DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Vision-Based Continual Reinforcement Learning for Robotic Manipulation Tasks

Marcel Bruckner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Vision-Based Continual Reinforcement Learning for Robotic Manipulation Tasks

Visionsbasiertes Continual Reinforcement Learning für Robotermanipulationsaufgaben

Author:	Marcel Bruckner
Supervisor:	Alois Knoll, Prof. Dr.-Ing. habil.
Advisor:	Josip Josifovski, M.Sc.
	Mohammadhossein Malmir, M.Sc
	Zhenshan Bing, Dr. rer.nat
Submission Date:	15.02.2022

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.02.2022 Marcel Bruckner

Acknowledgments

Throughout my writing of this thesis I received a great deal of support and assistance. I would like to thank my advisor, Josip Josifovski, for providing me the opportunity to work together on our topic. Your extensive support and valuable ideas on all steps of my thesis journey helped me to grow beyond my limits.
In addition, I would like to thank my parents and friends for their unconditional support. You are always there for me.
Finally, I could not have completed this thesis without the support of my close friend, Julian Hohenadel, who provided stimulating discussions as well as happy distractions to rest my mind outside of my studies.

Abstract

Resent research shows that learning-based approaches have the potential to replace classical control algorithms for robotic manipulation. However, these approaches are mostly limited to learning a single task using a task-specific state representation. In this thesis we address these limitations by training reinforcement learning agents continually on multiple robotic manipulation tasks, using state representations learned continually from image observations. For this purpose, we propose novel hypernetwork-based approaches for continual state representation learning and for model-free continual reinforcement learning. We evaluate the proposed approaches under different settings for two robotic manipulation tasks in simulation. The results show that the hypernetwork-based approach for state representation learning significantly prevents catastrophic forgetting of the state embeddings of previous tasks. In addition, we find that the hypernetwork-based model-free reinforcement learning agents mitigate catastrophic forgetting completely. Finally, we show that using the state representation learned from images can either outperform or yield comparable performance to the hand-crafted numeric state representation baselines for the different robotic tasks.

Contents

1 Introduction

The human brain learns continually over its lifetime as it approaches new and often distinct tasks. It learns from birth to death without forgetting what it has learned previously, and it transfers old knowledge onto new tasks and improves performance on old ones by new experiences.

Machine Learning (ML) approaches generally lack this human ability to retain old knowledge. ML models assume the data they observe to be independent and identically distributed. As we fit a model to a dataset, we estimate the distribution of the data during training to predict new data points at inference time. When the task changes, due to changes in the dataset or by changes in the training, the estimated distribution of the model changes accordingly [Les+19a]. A neural network (NN) — a specific ML model — is a universal function approximator that estimates data distributions. It consists of layers of neurons that carry a specific weight that is learned by gradient descent during training. As soon as the weights do not change anymore the NN has fit the data and can explain the data distribution.

In Continual Learning (CL) the aim is to train NNs on a sequential stream of tasks, comparably to how humans approach them. The gradient descent method used to train NNs thus sequentially adapts the network weights for the new task. This procedure incrementally overrides the learned weights — the learned knowledge — which leads to **catastrophic forgetting** [Kir+17]. To prevent forgetting old tasks NNs need to exhibit a form of stability to retain the knowledge, analogously to how the human brain stores memories. On the other hand, a CL approach has to exhibit enough plasticity to acquire new knowledge on previously unseen tasks, equally to how humans are able to learn new tasks even at high age. This is known as the **stability-plasticity dilemma** [Kir+17]. Balancing the stability-plasticity dilemma while preventing catastrophic forgetting is the main goal of CL. Additionally, desired abilities of a CL system are to reuse knowledge from previous tasks to improve learning and performance on future tasks, and to incorporate new experiences to improve performance on already learned tasks. This forward and backward transfer of knowledge happens daily in the human brain, and CL aims to enable NNs to exhibit comparable capabilities.

1.1 Vision-Based Continual Reinforcement Learning for Robotic Manipulation Tasks

A reinforcement learning (RL) agent perceives its environment comparably to how humans experience the world. As a RL learning agent interacts with its environment it receives a constant stream of data and tasks. This data stream evolves over time and the environment changes as the agent moves, manipulates, and observes. Continual reinforcement learning (CRL) addresses these continual and non-stationary data streams and aims to maximize the performance of the RL agent when it is trained on a sequence of tasks.

In vision-based RL, the sole input to the agent are images taken from one or multiple cameras. The RL agent processes the images and predicts what to do next, in the same way as humans process and react to what they see. In a setup where a camera is mounted to a robot and the images are from the ego perspective, this is equivalent to the viewing of humans. In practice, only seeing from the ego perspective is often not enough to reliably control a robot. Instead, a common approach is to use a third-person perspective with a camera mounted above the robot, as it is often the case in video games. This vision-based approach removes the need to measure positions and velocities of robots directly as numeric values, and brings the perception of the robot closer to how a human perceives the world.

In humans, we observe a decoupling of the perception and interaction, as our sight receives and processes the visual input and passes the extracted information to other regions of the brain. State representation learning (SRL) is a comparable dimensionality reduction technique for NNs to extract relevant information from high-dimensional images. The main model in SRL are encoder NNs that extract meaningful latent state embeddings from the images. This embedding is especially useful for RL, as the agent can focus on the already extracted relevant information and excel in solving the tasks it is presented. The SRL encoder model learns to extract the information using gradient descent, and is thus naturally subject to catastrophic forgetting.

The growing need of robotic aid in production lines and the emerging of service robots that help on a multitude of household tasks enforces to transfer the vision-based and lifelong learning capabilities of humans onto robots. Vision-based CRL for robotic manipulation tasks is the subfield of RL that is concerned in developing strategies to equip robots with one or multiple manipulation arms with the CL capabilities and the vision-based perception of humans.

1.2 Research Questions

We approach the problem of vision-based continual reinforcement learning for robotic manipulation tasks by answering the following research questions:

1. How well does a RL agent trained on images perform compared to an agent trained on hand-crafted numeric states? Can the vision-based agent solve the given tasks as well as the numeric-based agent?
2. How well does an agent trained on image feature vectors perform compared to an agent trained upon the latent state embedding of a SRL encoder model? Is there a benefit in using the latent state embedding over the image feature vectors?
3. How well can a continual SRL model encode relevant information into the latent state representation when trained on sequential tasks compared to a non-continual SRL module trained on the tasks separately? How well can the continual SRL model avoid catastrophic forgetting?
4. How well does a continual RL agent trained sequentially on either hand-crafted numeric states or latent state embeddings perform compared to a non-continual RL agent trained separately on each task? How well can the continual RL agent avoid catastrophic forgetting?

2 Background

Everywhere in nature, and especially in humans, we observe intelligent behavior that enables living beings to perceive and experience the world. Based on this perception intelligent beings can classify, interact, and envision their surrounding. Since the early 20^{th} century, science fiction authors envisioned that it might be possible to give artificial machines the same intellectual abilities.

In the 1980s, artificial intelligence gained popularity through the buzzword of deep learning. The exponential increase in computing power and the availability of large amounts of data fueled the development of machines that learn from (human) experience [Har]. Since then, breakthroughs in Natural Language Processing, Time Series Forecasting, Computer Vision and Robotic Control base upon the use of deep learning.

Deep Neural Networks Deep Learning (DL) revolves around the idea to train models based on artificial neural networks (NN). NNs are parametric models that approximate functions. The universal approximation theorem [LL20] implies that NN can represent any function when given the appropriate weights (parameters). An NN is thus a function $f_\Theta : \mathbb{R}^d \mapsto \mathbb{R}^D$ that maps from some input space to some output space parametrized by the weights Θ. The weights of the NN can in general not be known in advance, nor be derived directly by human inspection. In supervised learning, the NN learns the parameters through a dataset $\{X, Y\}$ consisting of data points $x \in X$ and corresponding labels $y \in Y$. Starting from an initial untrained configuration of Θ the model estimates labels $\hat{y}$, and by comparison of the estimates $\hat{y}$ with the correct labels y the parameters can be adjusted as displayed in Figure 2.1.

Typically, in DL the shape of the NN model is inspired by the human brain. The models are compiled by multiple layers of neurons that include the parameters Θ. Each of the neurons computes a linear function $W_{n+1} * x_n + b_{n+1} = x_{n+1}$ with $W, b \in \Theta$, and passes the result x_{n+1} through some non-linear activation function σ. By stacking multiple layers of neurons we arrive at the typical architectures of deep NN models displayed in Figure 2.2.

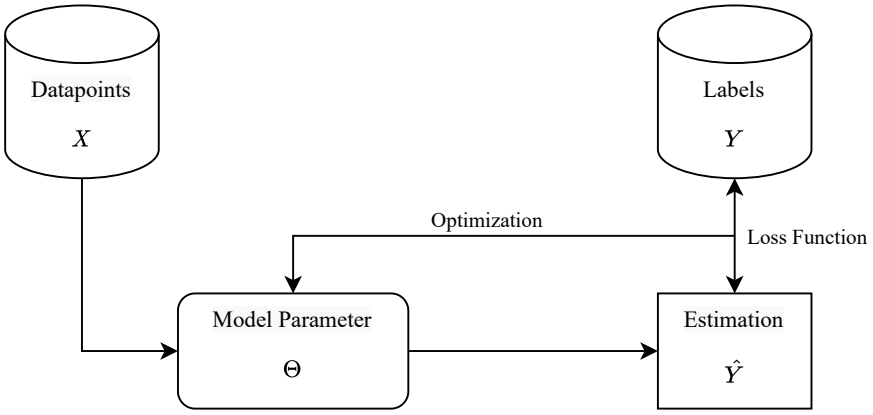


Figure 2.1: The learning scheme of data-driven parametric models. The parameters Θ are estimated based on the dataset $\{X, Y\}$. The loss function compares the quality of the estimate $\hat{Y}$ to the true labels Y. Based on the remaining estimation error the model parameters Θ are optimized.

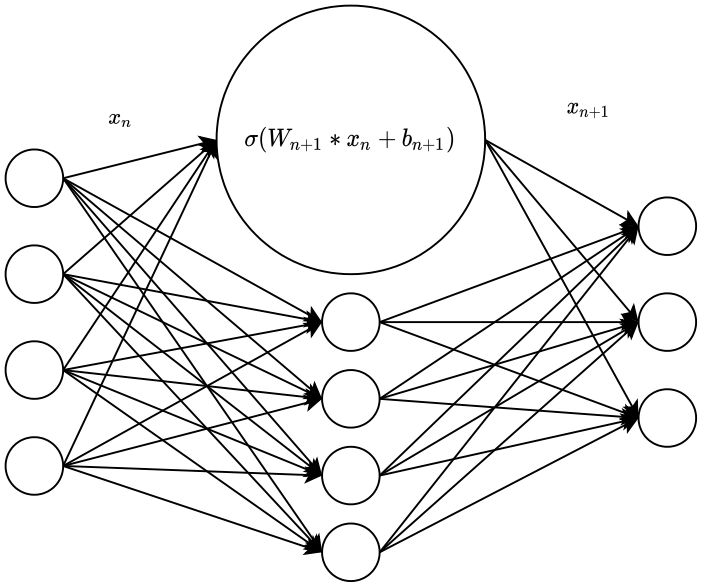


Figure 2.2: A typical single layer fully connected network. Each neuron calculates the function $\sigma(W_{n+1} * x_n + b_{n+1}) = x_{n+1}$ based on the input x_n, the network parameters $W_{n+1}, b_{n+1} \in \Theta$, and the non-linear activation function σ. — From left to right: The input layer, one hidden layer, and the output layer.

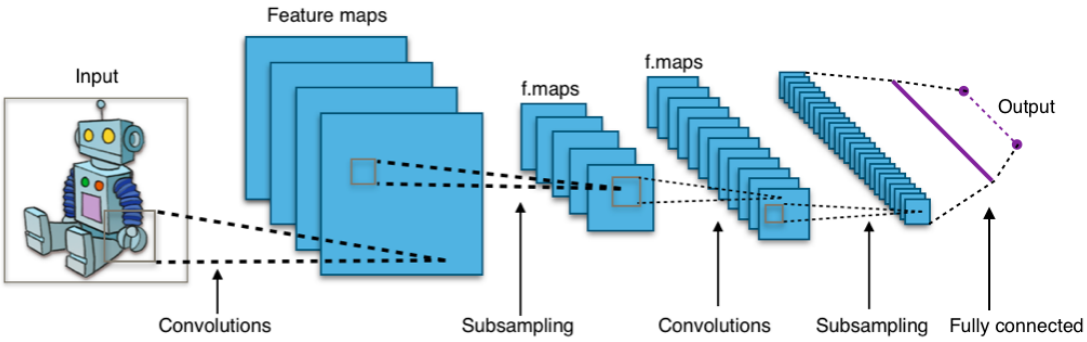


Figure 2.3: The typical architecture of a CNN. Convolutional filters slide across the input image and produce multiple feature maps. By stacking convolutions and subsampling layers the dimensions of the input shrinks and the number of feature maps increases. The last fully connected layer produces the desired output.[1]

[1] Source: `https://de.wikipedia.org/wiki/Convolutional_Neural_Network` [Wik21]

Convolutional Neural Networks When dealing with images, fully connected NNs are often not feasible to use, as for each pixel in the image a neuron in the input layer is needed. The subsequent layers will then also have a large number of neurons and the number of trainable weights quickly increases [Kop00]. Additionally, visual features like edges or corners are often found at multiple positions in the image, and we desire a shared representation of these features.

