurrent kalman networks: Factorized inference in high-dimensional deep feature spaces. *arXiv preprint arXiv:1905.07357*, 2019.

[39] Joschka Boedecker, Jost Tobias Springenberg, Jan Wülfing, and Martin Riedmiller. Approximate real-time optimal control based on sparse gaussian process models. In *2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 1–8. IEEE, 2014.

[40] Jonathan Ko and Dieter Fox. Gp-bayesfilters: Bayesian filtering using gaussian process prediction and observation models. *Autonomous Robots*, 27(1):75–90, 2009.

[41] Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Peter Deisenroth. Manifold gaussian processes for regression. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 3338–3345. IEEE, 2016.

[42] Marvin Zhang, Sharad Vikram, Laura Smith, Pieter Abbeel, Matthew J Johnson, and Sergey Levine. Solar: deep structured representations for model-based reinforcement learning. *arXiv preprint arXiv:1808.09105*, 2018.

[43] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[44] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I Jordan, Joseph E Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv preprint arXiv:1803.00101*, 2018.

[45] Balaraman Ravindran and Andrew G Barto. Approximate homomorphisms: A framework for non-exact minimization in markov decision processes. 2004.

[46] Shravan Narayanamurthy and Balaraman Ravindran. On the hardness of finding symmetries in markov decision processes. pages 688–695, 01 2008.

[47] Jonathan Taylor, Doina Precup, and Prakash Panagaden. Bounding performance loss in approximate mdp homomorphisms. In *Advances in Neural Information Processing Systems*, pages 1649–1656, 2009.

[48] Ondrej Biza and Robert Platt. Online abstraction with mdp homomorphisms for deep learning. *arXiv preprint arXiv:1811.12929*, 2018.

[49] Elise van der Pol, Thomas Kipf, Frans A Oliehoek, and Max Welling. Plannable approximations to mdp homomorphisms: Equivariance under actions. *arXiv preprint arXiv:2002.11963*, 2020.

[50] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE, 2012.

[51] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[52] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[53] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[54] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[55] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[56] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[57] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

[58] Aravind Srinivas, Michael Laskin, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

[59] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[60] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[61] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms. *arXiv preprint arXiv:1904.06979*, 2019.

[62] Roberta Raileanu, Max Goldstein, Denis Yarats, Ilya Kostrikov, and Rob Fergus. Automatic data augmentation for generalization in deep reinforcement learning. *arXiv preprint arXiv:2006.12862*, 2020.

[63] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

7 Appendix - Notation Overview

Symbol	Description	Symbol	Description
t	Time step	ϕ	State transformation
T	Trajectory length	ψ	State-action transformation
s, s_t	Environment state (at some time t)	ν	Parameterization of some transformation
a, a_t	Action (at some time t)	$\mathcal{T}$	Set of possible parameterizations ν
r, r_t	Reward (at some time t)	M	Number of augmented samples used
o, o_t	Observation (at some time t)		
$o_{t:t+\tau}$	Observations obtained from step t to step $t+\tau$		
S_t	Random variable of the environment stat		
A_t	Random variable of the chosen action		
R_t	Random variable of the reward		
O_t	Random variable of the observation		
$\mathcal{S}$	State space		
$\mathcal{A}$	Action space		
Ψ	Permittable state-action pairs		
$\mathcal{P}(s', r\|s, a)$	Transition function		
$\mathcal{O}(o\|s, a)$	Observation function		
$\mathcal{M}$	(PO)MDP		
G_t	Return/ sum of reward		
$\pi(a\|s)$	Policy		
$v_\pi(s)$	Value function		
$q_\pi(s, a)$	Action-value function		
$bel(s)$	State belief distribution given the observations		
$\overline{bel}(s)$	State belief distribution prior to most recent observation		
D	Planning horizon		
I	Nr. of policy updates		
J	Nr. of candidate trajectories		
K	Nr. of top candidates		

Table 3: Table containing the notations used in this document.

8 Appendix - Implementation

The implementation of the experiments that were run can be found here[9]. All experiments were executed by running `main.py` under different hyperparameters.

[9]https://github.com/ronvree/DataAugmentationInMBRL

9 Appendix - Additional Hyperparameters & Implementation Details

All hyperparameters for each run are automatically logged by the implementation and can therefore also be found in the experiment log.