Convolutional Neural Networks (CNNs) consist of layers of convolutional filters. The filters slide across the image and calculate for each window a convolution based on the filter weights. The weights of the filters are shared across the whole image. This imposes a structural bias on the features present in images, as similar features activate the filters comparably. With increasing depth of the network the convolutional filters build up more complex representations of the features in the image. Figure 2.3 displays the CNN architecture based on convolutional filters.

Deep CNNs are often hard to train, as training based on gradient descent suffers from vanishing or exploding gradients [Hoc91]. One state-of-the-art CNN architecture that mitigates vanishing gradients is the DenseNet family. Densely connected CNNs (DenseNets) [Hua+16] add several parallel skips of layers with their subsequent layers. Deeper layers in the DenseNet receive as input the feature maps of all previous layers. This combines the increasingly more complex filters of deeper layers with the general capabilities of shallow layers. Throughout the experiments we make extensive use of a DenseNet to extract feature vectors from images.

2.1 Reinforcement Learning

In Reinforcement Learning (RL) an agent learns how to solve tasks via sequences of decisions, opposed to supervised learning where an agent learns an input-output mapping from labelled data. The RL agent learns to solve tasks while acting in an environment that is uncertain and potentially complex. The underlying assumption in RL is that the environment behaves according to a Markov Decision Process (MDP) [Bel57].

An MDP is a tuple

$$\text{MDP} := \langle S, A, P, R, \gamma \rangle . \tag{2.1}$$

One of the components of the MDP is the space S of possible states the environment can take on [SB18]. The agent principally cannot observe the full state of the environment, as the state might be to complex or the sensors of the agent do not have enough sensitivity. The agent does not perceive the true state, but observes the state of the environment via its observation function

$$o := observe(s \in S), \tag{2.2}$$

whereas s is the true state of the environment and o is the observed state [SB18].

The RL agent interacts with the environment via a set of actions A. These actions define the possibilities the agent has to influence the environment and to change its own and the environments state. The state transition matrix P defines how the environment evolves as the agent interacts. Especially in RL, the state transition probability matrix

$$P^a_{s,s'} := \mathbb{P}[s_{t+1} = s' | s_t = s, s_t = a] \tag{2.3}$$

defines the transition probabilities from all states $s \in S$ to their successor states $s' \in S$ as the agent takes the action $a \in A$ [SB18].

The reward function R gives feedback to the agent based on how valuable the current state it observes is in relation to the possible actions the agent can execute. The reward function

$$R^a_s := \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.4}$$

is a function of the current state $s \in S$ and actions $a \in A$ the agent can take and estimates the expected reward over all possible combinations [SB18]. The goal of the agent is to maximize the cumulative reward

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

over all future time steps. The discount factor $\gamma \in [0, 1]$ gives diminishing weights to future rewards. It discounts rewards that are distant in the future as the agent cannot predict with certainty how the rewards will evolve. Additionally, as the goal of the agent is to maximize the cumulative reward function G, the decreasing weight of the distant rewards ensures that the cumulative reward is bound from above [SB18].

To maximize the reward, the agent learns a function that determines what action to take based on the current state. The function to select the action is called the policy

$$\pi(a|s) \coloneqq \mathbb{P}(a_t = a|s_t = s) \tag{2.6}$$

and assigns probabilities to the possible actions a the robot may execute when it is in s at some timestep t [SB18].

To learn a policy — and to learn to select an action that is beneficial to take from the current state — the agent learns a value function. The value function

$$v_\pi(s) \coloneqq \mathbb{E}_\pi[G_t|s_t = s] \tag{2.7}$$

represents its desirability to be in a certain state s and describes the expected return G_t starting from the given state [SB18]. The value function is dependent on the current policy π.

The action-value function — also called Q-function — is an extension of the value function to state-action pairs. The Q-function

$$q_\pi(s, a) \coloneqq \mathbb{E}[G_t|s_t = s, a_t = a] \tag{2.8}$$

assigns a value to the expected return of the agent being in state s and taking an action a selected by its policy [SB18].

The benefit of using a value function or the Q-function is that the agent can directly assess the quality of his current state rather than wait for some long-term rewards. It summarizes all future possibilities by estimating the expected returns and allows the agent to assess the quality of different policies.

The advantage function

$$A(s, a) \coloneqq q_\pi(s, a) - v_\pi(s) \tag{2.9}$$

combines the value function and the Q-function and describes the difference between the Q-value of a state-action pair and the value of the state.

The Bellman equation [Bel57] of the value function

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) = \sum_a \pi(a|s) \left(R_s^a + \gamma \sum_{s'} P_{s,s'}^a v_\pi(s') \right) \tag{2.10}$$

separates the value of the current time step from the value of the subsequent time steps when following the policy.

To solve a task and maximize the reward, the agent has to follow an optimal policy π^*. The optimal policy π^* is the policy that performs at least as good as any other policy, whereas the performance of the policy is determined by the optimal value function

$$v^*(s) = \max_\pi v_\pi(s) = \max_a q_\pi(s, a) \tag{2.11}$$

The optimal policy π^* is then the policy that selects actions that maximize the Bellman optimality equation.

2.1.1 Deep Reinforcement Learning

For MDPs with a small and finite number of states and actions an often used approach is tabular Q-learning. In this approach a table of Q-values is stored that can be queried for the optimal action and value per state. Real world problems in RL, like playing the game of Go [Sil+16], autonomous driving of vehicles [Gri+19; Kir+20] or controlling robots [AW21; Gif+17; KCD19; Les+19a] exhibit state and action spaces too large or continuous to be solvable by classical Q-learning.

In Alpha Go [Sil+16], the number of possible sequences of states is approximately $250^{150} = 4.9 * 10^{359}$ and thus greatly exceeds the number of atoms in the universe at $10^{78} \sim 10^{82}$ atoms [Col+15]. Autonomous vehicles deal with continuous state spaces. The vehicle can be anywhere on the globe, outer influences like weather and sunlight influence the perception, and multi-modal sensors with possible extensive data streams extend the number of possible states to infinity [Gri+19; Kir+20]. These unimaginable numbers of possible states makes tabular Q-learning infeasible in practice. The breakthroughs in DL of the last years as described in paragraph 2 thus sparked the question how to combine the universal function approximators (NNs) with the approaches in RL.

In model-based RL (MBRL), an RL agent decides on what actions to take based on a model of the environment. The model $f(s, a) = s'$ is directly related to the state transition matrix as it predicts for any state-action pair $\{s, a\}$ the resulting state s'. As previously described, the space of possible states may be intractable. The deep RL approach to model-based RL is thus to parametrize the model by an NN that approximates the true model of the environment

$$f_\Theta(s, a) \approx f(s, a) \tag{2.12}$$

based on the network weights Θ [Chu+18; DR11; Haf+18; Haf+19; HCM15; Hua+21; PJB19; RT09].

In model-free RL — we use the term RL for model-free RL throughout the following chapters — the agent does not rely on a model of the environment. Instead, a deep model-free RL agent relies on approximations of the value function $v_\pi(s)$, the Q-function $q_\pi(s, a)$ or the policy $\pi(a|s)$ itself [Sil15]. The solution for large MDPs with possibly infinite number of states is thus to estimate the above functions with parametric NNs. The formulas thus change to

$$\begin{aligned} v_\Theta(s) &\approx v_\pi(s) \\ q_\Theta(s, a) &\approx q_\pi(s, a) \\ \pi_\Theta(a|s) &\approx \pi(a|s) \end{aligned} \tag{2.13}$$

based on NNs with parameters Θ [Sil15]. The assumption made by approximating the exact functions with NNs is that these can generalize to previously unseen states and thus can reliably predict their respective outcome for the whole (possibly infinite) state space [Esp+18; FHM18; Fle+21; Haa+18; KT99; Lil+15; Mni+13; Mni+16; Sch+15; Sch+17; Zha+20].

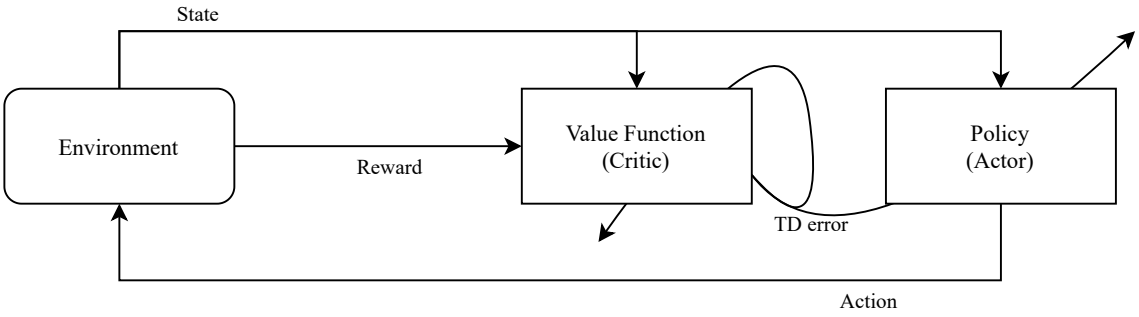


Figure 2.4: The learning scheme of an actor-critic RL agent. The actor performs an action based on the state received from the environment. The critic evaluates the temporal difference (TD) error of the new state and reward and updates the value function and policy. [Lee05]

2.1.2 Proximal Policy Optimization

On one hand, deep RL agents can rely on a learned value (or Q) function and implicitly follow the optimal policy as described in Equation 2.11. On the other hand, a pure policy-based deep RL agent does not rely on a value (or Q) function, but directly learns a policy [Sil15]. It is dependent on the problem statement when one over the other is beneficial [Sil15]. Policy-based RL agents in general show better convergence properties, navigate effective in high-dimensional or continuous action spaces and can learn stochastic policies [Sil15]. Typically, they converge to local rather than global optima, and evaluating policies is inefficient and prone to high variance [Sch+17; Sil15]

In actor-critic RL [KT99; Sch+17], the agent learns a Q-function and a policy function simultaneously. The critic in this setting is an NN that estimates the Q-function

$$q_w(s,a) \approx q_{\pi_\Theta}(s,a) \tag{2.14}$$

based on the parameters w. The actor is the policy function approximated by an NN

$$\pi_\Theta(a|s) \approx \pi^*(a|s) \tag{2.15}$$

with parameters Θ. During learning, the critic updates its Q-function network parameters w to approximate the Q-function associated with the policy π_Θ. The actor in turn, updates its own policy network parameters Θ, in a direction suggested by the critic. The update scheme of these policy-gradient-methods [Kar16; Sil15] is displayed in Figure 2.4.

Many of the recent breakthroughs in deep RL base on the actor-critic architecture. Nonetheless, the models are hard to train as they are especially sensitive to the choice of learning rate during gradient descent. A too small learning rate and the training will be slow, a too large learning rate might lead to much noise in the reward signal and the performance may catastrophically drop [SB18]. Finally, it is often necessary for the agent to take many millions (or billions) of steps in the environment to learn even simple tasks.

The state-of-the-art deep RL architectures of Proximal Policy Optimization (PPO) [Sch+17] circumvent these obstacles. With little hyperparameter tuning, a lower sample

complexity and easy implementation it outperforms other recent algorithms like ACER [Wan+16] or TRPO [Sch+15]. The main difference in training an PPO agent is the novel surrogate objective function

$$L_{clip}(\Theta) = \hat{\mathbb{E}}_t[\min(r_t(\Theta)\hat{A}_t, clip(r_t(\Theta)), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{2.16}$$

where Θ are the parameters of the policy network, $\hat{\mathbb{E}}_t$ is the empirical expectation over time steps, r_t is the ratio of the probability under the new and old policies, $\hat{A}_t$ is the estimated advantage at time t, and ϵ being a hyperparameter [Sch+17]. This objective assures that for each update step the deviation from the previous policy is relatively small and the policy does never deviate. We use PPO agents extensively in the following experiments.

2.1.3 Vision-Based Deep Reinforcement Learning

In vision-based deep RL, the learning agent does not have access to a hand-crafted numeric state of the environment. The input to the agent is one or multiple images captured by one or multiple cameras. The agent cannot observe the true state directly, but observes it through the observation function

$$o = observe(s \in S) \tag{2.17}$$

where o is an image observation capturing parts of the true state s.

In the case of robotic manipulation, training on visual data reduces the need for specialized sensors to measure the robots joint angles and velocities. Additionally, the need to measure the location of objects and targets the robot can interact with is removed. These sensors are prone to errors resulting from drift and are relatively expensive compared to cameras [San+15]. Images are high-dimensional, especially compared to hand-crafted numeric states. This imposes the problem that the networks of the RL agent increase drastically in the number of parameters when trained on images compared to the agents trained on hand-crafted numeric states. Furthermore, the networks of the image-based agent use much of their capacity on the perception of the image instead of on the task to solve. To overcome this obstacle, state representation learning is often used to extract relevant information from the images beforehand.

2.2 State Representation Learning for Reinforcement Learning

State representation learning (SRL) is a subset of unsupervised learning. Unsupervised learning is an ML technique to extract patterns from unlabelled data and compress the information into a lower dimensional representation

$$f : \mathbb{R}^D \mapsto \mathbb{R}^d; \quad d \ll D \tag{2.18}$$

with the dimension D of the unstructured input being high, and the dimension d of the representation being significantly lower. The SRL function f can be approximated by an NN f_Θ parametrized by its weights Θ. In fact, DL is a natural pick for SRL as a decreasing width of subsequent layers naturally encodes information into lower dimension. The decreasing number of neurons per layer give implicit compact representations that are of value for the following layers.

In SRL, the final layer is not a classification or regression layer as in supervised learning (paragraph 2). The networks in SRL are used to embed relevant information into the output state representation layer. The relevance of the information is task-specific and the task is not known in advance via labels as in supervised learning. A prominent example of networks using an implicit low dimension representation are variational auto encoders (VAE) [KW13]. VAEs compress images into a bottleneck via an encoding network and reconstruct the original data via a decoder network. By minimizing the reconstruction loss the encoder learns to encode all relevant information in the bottleneck representation.

The characteristics of a good state representation for RL are that it embeds the essential information needed for the task and discards irrelevant information. According to [Boh+15; Les+18] a state representation for RL must be

1. Markovian — The information in the state representation must be sufficient for an RL agent to choose an action, as the prediction of the RL agent can only depend on the current state and no later.

2. The state representation must be able to approximate the true state well enough for an RL agent to improve its policy.

3. The learned state representation must generalize well for unseen states so that the RL agent can approximate the value function.

4. The state representation must be of much lower dimension for efficient learning.

There are several approaches to SRL, as Time-Contrastive Networks (TCNS) [Ser+17] and Siamese Networks [Les+18]. Both approaches have in common that they use CNNs to extract image features from the input images. The image features are then passed through additional fully connected layers or a separate dense network to encode the relevant information. We call this combination of a CNN feature extraction backbone and some encoding layers the encoder model [Les+18] and make extensive use of it in the experiments.

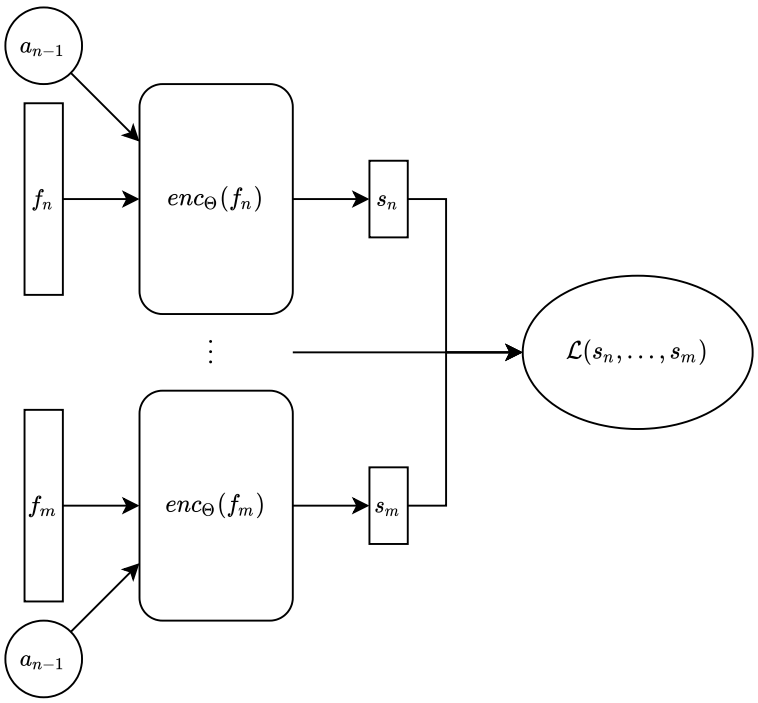


Figure 2.5: Our siamese networks enc_Θ receive the image feature vectors $f_n, \ldots, f_m$ and embed the relevant information into the latent state representation $s_n, \ldots, s_m$. We calculate the loss to train the siamese networks on similarities and differences of the latent states $s_n, \ldots, s_m$ in an unsupervised manner. We add the action a_{i-1} that resulted in the state that produced the image feature vector f_i to the input of the siamese networks to ease the prediction of the robots velocities.

2.2.1 Feature Extraction Backbone

A feature extraction backbone is a CNN (paragraph 2) that is pretrained on an extensive dataset that is related to the task. For vision-based RL a common approach [Tra+19] is to use a CNN pretrained on the ImageNet dataset [Rus+14]. The dataset consists of 14 million images from more than 20.000 categories. Pretraining a CNN on this dataset enables it to detect diverse features in the images and to classify these based on the labels.

In SRL, we are not interested in the classification of the images, but in the features in the images. To receive these features we remove the last classification layer [Tra+19]. The output of the feature extraction network f_Θ is then an image feature vector $f \in R^d$ with $d = 1024$ for a DenseNet.

2.2.2 Siamese Encoder Model

To learn a state representation, we pass the image feature vectors through the encoding layers of the encoder enc_Θ. As we do not know the structure of the embedding represen-

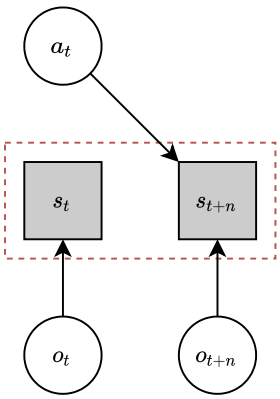


Figure 2.6: The siamese networks compare the latent states s_t, s_{t+n} at different timesteps using the robotic priors. [JB15; Les+18; Les+19b]

tation space, we need a specialized learning scheme for the SRL encoder model to explore and find patterns in the features that hint to relevant parts of the observation.

A siamese network architecture is capable to detect these relevant parts and consists of two or more networks that share the same weights. The networks learn to differentiate between their inputs and do not rely on labels, as the comparison happens on similarities and differences [JB15; Les+18; Les+19b]. Figure 2.5 displays an exemplary siamese architecture. While the networks learn to differentiate the inputs, we can put additional constraints on the shape of the latent state representation. These constraints can be on the distribution of values or on temporal causalities [Ser+17]. In SRL for RL we impose prior knowledge about the physical world and how the laws of physic constrain movements over time. We model physical properties like the slowness principle or the variability principle [JB15; Les+18; Les+19b].

2.2.3 Robotic Priors

In this thesis we are interested in RL for robotic manipulation and deal with robots that act in environments that follow the physical laws. We as humans act in the same physical domain, and we can predict the trajectory of a thrown ball as we inherently know about the underlying model of gravity, speed, and acceleration. In the same sense, the training of a SRL model can benefit from the domain knowledge of temporal, causal and physical properties of the robotic environment.

We use robotic priors [JB15; Les+18; Les+19b] to formalize the physical properties. The priors model the different temporal, causal and physical characteristics and thus guide the SRL model to detect physically relevant patterns in the data (Figure 2.6). The robotic priors shape the latent state representation so that the relevant information for the robot to act in the physical domain is preserved and the state is physically plausible. We use the robotic priors to train the siamese SRL encoder networks described in subsection 2.2.2.

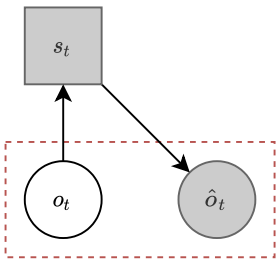


Figure 2.7: The auto-encoder model. We calculate the reconstruction loss on the difference between the reconstructed observation $\hat{o}_t$ and the actual observation o_t. [Les+18]

2.2.4 Auxiliary Models

The robotic priors have been proven to shape the latent state representation space desirably for robotic applications [JB15; Les+18; Les+19b].

Auxiliary models add additional constraints on the relationships between states at different timesteps, the actions taken at the states, and the resulting rewards. We approximate these additional relations by parametric NNs and the networks learn the actual relations in a supervised fashion. We train the siamese encoder on the output of the auxiliary models and — by backpropagation — pass the gradient through both of the models.

Decoder Model

A decoder model [Les+18] is used to reconstruct the original observation from the latent state representation [KW13] as displayed in Figure 2.7. Auto-encoder models (AE), a combination of an encoder and decoder model learn to reconstruct the input under constraints on their latent representation. The most prominent constraint is the dimensionality constraint, where the decoder reconstructs the high-dimensional input from a low-dimensional intermediate representation. AEs base on the joint learning of two functions

$$e_{\psi}^{-1}(e_{\Theta}(x)) = x \tag{2.19}$$

with e_{Θ} as the encoder NN parametrized by the weights Θ and the decoder e_{ψ}^{-1} parametrized by the weights ψ. It has been shown that siamese AE networks encode the system dynamics over time and constrain the latent representation on the transition between states [Fin+15; GML15; Hoo+19; Kar+16; Les+19b; Wat+15; WSD15].

Forward Model

The forward model [Les+18] predicts the state $\hat{s}_{t+1}$ from the observation o_t and action a_t as displayed in Figure 2.8. The transition from s_t to $\hat{s}_{t+1}$ enables the model to compare the states at subsequent timesteps.

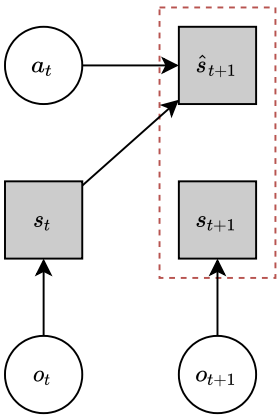


Figure 2.8: The forward model. We calculate the forward model loss on the difference between the predicted successor state $\hat{s}_{t+1}$ and the actual successor state s_{t+1}. [Les+18]

Updating the encoder based on the forward model is a two-step process. First, we encode the observations o_t, o_{t+1}, then we perform the transition from s_t to $\hat{s}_{t+1}$ using the forward model. Based on the comparison between the states $s_{t+1}, \hat{s}_{t+1}$ we compute the forward model loss. We pass the error to the encoder model by backpropagation through the forward model. Imposing constraints on the forward model enables the state representation model to further structure the state representation space [Ass+15; GML15; Hoo+19; Kar+16; Pat+17; Wat+15].

Inverse Model

The inverse model [Les+18] follows a similar idea as the forward model. Based on the states s_t, s_{t+1} the inverse model predicts the action $\hat{a}_t$ that is responsible for the transition between the states as displayed in Figure 2.9.

Updating the encoder based on the inverse model is again a two-step process. First, we encode the observations o_t, o_{t+1}, then we predict the action $\hat{a}_t$ responsible for the transition from s_t to s_{t+1} using the inverse model. Based on the comparison between the actions $a_{t+1}, \hat{a}_{t+1}$ we compute the inverse model loss. We pass the error to the encoder model by backpropagation through the inverse model.

The inverse model ensures that the state representation space includes enough information to reconstruct the actions responsible for the state transition. It thus constraints the latent space to be plausible regarding the actions of the robot [Agr+16; Pat+17; She+16; ZSP18].

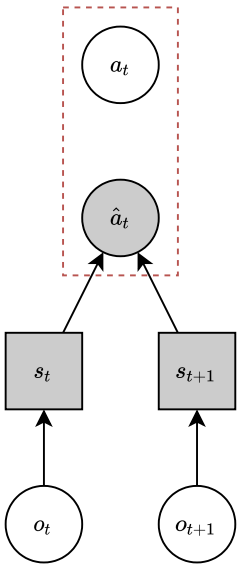


Figure 2.9: The inverse model. We calculate the inverse model loss on the difference between the predicted action $\hat{a}_t$ responsible for the state transition and the actual action a_t. [Les+18]

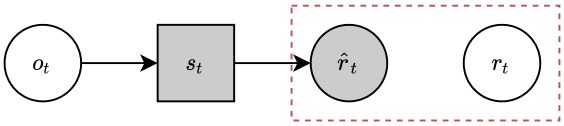


Figure 2.10: The reward model. We calculate the reward model loss on the difference between the predicted reward $\hat{r}_t$ and the actual reward r_t. [Les+18]

Reward Model

The reward model [Les+18] follows a similar idea as the forward and inverse models. Based on the states s_t the reward model predicts the reward $\hat{r}_t$ the agent receives for being in state s_t as displayed in Figure 2.10.