9.1 CEM

Description	Parameter	Value	Original Value
Planning horizon	D	12	12
Nr. of optimization iterations	I	10	10
Nr. of candidates	J	100	1000
Nr. of top candidates	K	10	100

Table 4: Cross-Entropy Method hyperparameters. The "Original Value" column refers to the values used in the paper that introduced the RSSM. The number of candidate action sequences that are evaluated is reduced because of the limited computing power available.

9.2 Recurrent State Space Model

All hyperparameters in the RSSM are the same as in the original paper.

9.2.1 State model

The RSSM's belief state consists of two components (i.e. $\tilde{s}_t = (h_t, z_t)$). $h_t \in \mathbb{R}^{200}$ is a hidden state of recurrent neural network and forms a deterministic component of the belief state. Based on h_t, a Gaussian distribution with mean and variance parameterized by a dense neural network is sampled for a stochastic belief state component $z_t \in \mathbb{R}^{30}$. The RNN is implemented as a GRU [63] with an initial state h_0 of all zeroes.

9.2.2 Reward model

The reward model consists of a Gaussian distribution with mean parameterized by a neural network and a variance of 1. The neural network is a simple dense network with two hidden layers of size 200. Given a belief state $\tilde{s}$ it outputs a predicted reward $r \in \mathbb{R}$.

9.2.3 Observation model

The observation model $\tilde{\mathcal{O}}$ acts as a decoder and generates observations given the current belief state $\tilde{s}_t = (z_t, h_t)$. First, a dense layer concatenates the (h_t, z_t) vectors to create an encoding in $\mathbb{R}^{1024}$. A 4-layer deconvolutional network transforms this encoding to a 64x64 RGB image. This parameterizes the mean of a Gaussian with identity covariance.

9.2.4 Encoder

The encoder is modeled as a diagonal Gaussian with mean and covariance parameterized by a convolutional neural network. The network consists of four convolutional layers that transform the input image to 256 4×4 feature maps, which form a 1024-dimensional encoding of the observation. This is concatenated with the state belief prior and passed through a dense net to obtain the belief posterior.

9.2.5 Training procedure

The RSSM is trained using batches of episode data sampled from the experience replay. The loss is computed over the complete episode. That is, the chunk length parameter is set to T, the total episode length.

Description	Parameter	Value	Original Value
Chunk length	L	200	50
Batch size	B	32	32
Number of batches	C	50	200
Learning rate	α	10^{-3}	10^{-3}
Adam epsilon	ϵ	10^{-4}	10^{-4}

Table 5: RSSM training procedure hyperparameters.

9.3 RSSM (Downscaled)

To reduce the time required to train the RSSM model a downscaled version has been used as well. The differences are highlighted here.
The latent state of the RSSM has its dimensions reduced. The GRU hidden state (h_t) is 32-dimensional, rather than 200. The stochastic state component z_t consists of 16 values, rather than 30. The observation encodings are reduced to $\mathbb{R}^{256}$. Furthermore, all dense layers in the state, observation and reward models contain 32 neurons rather than 200.

9.4 Action-Value Model

The action-value function model was implemented as a dense neural network with two hidden layers of size 200 and relu activation. Is input was a vector in which the RSSM belief state $\tilde{s}_t$ and action a_t were concatenated.

10 Appendix - Pseudocode

10.1 Cross-Entropy Method

Algorithm 1: Planner algorithm using the Cross-Entropy Method (see Section 2.8.1), with the adaptation in blue