Updating the encoder based on the reward model is again a two-step process. First, we encode the observations o_t, o_{t+1}, then we predict the reward $\hat{r}_t$ the agent receives for being in state s_t using the reward model. Based on the comparison between the rewards $r_{t+1}, \hat{r}_{t+1}$ we calculate the reward model loss. We pass the error to the encoder model by backpropagation through the reward model.

The reward model helps the encoder model to differentiate states and to learn task related state representations. This is especially helpful to disentangle meaningful state information and to tie the states to the task [JB15; Les+18; MKB16; OSL17].

2.3 Continual Learning

In the standard CL setting an NN $f(x, \Theta)$ with parameters Θ learns a set of tasks $\{\{X_1, Y_1\}, \ldots, \{X_n, Y_n\}\}$, with the input data X_i and corresponding labels Y_i. In a multitask setting, the model learns the tasks from all datasets at once [Les+18]. This is — especially in RL — not always possible, nor is it desirable to collect all data upfront and only learn at the end of an agents life.

Instead, in CL the goal is to learn the parameters Θ_t for new tasks t, starting from the weights Θ_{t-1} and observing only the current dataset $\{X_t, Y_t\}$, while

1. Retaining old knowledge (No catastrophic forgetting),
2. Improve performance on old tasks (positive backward transfer),
3. Solve new tasks utilizing past knowledge (positive forward transfer) [Les+18].

To solve the CL problem, the following subfields of CL have emerged.

2.3.1 Regularization

NNs are trained using gradient descent to update the learnable weights, whereas the gradient of different tasks often point into different directions. CL approaches based on regularization alter the gradients to prevent forgetting.

An approach to CL is to regularize the loss on new tasks to minimize forgetting of previous tasks. This approach estimates the importance of each model parameter for the previous tasks and penalizes changes to the parameters based on their importance [Had+20]. [Kir+17] introduced the Elastic Weight Consolidation (EWC) model, where they reduced the plasticity of neurons that are important to previous tasks. They regularize using the difference between the parameters of the old and the new tasks and incorporate the fisher information matrix to estimate the importance of each neuron. [ZPG17] introduced the Synaptic Intelligence (SI) model, where each neuron locally estimates its importance in solving tasks the NN has been trained on.

[HVD15] proposed to transfer knowledge of a NN into another using knowledge distillation. When a NN approaches a new task, a copy of the NN is stored as the teacher model [Mir+19] to prevent overriding the embedded knowledge. In the Learning without Forgetting (LwF) approach by [LH16], during the ongoing training of the student model on a new task it is regularized using the copied teacher model. This forces the student model to infer the same predictions as the teacher on similar data, while learning about new data simultaneously. After training, the student model has embedded the knowledge about the previous tasks learned from the teacher together with the new task specific knowledge.

One drawback of regularization-based approaches is that regularization assumes a fixed network capacity within which the gradients or importance are altered to embed task knowledge. This approach implies that with a fixed capacity the possible amount of knowledge is also fixed.

2.3.2 Dynamic Architectures

In the dynamic architectures approach the network architecture is changed as new tasks are solved, opposed to the regularization-based approaches where gradients are changed.

Explicit architecture modifications can be done by adding, cloning or saving parts of the model parameters to avoid forgetting. In the Progressive Neural Networks (PNN) model by [Rus+16] they add a new network whenever there is a new task. They transfer knowledge of old tasks to the new instances via lateral connections between the networks layers. This approach specifically prevents catastrophic forgetting as they only train the newest instance and fix the old networks weights. No backwards transfer can be performed by PNNs as older network instances have fixed weights. In the LwF approach [LH16] the authors add output layers per task and thus reuse lower layer knowledge across tasks. The dynamically expanding network (DEN) models of [Yoo+17] increase their number of trainable weights according to the estimated needed capacity for new tasks.

Implicit architecture modifications inactivate neurons based on their importance for learned tasks or change the forward pass path according to the task to solve. These approaches do not change the architecture of the network, but rather introduce an ensemble of networks within the network. They achieve the implicit modification by masking parameters per task as it is done in the PackNet and Piggyback models by [MDL18; ML17]. The Hard Attention to the Task (HAT) model by [Ser+18] uses an attention mechanism that prevents forgetting of previous task information while preserving plasticity of the network to learn new tasks. For training the PathNet model by [Fer+17] the authors used a genetic algorithm to learn the optimal paths for gradient flow through the network per task.

Dual and multi architecture approaches split the network architecture into two or more models. A plastic and fast adapting model learns the currently observed task, while one or multiple models store memory about past experiences. In the FearNet model by [KK17] three models resemble the dual memory as found in the human brain.

A problem with dynamic architectures is the growing memory consumption, as new tasks require the allocation of network capacity.

2.3.3 Memory Systems

Inspired by the human brain there is also the approach of using memory systems to encode, store and recall knowledge or experience.

Replay-based — also called rehearsal-based — approaches store a set of observations from previously seen tasks. During training of the NN the memory system replays samples of the stored observations and interleaves them with currently seen observations. The NN thus sees data from different tasks simultaneously and learns as in a multitask setting. To minimize memory consumption we only store observations that have a high impact for learning, such as observations that exhibit a large loss or resulted in strong gradients. In the iCaRL model by [Reb+16] the authors carefully sort the seen

observations and only store a coreset consisting of only the most representative samples. Active Long Term Memory (A-LTM) networks introduced by [Fur+16] store a coreset that is used for regularization during knowledge distillation. Gradient Episodic Memory (GEM) and Averaged Gradient Episodic Memory (A-GEM) networks by [Cha+19; LR17] regularize the gradients based on the stored episodes.

Pseudo-rehearsal approaches do not store explicit knowledge, but rather encode knowledge into a generative model. The generative models are usually autoencoder (AE) networks [KW13] or generative adversarial (GAN) networks [Goo+14]. During training, the pseudo-rehearsal system generates sample experiences from previous tasks and interleaves them with current experience. The model learns to solve the current and the replay task jointly and prevents catastrophic forgetting of old tasks. The FearNet model by [KK17] uses pseudo-rehearsal to resemble the long term memory of the human brain. In the Reinforcement-Pseudo-Rehearsal (RePR) model by [Atk+20] the authors use a combination of knowledge distillation and generated observations to transfer knowledge from a teacher to a student model.

Comparably to the dynamic architectures approach, one problem is the increasing memory consumption when storing experience in rehearsal and the possible demanding training of generative models for pseudo-rehearsal.

2.3.4 Meta-Learning

Meta-learning, often also called learning to learn, is a data driven approach for improving an agent's learning efficiency [Khe+20]. This approach is less concerned with forgetting old knowledge, but rather how to quickly adapt to new environments and tasks. Agents in this regime learn to alter their own optimization process, whereas the modifications to the learning process will generalize into the future.

In the meta-training and meta-testing setting a NN attempts to learn to alter its own optimization process. The learning consists of an inner loop that is responsible for fast time-scale learning and quick adaption to new tasks. Another outer loop is a slower process of learning about the learning of the inner loop. The meta-training phases consist of meta-updates where the inner loop optimizes on a meta-train training dataset. Based on the performance of the inner loop based on a meta-train testing dataset the outer loop performs a learning step. During a final meta-testing phase the inner loop trains on a hold out meta-test training dataset. The performance and the ability to quickly adapt to the meta-testing environment is measured on a meta-test testing dataset. The Model Agnostic Meta-Learning (MAML) framework by [FAL17] performs standard gradient descent within the inner loop, whereas the outer loop computes a gradient to improve the performance of the inner loop learning process.

2.3.5 Hypernetworks

A combination of the discussed CL approaches are task-conditioned hypernetworks. Hypernetworks generate the weights of a target model based on a given task identity.

In Continual Learning with Hypernetworks [Osw+19], the authors proposed a memory system approach at the meta level. Instead of retaining the parameters Θ of a task model $f(x, \Theta)$, a task-conditioned hypernetwork $f_h(e, \Theta_h)$ learns to map from a task embedding e to the task network weights Θ, parametrized by the hypernetwork parameters Θ_h. Using this conditioning, only a low dimensional task embedding e has to be memorized per task, in contrast to storing whole datasets $\{\{X_1, Y_1\}, \ldots, \{X_n, Y_n\}\}$ as in the memory systems approach (subsection 2.3.3).

In the memory systems approach, data from previous tasks and the corresponding model outputs are stored. To fix the model outputs while learning new tasks, an output regularizer

$$\mathcal{L}_{output} = \sum_{t=1}^{T-1} \sum_{i=1}^{|X_t|} \left\| f(x_{t,i}, \Theta^*) - f(x_{t,i}, \Theta) \right\|^2 \tag{2.20}$$

is used [Osw+19], where Θ^* are the network weights before learning the new task T, and f is the learning model [BRK18; LH16; Osw+19; Rob95]. This approach requires storing and repeatedly iterating over the previous data of the X_t, which is infeasible in practice regarding physical memory and computational constraints.

A hypernetwork deviates from the idea of storing input-output data points, and approaches the CL problem by maintaining sets of parameters $\{\Theta_t\}$, without explicitly storing them [Osw+19]. The task-conditioned hypernetwork regenerates the target network weights $\{\Theta_1 \ldots, \Theta_{T-1}\}$ while learning to generate the target weights Θ_T of the current task. Task-conditioned hypernetwork can be viewed as weight generators [HDL16; Sch92; VT18]. Instead of learning the parameters Θ of a specific NN f_Θ (the target model) directly, the hypernetwork learns the parameters Θ_h of the metamodel. The output of the metamodel, the hypernetwork, is Θ. The hypernetwork $f_h(e, \Theta_h)$ learns to map embedding vectors e to the target weights Θ.

Hypernetworks thus sidestep the problems of memory systems. In target network space, a single point has to be fixed per task, what can be efficiently achieved with task-conditioned hypernetworks by fixing the hypernetwork output on the respective task embedding. At first, a candidate change $\Delta\Theta_h$ that minimizes the loss $\mathcal{L}_{task}(\Theta_h, e_t, X_t, Y_t)$ w.r.t. the target network weights Θ is computed using any optimizer. The change of hypernetwork parameters is then calculated by minimizing the total loss

$$\begin{aligned} \mathcal{L}_{total} &= \mathcal{L}_{task}(\Theta_h, e_T, X_T, Y_T) + \mathcal{L}_{output}(\Theta_h^*, \Theta_h, \Delta\Theta_h, \{e_t\}) \\ &= \mathcal{L}_{task}(\Theta_h, e_T, X_T, Y_T) + \frac{\beta_{output}}{T-1} \sum_{t=1}^{T-1} \left\| f_h(e_t, \Theta_h^*) - f_h(e_t, \Theta_h + \Delta\Theta_h) \right\|^2 \end{aligned} \tag{2.21}$$

where Θ_h^* are the hypernetwork parameters before learning the task T, $\Delta\Theta_h$ is considered fixed, and β_{output} is a hyperparameter to control the strength of the regularizer [Osw+19].

Notable is that — unlike in Equation 2.20 — $\mathcal{L}_{output}$ does not depend on past data, but only on the collection of previous task embeddings $\{e_t\}_{t=1}^{T-1}$.

Hypernetworks do not suffer from catastrophic forgetting, as it has been shown on numerous CL benchmarks [Osw+19]. Instead, hypernetworks are capable of retaining memories with practically no decrease in performance [Osw+19] over long sequences of tasks. Additionally, task-conditioned hypernetworks exploit tasks-to-task similarities and transfer information forward in time to future tasks [Osw+19].

CL is a natural fit for hypernetworks, as they do not recall input-output relations as in the case of memory systems (subsection 2.3.3), nor do they add additional components like dynamic architectures (subsection 2.3.2). Additionally, neither do they learn specialized training schemes as the meta learning approaches (subsection 2.3.4), nor do they require complex regularization (subsection 2.3.1). In fact, hypernetworks only need to store the task embeddings in memory, which for even numerous tasks is a neglectable increase in capacity and computation. This approach enables hypernetworks to exhibit large capacity for retaining previous memories even on long sequences of tasks [Osw+19].

In the following approach and experiments we propose — to the best of our knowledge — the first hypernetwork-based approach to SRL and the first hypernetwork-based approach to model-free RL.

3 Related Work

The goal of our thesis is to enable a RL agent to learn from images only and continually. The two domains of vision-based RL and continual RL are under fast-paced development and recent research has brought impactful improvements in the fields.

3.1 DisCoRL: Continual Reinforcement Learning via Policy Distillation

In Continual Reinforcement Learning via Policy Distillation (DisCoRL) [Tra+19] the authors present a vision-based method for CRL combining SRL and policy distillation [Rus+15]. The first step in the proposed architecture is to sample raw image data from the environment following a random policy. On the sampled data, the authors train an AE to learn a state compression that provides useful features as input for the policy learning network. Once the learning network has converged to the policy π, the authors generate in an on-policy manner a distillation dataset D_π that consists of sequences of observations and associated actions. For each task i, the authors store a separate distillation dataset D_{π_i} and use it to distill the stored policy π_i (the teacher) into a student model $\pi_{d:i}$. Opposed to the originally proposed policy distillation procedure [Rus+15], the authors only store about 10K samples per distillation set and do not require to store the original policy network. This enables them to distill the knowledge of all D_{π_i} into one final policy $\pi_{d:1,\dots,n}$, which learns all tasks while avoiding catastrophic forgetting.

The authors train a 3-wheel omnidirectional robot on a sequence of three simulated 2D navigation tasks. They combine techniques of SRL and continual RL to train the robot in simulation and successfully transfer the model to a real robot. Opposed to our hypernetwork-based approaches, the authors present a non-continual approach to SRL combined with a regularization-based approach (subsection 2.3.1) for continual RL.

3.2 Sim-to-Real Robot Learning from Pixels with Progressive Nets

DeepMind introduced Sim-to-Real Robot Learning from Pixels with Progressive Nets [Rus+16; Rus+17] and uses the proposed method to train a robotic manipulation arm on image observations on a sequential set of tasks. To close the reality gap, the authors mix tasks in simulation and in reality. The approach bases on progressive neural networks [Rus+16], a CRL approach using dynamic architectures as described in subsection 2.3.2.

The authors train an actor-critic network in simulation on several tasks. When the network has learned the current task, they freeze the network weights and initialize a new column network. The new network has lateral connections between each of its layers and the corresponding layers of previous columns. The lateral connections allow the new column network to reuse all knowledge of the previously learned tasks while being flexible to learn the new task. The connections on every layer allow the networks to reuse all knowledge, from low-level visual features to high-level policies.

The authors trained several column networks on tasks in simulation and then trained the last column network on a real robotic arm. This enabled them to transfer the knowledge of simulation into reality while avoiding catastrophic forgetting and performing forward transfer. Opposed to our hypernetwork-based approach, the authors propose an approach based on a dynamic architecture that allocates new models with each task. They show that in their setup, a robotic manipulation arm trained on image observations of multiple tasks in simulation can quickly adapt to a real setup within some hundred attempts. This closes the reality gap, and makes using the robotic manipulation arm in reality feasible.

3.3 S-TRIGGER: Continual State Representation Learning via Self-Triggered Generative Replay

In S-TRIGGER: Continual State Representation Learning via Self-Triggered Generative Replay [DGF19] the authors present a generative approach to continual SRL. They propose to use generative replay (subsection 2.3.3) to generate samples of past tasks during the training on the current task. Their model efficiently compresses the observations without catastrophic forgetting and enables RL agents trained on the latent state representation to quickly solve given tasks.

Opposed to our approach the authors solve the continual SRL approach using the generative approach. The generative model stores knowledge about past tasks in the regenerated samples and prevents catastrophic forgetting by interleaving the past knowledge with current observations. This approach exhibits the problem detailed in subsection 2.3.5 that during learning multiple iterations over the whole generated dataset need to be done to prevent forgetting. Hypernetworks sidestep this problem by only storing the weight realizations as single point in target network space, with the trade-off of not being task-agnostic.

3.4 Continual Model-Based Reinforcement Learning with Hypernetworks

In Continual Model-Based Reinforcement Learning with Hypernetworks [Hua+21] the authors present a hypernetwork-based approach to model-based RL. As described in subsection 2.1.1, in model-based RL the environment model

$$f_\Theta(s_t, a_t) = s_{t+1} \tag{3.1}$$

predicts the upcoming state s_{t+1} of the environment based on the current state s_t and the action a_t. The authors propose to generate the model parameters Θ by a hypernetwork

$$f_h(\{e\}, \Theta_h) = \Theta \tag{3.2}$$

based on the learnable task embeddings $\{e\}$ and the hypernetwork parameters Θ_h. The authors show that their approach prevents catastrophic forgetting of the environment model over a sequence of five robotic manipulation tasks.

Their approach is similar to our approach as they use hypernetworks to prevent catastrophic forgetting in a robotic manipulation setup. They focus on continual model-based RL on hand-crafted numeric observations, whereas we approach the problem of vision-based continual model-free RL.

4 Approach

In this thesis we propose — to the best of our knowledge — for the first time to solve the problem of vision-based CRL for robotic manipulation tasks based on hypernetworks [Osw+19]. In the following chapter we describe in-depth our novel approach to hypernetwork-based continual SRL and hypernetwork-based model-free continual RL.

4.1 Environment

We use the KUKA LBR iiwa robot [Kuk21] displayed in Figure 4.1. It is an industrial grade assembly robot that is lightweight and designed for cooperation with humans. The robot consists of seven rotating joints that can move between 252 and 359 degrees. The seven degrees of freedom enable the robot to move freely in a radius of up to 1.6 meters around its base. We aim to control the robot using the model-free RL agent detailed in section 4.3.

Training robot control NNs on real robots has several drawbacks: 1. Training is time-consuming 2. The sensors are prone to measurement errors, as a camera might be accidentally moved, or rotational sensors might drift 3. As the robot explores the state space, collisions with the floor or itself can occur, which may cause damage on the robot. We thus use a simulator with a virtual KUKA LBR iiwa robot to train the networks without the drawbacks of the above-mentioned real-world obstacles.

4.1.1 Simulator

The simulator we use in this thesis is developed at the Chair of Robotics, Artificial Intelligence and Real-time Systems at Technische Universität München (TUM) [Jos+20]. The simulator uses the Unity [Haa14] game engine and mainly its built-in physics and rendering engines. The simulator acts as a server and distributes the simulation result using the gRPC protocol [Goo].

The simulator provides a variable number of simulation instances and one KUKA LBR iiwa robot, a target, and an object reside in one instance. The center of the base of the robot defines the origin of the right-handed reference coordinate system $\mathbb{R}^3$ per simulation. The shapes, colors and physical properties of the objects and targets are task specific and explained in the following sections.

We interact with the simulator via python scripts that wrap the functionality of one simulation following the OpenAI Gym [Bro+16] interface. The simulations are wrapped in a vectorized environment using Stable-Baselines3 [Raf+21]. The client requests a step of the simulator by sending the actions for all simulation instances via the gRPC method.

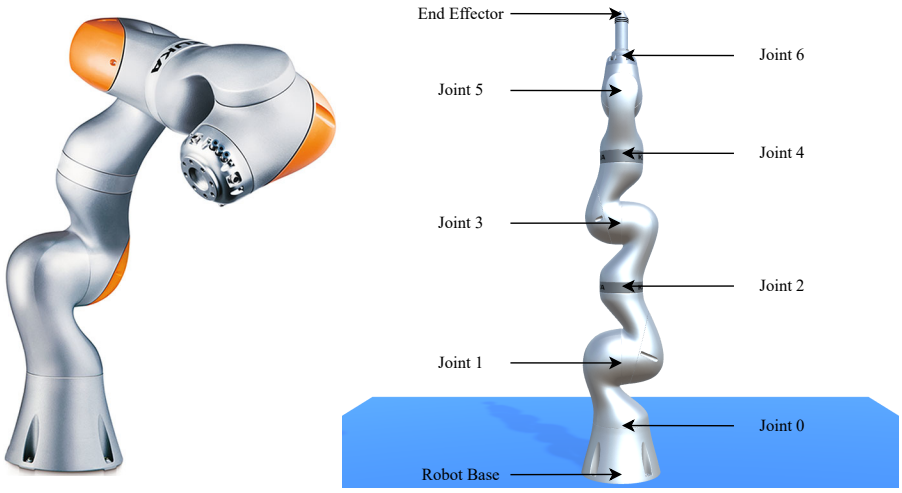


Figure 4.1: Left: The Kuka IIWA [Kuk21] robot we use in this thesis.[2] — Right: The virtual robot we use in the simulation. The seven joints of the robot are labelled with their respective indices used to access it in the simulation.

[2] Source: `https://robots.ieee.org/robots/lbriiwa/`.

The server replies with the observations and the rewards from the simulations after applying the actions and executing one simulation timestep. Figure 4.2 displays a schema of the client-server simulator architecture. Each episode in the simulator consists of 250 timesteps of 20 milliseconds. We do neither reset the simulation on a collision nor when the agent solves its task.

4.1.2 Action Space

An image of the robot can be seen in Figure 4.1. All seven joints of the robot can be controlled. The RL agents generate the actions on the client side and send the actions to the simulation server. The simulator expects actions a with entries $a_i \in [-1, 1]$. The action gets multiplied with a fixed velocity multiplier $\omega = 200$ so that arbitrary — but fixed per simulation — maximal speeds can be reached. Joints $\{0, 2, 4, 6\}$ can rotate from -180 to 180 degrees, Joints $\{1, 3, 5\}$ can rotate from -126 to 126 degrees.

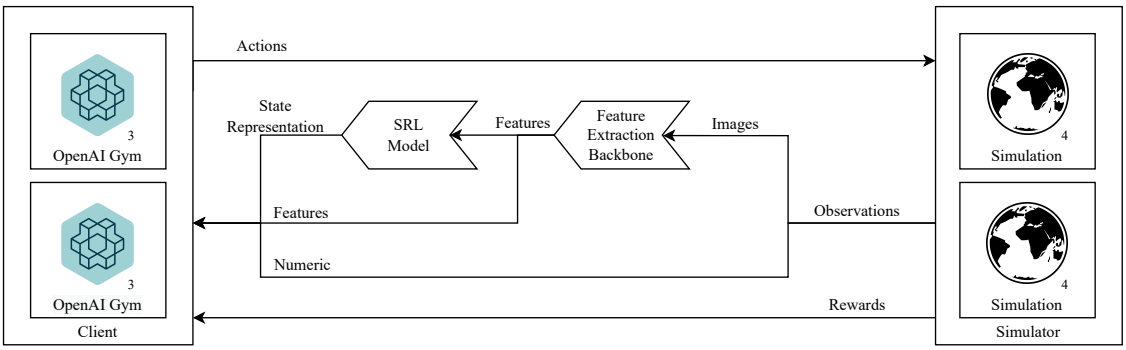


Figure 4.2: The schema of communication with the simulator. The client sends actions to the simulator via gRPC. The simulator server replies with the rewards and observations. — We pass the numeric observations directly to the client. We forward the image observations to the feature extraction backbone and either pass the image feature vectors to the client or the SRL model. The SRL model passes a latent state representation to the client.

[3] Source: `https://gym.openai.com/assets/dist/home/header/home-icon-54c30e2345.svg`

[4] Source: `https://upload.wikimedia.org/wikipedia/commons/thumb/6/60/Simple_Globe.svg/1024px-Simple_Globe.svg.png`

4.1.3 Observation Spaces

To answer the research questions (section 1.2) we implement the simulator to output either hand-crafted numeric observations or RGB-images.

Numeric In the numeric setting, the space of possible observations the simulator may return consists of vectors $o \in \mathbb{R}^{32}$ with the 32 entries as described in Table 4.1. We observe that training the robot control NNs on unnormalized state vectors leads to a low training and evaluation performance. We thus normalize the numeric state vectors o to be $o_{normalized} \in [-1, 1]^{32}$ using the ranges displayed in Table 4.1.

Vision-Based In the vision-based setting, the space of observations are RGB-images $o_{image} \in [0, 255]^{224 \times 224 \times 3}$ with a width and height of 224 pixels with 3 color channels per pixel. We capture the images with a virtual camera placed above the robot at a distance of 3 meters from the robots base, facing vertically downwards. We first normalize the images to be in $o_{image_{normalized}} \in [0, 1]^{224 \times 224 \times 3}$ by dividing by 255 and standardize the normalized images using the mean and standard deviation of the ImageNet [Rus+14] dataset. Figure 4.3 displays some exemplary images taken from the simulation.