input :
- D Planning horizon distance
- I Optimization iterations
- J Candidates per iteration
- K Number of top candidates to fit
- $\tilde{\mathcal{M}}$ Environment model

```
1  Initialize factorized belief over action sequences f(a_{t:t+D}) ← N(0, I)
   // Perform optimization iterations as defined in Eq. 22
2  for optimization iteration i=1..I do
       // Evaluate J action sequences from the current policy
3      for candidate action sequence j=1..J do
           // Sample the policy
4          a^{(j)}_{t:t+D} ~ f(a_{t:t+D})
5          Obtain o^{(j)}_{t+1:t+D}, r^{(j)}_{t+1:t+D} by executing a^{(j)}_{t:t+D-1} in the
            environment model M̃
           // Estimate the return G from the trajectory
           // Following Monte Carlo estimation or Eq. 9
6          G^{(j)} = Σ_{τ=t+1}^{t+D} r^{(j)}_τ + q̂_π(s^{(j)}_{t+D}, a^{(j)}_{t+D}; θ)
7      end
       // Re-fit the policy to the K best sequences (Eq.23)
8      K ← argsort({G^{(j)}}_{j=1}^{J})
9      μ_{t:t+D} = 1/K Σ_{k∈K} a^{(k)}_{t:t+D}
10     σ_{t:t+D} = 1/(K−1) Σ_{k∈K} |a^{(k)}_{t:t+D} − μ_{t:t+D}|
11     f(a_{t:t+D}) ← N(μ_{t:t+D}, σ²_{t:t+D} I)
12 end
13 return first action mean μ_t
```

Line 1: $f(a_{t:t+D}) \leftarrow \mathcal{N}(0, \mathbb{I})$

Line 4: $a^{(j)}_{t:t+D} \sim f(a_{t:t+D})$

Line 5: Obtain $o^{(j)}_{t+1:t+D}, r^{(j)}_{t+1:t+D}$ by executing $a^{(j)}_{t:t+D-1}$ in the environment model $\tilde{\mathcal{M}}$

Line 6: $G^{(j)} = \sum_{\tau=t+1}^{t+D} r^{(j)}_\tau + \hat{q}_\pi(s^{(j)}_{t+D}, a^{(j)}_{t+D}; \theta)$

Line 8: $\mathcal{K} \leftarrow \text{argsort}\left(\left\{G^{(j)}\right\}_{j=1}^{J}\right)$

Line 9: $\mu_{t:t+D} = \frac{1}{K} \sum_{k \in \mathcal{K}} a^{(k)}_{t:t+D}$

Line 10: $\sigma_{t:t+D} = \frac{1}{K-1} \sum_{k \in \mathcal{K}} \left| a^{(k)}_{t:t+D} - \mu_{t:t+D} \right|$

Line 11: $f(a_{t:t+D}) \leftarrow \mathcal{N}(\mu_{t:t+D}, \sigma^2_{t:t+D}\mathbb{I})$

Line 13: **return** *first action mean* μ_t

10.2 PlaNet

Algorithm 2: Main loop

input :

B	Batch size	$\mathcal{M}$	True environment model
S	Seed episodes	$\tilde{\mathcal{M}}$	RSSM
C	Nr. of samples	$p(\epsilon)$	Exploration noise
L	Chunk length		
α	Learning rate		

1 Initialize dataset $\mathcal{D}$ with S random seed episodes
2 Initialize model parameters θ randomly
3 **while** not converged **do**
  `// Model fitting using Minibatch Gradient Descent`
4   **for** *update step* $c = 1..C$ **do**
5   Draw sequence chunks $\{(o_t, a_t, r_t)_{t=k}^{L+k}\}_{i=1}^{B} \sim \mathcal{D}$ uniformly at random from the dataset
6   Compute loss $\mathcal{L}(\theta)$ as described in section **??**
7   Update model parameters $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$
8   **end**
  `// Data collection`
9   $o_0 \leftarrow$ env.reset()
10   $\overline{bel}(\tilde{s}_0) \leftarrow$ RSSM.reset()
11   **for** *time step* $t = 0..T$ **do**
12   Use the obtained observation o_t and encoder q to update the RSSM state belief distribution from $\overline{bel}(\tilde{s}_t)$ to $bel(\tilde{s}_t)$ (Eq. 14)
    `// Select action using the CEM, based on the current state belief`
13     $a_t \leftarrow$ planner($bel(\tilde{s}_t)$), See Algorithm 1
14   Add exploration noise $\epsilon \sim p(\epsilon)$ to the action
    `// Execute the action in the environment model`
15     $r_{t+1}, o_{t+1} \leftarrow$ env.step(a_t)
    `// Execute the action in the learned environment model`
16     $\overline{bel}(\tilde{s}_{t+1}) \leftarrow$ RSSM.step(a_t)
17   **end**
18   $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_{t+1})_{t=1}^{T}\}$
19 **end**