We pass the images through a feature extraction backbone based on the DenseNet architecture (paragraph 2). The backbone extracts image feature vectors $f \in \mathbb{R}^{1024}$ from the images and lowers the dimension from $224 * 224 * 3 = 150.528$ to 1024. This first dimensionality reduction is beneficial for the downstream tasks as described in

Observation Entry	Name	Range
0 – 2	End effector position	±2 [m]
3 – 5	End effector direction	±1
6, 8, 10, 12	Joint angle (0, 2, 4, 6)	±180 [deg]
7, 9, 11	Joint angle (1, 3, 5)	±126 [deg]
13 – 19	Joint angle velocity	±10 [rad/s]
20 – 22	Target position	±2 [m]
23 – 25	Target direction	±1
26 – 28	Object position	±2 [m]
29 – 31	Object direction	±1

Table 4.1: The entries of the numeric observations. All positions are given in meters, the direction vector is unitless and faces in z-direction if all entries are zero. The joint angles are given in degrees and converted to radians, the joint velocities are given in radians per second.

subsection 2.2.1. We use the image feature vectors in three subsequent tasks:

1. Train the RL agent directly on the image feature vectors $f \in \mathbb{R}^{1024}$.
2. Pass the image feature vectors through a SRL model $srl : \mathbb{R}^{1024} \mapsto \mathbb{R}^{128}$ mapping to a lower-dimensional embedding space. Train the RL agent on the latent state embedding $s \in \mathbb{R}^{128}$.
3. Pass the image feature vectors through a continual SRL model $csrl : \mathbb{R}^{1024} \mapsto \mathbb{R}^{128}$ mapping to a lower-dimensional embedding space. Train the RL agent on the latent state embedding $s \in \mathbb{R}^{128}$.

We do not normalize the image feature vectors and latent space embeddings, as we do not know the ranges prior to training the robot. In future research a running estimate of the statistics of the image feature distribution may be used for normalization, but this is out of the scope of this thesis.

Figure 4.3: The view of the camera. The images are blurry as the camera captures at a resolution of 224×224 pixels. — Left: An image of the environment of the reaching task. The black target disk is visible on the lower left, the robot in its initial configuration faces vertically upward. — Right: An image of the environment of the pushing task. The green box is visible on the top right, the red target square is visible on the right. The robot in its initial configuration has its upper part nearly parallel to the floor.

4.1.4 Tasks

In this thesis we solve the problem of vision-based continual RL for sequential robotic manipulation tasks. We implement two distinct tasks in the simulation environment:

Reaching

The first task we implement is a reaching task and the robot has to touch a target with its end effector. The initial joint angles are: 90 degrees for Joint 4, 0 else. We constrain the robot to only control the three joints $\{0, 1, 3\}$. The position of the end effector is displayed in Figure 4.1, exemplary images seen from the virtual camera are displayed in Figure 4.3. This leaves the robot in the upright initial position displayed in Figure 4.1 and Figure 4.3.

The target the robot tries to reach is defined as a two-dimensional black disk with a diameter of 10 cm as shown in Figure 4.3. The disk is not subject to gravity but hovers freely at its spawn location. The disk is oriented to be parallel to the floor. We spawn the target disk semi-randomly: We randomly generate an angle $\alpha \in \pm\pi$, and a radial distance $r \in [0.5, 1]$ in meters. The position of the target is calculated as the vector $t := (sin(\alpha) * r, 0.5, cos(\alpha) * r) \in \mathbb{R}^3$.

We place the target at the fixed height $t_1 = 0.5$ meters as we observe that estimating

the height is especially hard in our monocular setting. We presume that the learning models benefit from the fixed target height, as no depth has to be estimated from the single camera image. We decide on this relaxation of the reaching problem, as we focus on vision-based continual RL, especially evaluating the performance training on camera images and in a continual fashion. As monocular depth estimation is an open field of research [Bho19] we fix the target height at 0.5 meters above the ground.

Reaching Reward Function We train the RL agent via the dense reaching reward function

$$\begin{aligned} r_t^{reaching} = &- \left\| p_t^{endeffector} - p_t^{target} \right\|_2 + 1 \\ &+ task(p_t^{endeffector}, p_t^{target}) \\ &- collision(s_t) \end{aligned} \tag{4.1}$$

This reaching reward function incorporates three separate terms the agent optimizes:

1. $-\left\| p_t^{endeffector} - p_t^{target} \right\|_2 + 1$ is the negative euclidean distance between the tip of the end effector and the center of the target, shifted by one. This main objective of the robot is monotonically increasing as the end effector approaches the target. Shifting the negative distance by one, the interval of possible values is $[-\infty, 1]$. The shift to have distances lower than one meter to result in a positive reward encourages the robot to solve the task as fast as possible, to accumulate as much reward as possible.

2. $task(p_t^{endeffector}, p_t^{target})$ is a binary reward signaling the robot if the reaching task is solved. This signal takes on two values: 1 if $\left\| p_t^{endeffector} - p_t^{target} \right\|_2 < 0.1$, 0 else. This reward signal enforces a small final distance of the end effector and target.

3. $-collision(s_t)$ is another binary reward signaling the robot if a collision with the floor occurred. Based on the robot state s_t, $collision(s_t)$ is 1 if a collision occurred, 0 else.

We employ no reward clipping, as we observe that training does not benefit from it.

Pushing

The second task we implement is a pushing task and the goal of the robot is to push a green box onto a red square target. The initial joint angles are: -118.8 degree for Joint 1, -18 degree for Joint 3, 90 degree for Joint 4, 0 else. Figure 4.3 displays the robot in its initial configuration, the target, and the object seen from the virtual camera. The robot bents sharply at joint 2, and in the opposite direction at joint 4. This bending renders the upper part of the robot in a nearly parallel position to the floor. We constrain the robot to only control the three joints $\{0, 5, 6\}$. The robot is thus only able to rotate around its base and to rotate the end effector.

We use this relaxation of the pushing objective so that the robot cannot collide with the floor. We observe that giving the robot more degrees of freedom is possible but requires a

signal to avoid collisions with the floor. We observe that including a collision penalty — as in the reaching task — inhibits learning for a large amount of training iterations. For a few million timesteps the RL agent only learns to avoid collision and only after many million timesteps starts to discover how to solve the pushing task. We thus decided to relax the problem using the above-mentioned initial configuration to lower training times, which especially in the vision-based settings was crucial for development. This relaxation is without loss of generality, as we focus on vision-based continual RL, and not on solving the RL problem with as many joints as possible. We leave the extension of the ideas to harder robot configurations as future research.

We spawn the target and object semi-randomly in a two-step random process:

1. Randomly choose a side $s \in \{-1, 1\}$ of the initial robot configuration.
   Generate a random angle $\alpha \in [0.25 * \pi + 0.2, 0.5 * \pi + 0.2]$.
   Set the target position to $t := (s * sin(\alpha) * r, 0.005, s * cos(\alpha) * r) \in \mathbb{R}^3$ with the radial distance $r := 0.9$ meters.

2. Set the object position
   $o := (s * sin(\alpha - 0.25 * \pi) * r, 0.005, s * cos(\alpha - 0.25 * \pi) * r) \in \mathbb{R}^3$ with the radial distance $r := 0.9$ meters.

Setting the object in the angular distance of $0.25 * \pi$ radians from the target ensures that the initial distance between the object and target are equal over the runs. This makes training the robot feasible in a low number of timesteps and comparable across simulations. The random generation of the target position on the half circle around the initial robot configuration ensures that the RL model does learn a meaningful policy.

Pushing Reward Function We train the RL agent via the dense pushing reward function

$$r_t^{pushing} = max(-\left\|p_t^{object} - p_t^{target}\right\|_2 * s + 1, -1) \tag{4.2}$$

$$+ task(p_t^{object}, p_t^{target}) \tag{4.3}$$

This reward function incorporates two separate terms the agent optimizes:

1. $max(-\left\|p_t^{object} - p_t^{target}\right\|_2 * s + 1, -1)$ is the negative distance between the center of the object and the center of the target, scaled by $s := 2$ and shifted by one. This is the main goal of the robot, as it is monotonically increasing as the object approaches the target. Shifting the negative distance by one and clipping the reward at minus one, the interval of possible reward values is $[-1, 1]$. The scaling and shift to have distances lower than 0.5 meter to result in a positive reward encourages the robot to solve the task as fast as possible, to accumulate as much reward as possible.

2. $task(p_t^{object}, p_t^{target})$ is a binary reward signaling the robot if the pushing task is solved. This signal takes on two values: 1 if $\left\|p_t^{object} - p_t^{target}\right\|_2 0 < 0.1$, 0 else. This reward signal enforces a small final distance of the object to the target.

We employ no reward clipping as we observe that training does not benefit from it.

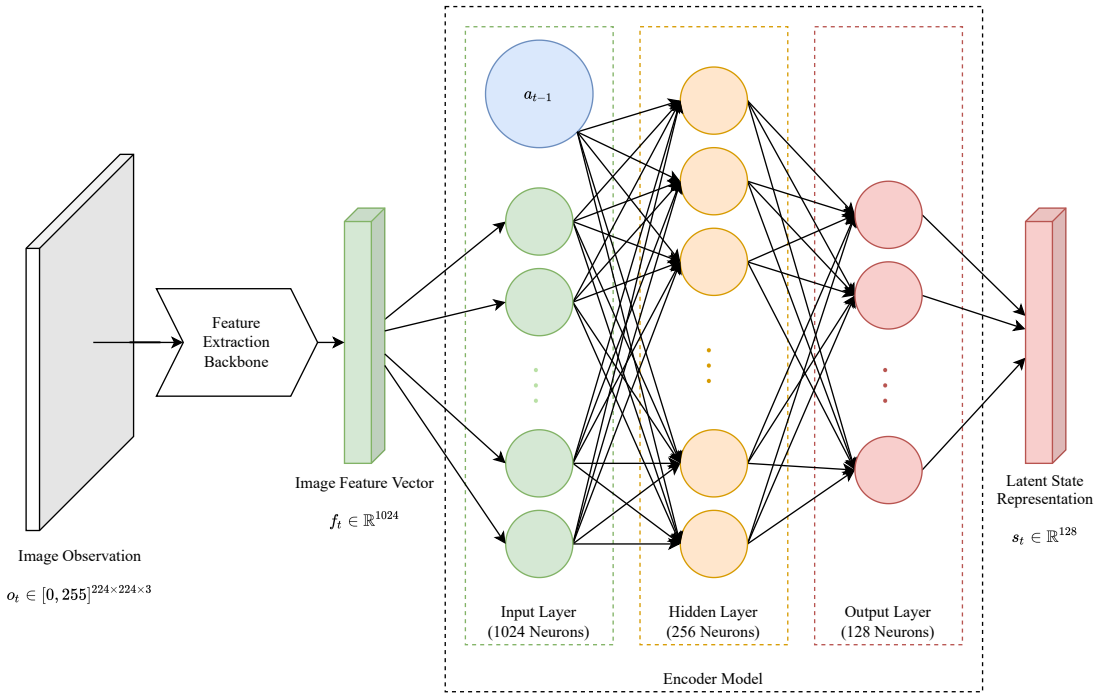


Figure 4.4: The architecture of the SRL encoder model. The feature extraction backbone extracts image feature vectors f_t from the image observations o_t. The input layer of the encoder model expects a stacked vector of: 1) The action a_{t-1} taken at state s_{t-1} that resulted in the observed image feature vector f_t. 2) The image feature vector f_t. The encoder model outputs the latent state representation s_t.

4.2 Continual State Representation Learning

As detailed in chapter 1 we decouple the perception and the interaction of the RL agent. We enable our robot to perceive a meaningful state by training a SRL model to encode relevant information from images into a state embedding. The image observations from the simulator pass through a DenseNet feature extraction backbone model (subsection 2.2.1) and then through the encoder model (subsection 2.2.2). The feature extraction backbone performs the mapping $f : \mathbb{R}^{224 \times 224 \times 3} \mapsto \mathbb{R}^{1024}$ from RGB images to the image feature vector.

4.2.1 Encoder Architecture

The SRL encoder model performs the mapping $enc_\Theta : \mathbb{R}^{1024+3} \mapsto \mathbb{R}^{128}$ from image feature vectors and previous actions to the latent state representation as detailed in subsection 2.2.1.

The encoder model enc_Θ is an NN parametrized by the network weights Θ. The NN consists of the input layer with $1024 + 3$ neurons, one hidden layer of 256 neurons and the output layer of 128 neurons. The input to the encoder is the image feature vector

Entry Name	Description	Ranges
Previous action	The action taken at the previous timestep.	$a_{t-1} \in [-1, 1]^3$
Previous features	The image feature vector corresponding to the previous image observation.	$f_{t-1} \in \mathbb{R}^{1024}$
Action	The action taken at the current timestep.	$a_t \in [-1, 1]^3$
Features	The image feature vector corresponding to the current image observation.	$f_t \in \mathbb{R}^{1024}$
Reward	The reward received after observing the image feature vector f_t and taking action a_t.	$r \in \mathbb{R}$
Previous numeric state	The numeric state observed at the previous timestep. Only used for evaluation.	$s_{t-1} \in \mathbb{R}^{32}$
Numeric state	The numeric state observed at the current timestep. Only used for evaluation.	$s_t \in \mathbb{R}^{32}$
Is training	A flag if the data point belongs to the training or validation set.	$t_t \in [True, False]$

Table 4.2: A data point in our dataset tailor towards state representation learning with robotic priors.

$f_t \in \mathbb{R}^{1024}$ corresponding to the image observation at the current timestep t stacked with the action $a_{t-1} \in \mathbb{R}^3$ taken at the previous timestep $t-1$ that resulted in the current observation. The complete input to the encoder is thus the stacked vector $s \in \mathbb{R}^{1024+3}$. Stacking the image feature vector with the action taken at the previous timestep enables the encoder to better predict the velocities at the joints, as the velocity is directly related to the acceleration resulting from the action. The architecture is visualized in Figure 4.4.

4.2.2 Dataset for Robotic Priors

To train the SRL encoder model, we generate a dataset specifically tailored to learning meaningful state representations. The dataset consists of data points according to the schema displayed in Table 4.2.

We record the dataset prior to training the SRL encoder. For each task, we record a dataset consisting of $\sim$ 200.000 training data points and $\sim$ 50.000 validation data points. To record the datasets, we employ pretrained PPO agents trained on the numeric features. To avoid overfitting the data set to only include data points that are directly from solving the desired task, we sample from the action distribution that the PPO agent generates. This generation scheme ensures that the data set includes meaningful states as well as random states to sample the whole observation space.

The proportionality, repeatability, and causality robotic priors ([JB15; Les+18; Les+19b]) require pairs of data points that include the same action taken for different states and timesteps. To generate these pairs of data points, we generate a set of actions only for

a randomly selected half of the simulations. The other half of the simulations gets the same set of actions randomly assigned prior to passing the actions to the simulator and taking a step in the environment. This ensures that there always exists at least one other data point that shares the same action taken, but at a different state of the simulator. Not always taking the predicted optimal action, but a randomly assigned optimal action from another simulation, additionally ensures that the sampling model thoroughly explores the state space.

4.2.3 Encoder Learning Scheme

We employ a siamese network architecture as described in subsection 2.2.2. This is necessary to calculate the robotic priors that compare the latent state representations at different timesteps. By comparing the latent states the encoder learns to detect similarities and differences in the observations and embeds the physical information in the state representation.

We randomly sample data points from the training data set and retrieve the data points that share the same action. We pass the data points through the siamese encoders that share the same network weights to extract the latent state representations as described in subsection 2.2.2.

Robotic Priors Loss We calculate the loss represented by the robotic priors [JB15; Les+18; Les+19b] as

$$\begin{aligned}\mathcal{L}_{Priors} = \rho_{Slowness} * \mathcal{L}_{Slowness} + \rho_{Variability} * \mathcal{L}_{Variability} \\ + \rho_{Proportionality} * \mathcal{L}_{Proportionality} + \rho_{Repeatability} * \mathcal{L}_{Repeatability} \\ + \rho_{Causality} * \mathcal{L}_{Causality}\end{aligned} \tag{4.4}$$

where the $\mathcal{L}_{prior}$ are the respective robotic priors and ρ_{prior} is a weighting factor for the individual priors. Through hyperparameter search we found the best set of ρ_{prior} to be: $\rho_{Slowness} = 0.05$, $\rho_{Variability} = 3$, $\rho_{Proportionality} = 0.05$, $\rho_{Repeatability} = 0.05$, $\rho_{Causality} = 4$.

Inverse Model Loss In conjunction with the robotic priors, we use some of the auxiliary losses described in subsection 2.2.4. We use the inverse model that predicts the action $\hat{a}_t$ from the state transition $s_t \mapsto_{a_t} s_{t+1}$. The inverse model is an NN that approximates the action prediction function parametrized by the network parameters Θ. The NN consists of one input layer with 128 neurons, two hidden layers with 128 neurons and an output layer with three neurons. We define the inverse model loss

$$\mathcal{L}_{Inverse} = \|\hat{a}_t - a_t\|_1 \tag{4.5}$$

as the absolute distance between the predicted and the actual action. The input to the inverse model is the latent state representation of two successive states and outputs the action responsible for the state transition. We pass the inverse model loss through the inverse model to the encoder model to update the encoder parameters.

Parameter	Value
Number of training data points (per task)	~ 200.000
Number of validation data points (per task)	~ 50.000
Normalization (Features, Actions, Rewards)	Mean Normalization
Batch size	256
Epochs	50
Bias (Neuron)	True
Activation function	Leaky ReLU
Leaky ReLU Slope	0.01
Optimizer	Adam
Initial learning rates	0.003
Learning rate schedule	Step Decay [WMJ21]
Step Decay: Step size	5
Step Decay: Gamma	0.25
Gradient clipping (upper limit)	5
Weight Decay (only inverse model)	0.08

Table 4.3: The common training parameters of the models used in non-continual state representation learning.

Reward Model Loss We use the reward model that predicts the reward $\hat{r}_t$ from the state s_t. The reward model is an NN that approximates the reward prediction function parametrized by the network parameters Θ. The NN consists of one input layer with 128 neurons, three hidden layers with 256 neurons and an output layer with one neuron. We define the reward model loss

$$\mathcal{L}_{Reward} = \|\hat{r}_t - r_t\|_1 \tag{4.6}$$

as the absolute distance between the predicted and the actual reward. The input to the reward model is the latent state representation at some timestep and outputs the expected reward for that timestep. We pass the reward model loss through the reward model to the encoder model to update the encoder parameters.

4.2.4 State Representation Learning Loss

We use the combination of the robotic priors, inverse model and reward model to train the SRL encoder on the total loss

$$\mathcal{L}_{SRL} = \omega_{Priors} * \mathcal{L}_{Priors} + \omega_{Inverse} * \mathcal{L}_{Inverse} + \omega_{Reward} * \mathcal{L}_{Reward} \tag{4.7}$$

where the $\mathcal{L}_{loss}$ is the respective loss term and ω_{loss} is a weighting factor. Through hyperparameter search we found the best set of ω_{loss} to be: $\omega_{Priors} = 1$, $\omega_{Inverse} = 10$, $\omega_{Reward} = 5$. A detailed overview over the training hyperparameters is displayed in Table 4.3.

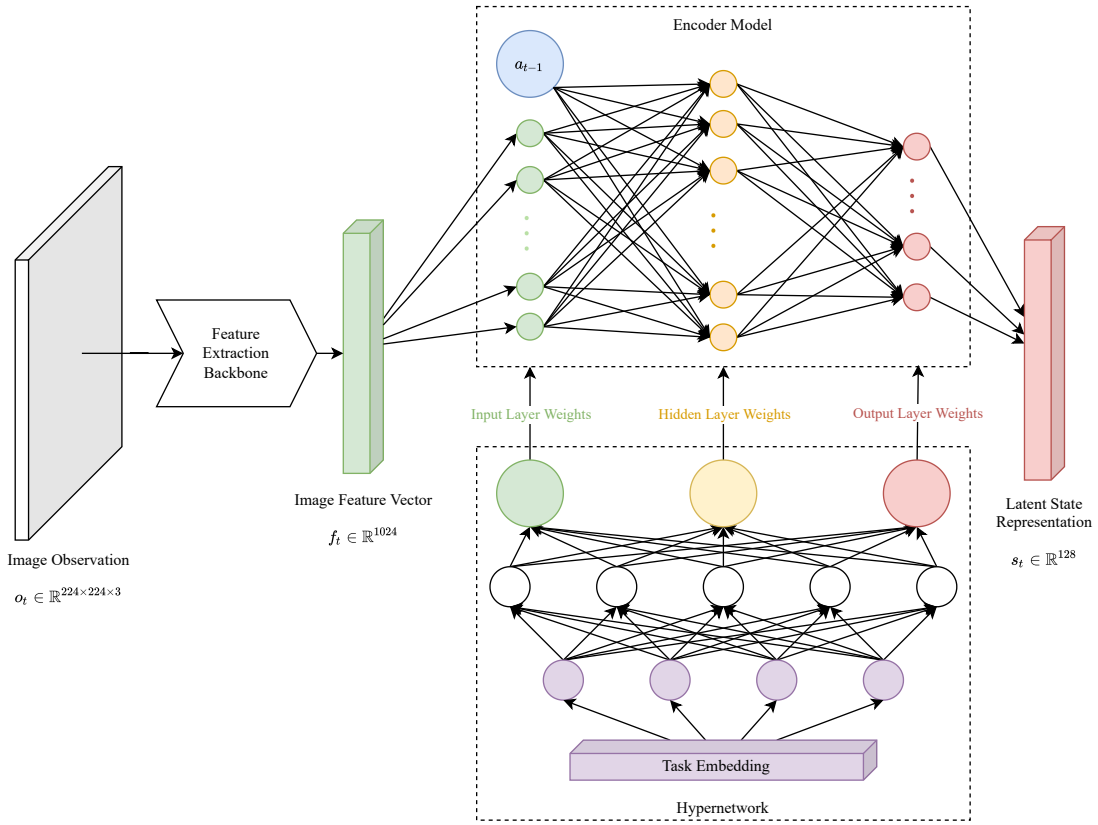


Figure 4.5: The architecture of the SRL encoder model and hypernetwork. Feature extraction and generation of the latent state representation works equivalently to the non-continual SRL encoder model displayed in Figure 4.4. The hypernetwork generates the layer weights of the encoder model based on the task embeddings.

4.2.5 Hypernetwork Architecture

The detailed description of the training of our SRL model includes the training of a model for a single task. In this thesis, we are interested in continual SRL, as we want to enable the learning agent to maintain meaningful state representations over a sequence of tasks. We thus propose a novel hypernetwork-based approach to the CL problem.

As described in subsection 4.2.1 the SRL encoder f_Θ is a dense NN with a single hidden layer and parameters Θ. This architecture can be generated by a hypernetwork following the ideas in subsection 2.3.5. Instead of maintaining the network weights Θ of the encoder model directly in memory, we maintain a hypernetwork $f_h(e_T, \Theta_h)$ that learns to generate the encoder weights Θ based on the task embedding e and hypernetwork parameters Θ_h [Osw+19].

The hypernetwork we use in this thesis is itself an NN. The hypernetwork consists of an input layer of 32 neurons that represent the learnable task embeddings $\{e\}$, two

Parameter	Value
Activation function	ReLU
Bias (Neuron)	True
Task embeddings	$e_T \in \mathbb{R}^{32}$
Input layer	32
Hidden layers	256×256
Output layer	296.576
Optimizer (Θ_h)	Adam
Optimizer ($\{e\}$)	Adam
Initial learning rates	0.003
Learning rate schedule	Step Decay [WMJ21]
Step Decay: Step size	5
Step Decay: Gamma	0.25
Gradient clipping (upper limit)	5

Table 4.4: The common training parameters of the hypernetworks used in continual state representation learning.

hidden layers with 256 neurons, and an output layer of 296.576 neurons that represent the parameters Θ of the encoder. The task embeddings $\{e\}$ are a set of learnable embedding vectors $e_T \in \mathbb{R}^{32}$ with each embedding representing one task realization. Figure 4.5 displays an exemplary hypernetwork architecture for SRL.

4.2.6 Hypernetwork Training

We follow the learning scheme proposed in [Osw+19] and detailed described in subsection 2.3.5. When the hypernetwork is presented a new task T, it generates a new task embedding $e_T \in \mathbb{R}^{32}$ with $e_i \sim \mathcal{N}(0, 1)$ drawn from the normal distribution with zero mean and unit variance. Another approach to generating the task embeddings is to employ another task detection model that outputs task embeddings. We discussed implementing such a model as this enables the setup to operate in a fully task-agnostic manner. Nonetheless, as we are using the reward in the robotic priors as well as the reward model, we already tie the state representation to the specific task. We thus decide to leave a task-agnostic approach for future research as it would exceed the scope of this thesis.

We train the hypernetwork based on the set of task embeddings to solve the SRL task by minimizing the state representation loss $\mathcal{L}_{SRL}$ (subsection 4.2.4). Instead of directly training the encoder f_Θ with weights Θ, the hypernetwork $f_h(e_T, \Theta_h)$ learns to generate the Θ that minimize $\mathcal{L}_{SRL}$. By backpropagation of the error $\mathcal{L}_{SRL}$ through the encoder weights Θ we update the hypernetworks weights Θ_h to solve the SRL problem.
The SRL problem formulation thus changes to

$$\mathcal{L}_{SRL} := \mathcal{L}_{SRL}(\Theta, X_T, Y_T) = \mathcal{L}_{SRL}(\Theta_h, e_T, X_T, Y_T) \tag{4.8}$$

where the encoder weights Θ are generated by the hypernetwork $f_h(e_T, \Theta_h)$.

Training the SRL model on the loss $\mathcal{L}_{SRL}$ itself does not solve the continual learning problem, as it does not include mechanism to maintain the knowledge about previous tasks. Upon obtaining a new task T, the hypernetwork would override the knowledge about previous task $t < T$ stored in its parameters Θ_h. As a result of [Osw+19] we protect the hypernetwork from catastrophically forgetting the previous tasks using the task embeddings of the previous tasks $e_{t<T}$ and a simple output regularizer [Osw+19].

We propose the novel continual SRL objective

$$\mathcal{L}_{CSRL} = \mathcal{L}_{SRL}(\Theta_h, e_T, X_T, Y_T) + \mathcal{L}_{output}(\Theta_h^*, \Theta_h, \Delta\Theta_h, \{e_t\}) \tag{4.9}$$

$$= \mathcal{L}_{SRL}(\Theta_h, e_T, X_T, Y_T) + \frac{\beta_{output}}{T-1} \sum_{t=1}^{T-1} \left\| f_h(e_t, \Theta_h^*) - f_h(e_t, \Theta_h + \Delta\Theta_h) \right\|_2^2 \tag{4.10}$$

where $\mathcal{L}_{SRL}(\Theta_h, e_T, X_T, Y_T)$ is the known SRL task loss and $\mathcal{L}_{output}(\Theta_h^*, \Theta_h, \Delta\Theta_h, \{e_t\})$ is the output regularizer [Osw+19], and Θ_h^* are the hypernetwork parameters before learning the task T. Optimizing the hypernetwork thus becomes a two-step optimization procedure:

1. We compute a candidate change $\Delta\Theta_h$ that minimizes $\mathcal{L}_{SRL}$ using the ADAM optimizer [KB14], without applying the change to the parameters Θ_h. The change $\Delta\Theta_h$ is then considered fixed for the current optimization step.

2. Compute the $\mathcal{L}_{CSRL}$ and the change in the hypernetwork parameters using the output regularizer $\frac{\beta_{output}}{T-1} \sum_{t=1}^{T-1} \left\| f_h(e_t, \Theta_h^*) - f_h(e_t, \Theta_h + \Delta\Theta_h) \right\|_2^2$ that fixes the hypernetwork output for the previous tasks e_t. As the regularization strength we found $\beta_{output} = 1$ through hyperparameter search.

Using this formulation we jointly learn the set of task embeddings $\{e\}$ and the hypernetwork parameters Θ_h. The training scheme enables us to perform continual SRL without the need for separate SRL models per task. Instead of storing the encoder network parameters Θ per task, we only store the hypernetwork parameters Θ_h once and the set of task embeddings $\{e\}$.

4.3 Continual Reinforcement Learning

In this thesis we aim to solve the continual RL problem, avoiding catastrophic forgetting while maintaining enough plasticity to solve a wide variety of tasks. To solve the two tasks presented in subsection 4.1.4, we train PPO agents [Sch+17] using the described reward functions introduced in subsection 4.1.4. The PPO agents base upon the implementations contained in the Stable-Baselines3 repository [Raf+21]. The agents learn to solve the tasks while acting in the reaching and pushing environments described in subsection 4.1.4.

The PPO agent uses the actor-critic formulation [KT99; Lee05; Sch+17] described in section 4.3. We approximate the value and policy functions by an NN that consists of shared base layers that split up into an actor and critic head. Through hyperparameter search we found that a shared network architecture works best in our environment. The networks share the first two hidden layers with 256 neurons each, then split up in separate policy and value function approximation layers. The actor head — the policy approximation layers — takes the output of the shared layers, passes it through two hidden layers with 64 neurons each and output an action prediction $a_t \in [-1, 1]^3$. The critic head — the value approximation layers — takes the output of the shared layers, passes it through two hidden layers with 64 neurons each and output a state value prediction $v_t \in \mathbb{R}$.

The architecture of the actor-critic network is the same among tasks. Having the same architecture over tasks ensures that the number and shapes of the network parameters are equal and that a hypernetwork can generate all task networks. Training the PPO agents on different tasks thus results in equally shaped networks that differ only in the values of the parameters.

4.3.1 Training of Model-Free Reinforcement Learning Agents

To answer the research questions (section 1.2) we train multiple different PPO agents:

1. Numeric agent: PPO agent trained on the normalized numeric state $s \in [-1, 1]^{32}$.
2. Feature-based agent: PPO agent trained on the image feature vectors $f \in \mathbb{R}^{1024}$ extracted by the feature extraction backbone.
3. SRL agent: PPO agent trained on the latent state representation $s \in \mathbb{R}^{128}$ encoded by the SRL encoder model.
4. CSRL agent: PPO agent trained on the latent state representation $s \in \mathbb{R}^{128}$ encoded by the continual SRL encoder model.

We train each of the four agents on each set of reaching and pushing tasks, resulting in eight distinct models for each task and feature space combination. The architecture of the different agents is displayed in Figure 4.6.

We train the PPO agent using the simulator described in subsection 4.1.1. We use 100 simulation instances that act in the environment for 250 timesteps per episode. A

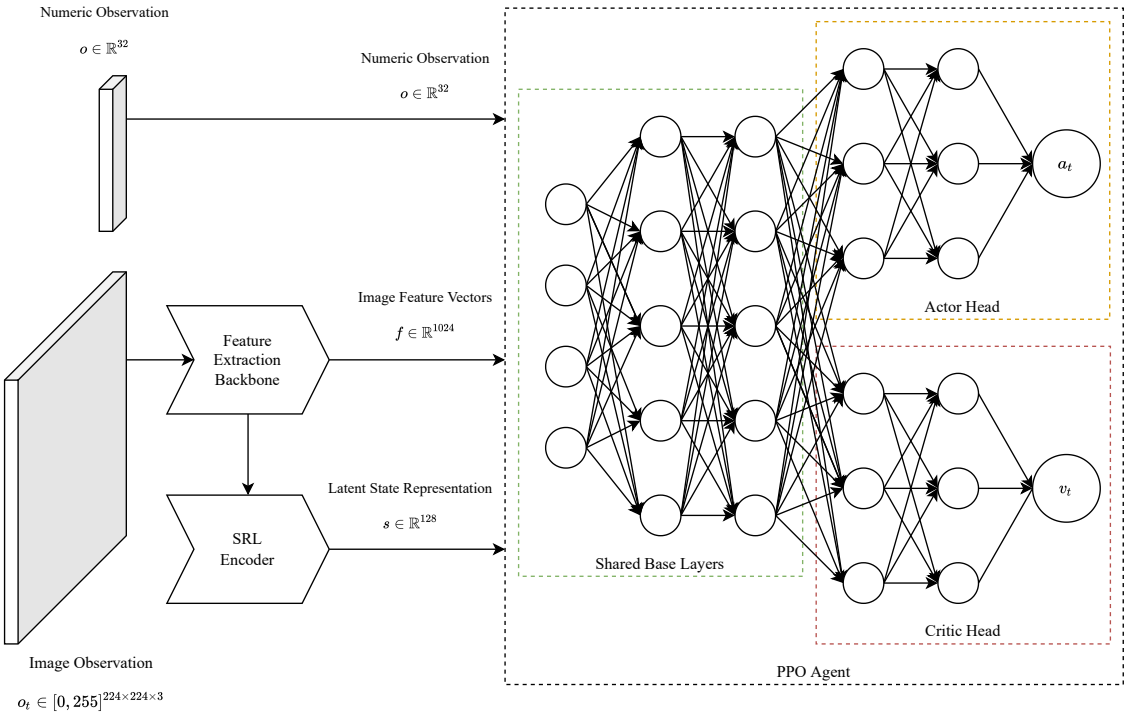


Figure 4.6: The actor-critic network architecture of our different PPO agents. The PPO agent receives as input either the numeric observation o, the image feature vectors f or the latent state representation s generated by a (continual) SRL encoder model. The actor-critic network consists of some shared base layers, an actor head for policy estimation, and a critic head for value function approximation.

shared PPO agent instance predicts the actions to take for all simulations, and the agent trains on the accumulated observations received from all simulations. The agent collects observations for 40 episodes per simulation, resulting in a total of 1.000.000 observations per task. We update the agent after each episode with a batch size of 250 state-action pairs for 10 epochs, resulting in 40 updates until the 1M total timesteps are reached. The agents start with an initial learning rate of 0.001 and an exponential decay [CZA17] using a decay factor of 5. Table 4.5 displays an extensive list of the hyperparameters we use during training of the PPO agents.

4.3.2 Hypernetwork Architecture

The above-mentioned training of the two PPO agents per task for each of the four feature spaces is subject to catastrophic forgetting. The PPO agents learn to solve single tasks, that differ in the input to the actor-critic network as well as the joints that are controllable per task as described in subsection 4.1.4. In this thesis, we are interested in continual RL,

Parameter	Value
Batch size	250
Epochs	10
N Steps	250
Activation function	Tanh
Input layer	Observation dependent
Shared base layers	256×256
Policy head layers	$64 \times 64 \times 3$
Value head layers	$64 \times 64 \times 1$
Optimizer	Adam
Initial learning rates	0.001
Learning rate schedule	Exponential Decay [CZA17]
Exponential Decay: Factor	5
Gradient clipping (upper limit)	0.5

Table 4.5: The common training parameters of the agents used in non-continual reinforcement learning.

as we want to enable a single RL agent to solve both tasks sequentially. We require that the agent does not have access to previous tasks when learning a new one, i.e. it can not go back to the old task and collect new state-action pairs to retain knowledge. We thus propose a novel hypernetwork-based approach to the continual model-free RL problem.

As described in section 4.3, the actor-critic network in the PPO agent is a dense NN parametrized by the network weights Θ. This architecture can be generated by a hypernetwork following the ideas in subsection 2.3.5. The hypernetwork we use to generate the actor-critic network in this thesis is itself an NN. The hypernetwork consists of an input layer of 32 neurons that represent the learnable task embeddings $\{e\}$, two hidden layers with 256 neurons, and an output layer that represent the parameters Θ of the actor-critic network. The size of the output layer is dependent on the input state type of the agent, as the first layer of the agent needs to handle either the numeric observations, the image feature vectors or the latent state embeddings extracted by the SRL encoder. For the agent trained on numeric observations the output layer consists of 115.719 neurons, for the agent trained on image feature vectors of 369.671 neurons, and for the agent trained on the latent states of 140.295 neurons. The task embeddings $\{e\}$ are learnable embedding vectors $e_T \in \mathbb{R}^{32}$ representing the task realizations. Figure 4.7 displays an exemplary hypernetwork architecture for continual model-free RL.

4.3.3 Hypernetwork Training

Opposed to the training of the hypernetwork-based SRL encoder, we do not follow the learning scheme proposed in [Osw+19] and detailed described in subsection 2.3.5. Instead, we propose a learning scheme that is compatible with the model-free RL agents

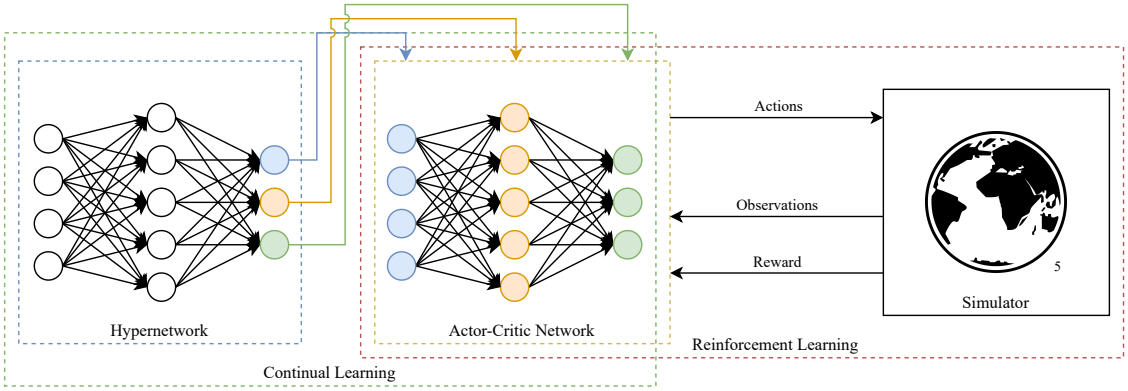


Figure 4.7: Our novel hypernetwork-based approach to CRL. The hypernetwork learns to generate the actor-critic network weights of the PPO agent. The PPO agent learns to solve the tasks in simulation as usual.

[5] Source: https://upload.wikimedia.org/wikipedia/commons/thumb/6/60/Simple_Globe.svg/1024px-Simple_Globe.svg.png

implemented in the Stable-Baselines3 [Raf+21] repository.

We propose training a hypernetwork-based PPO agent in a two-step process. We start by training an PPO agent on the first task — the reaching task — using the PPO surrogate loss

$$\mathcal{L}_{PPO} := \mathcal{L}_{clip}(\Theta) = \hat{\mathbb{E}}_t[\min(r_t(\Theta)\hat{A}_t, clip(r_t(\Theta)), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{4.11}$$

where $\mathcal{L}_{PPO}$ is the loss to minimize in order to solve the model-free RL problem.

We train this agent until we are satisfied with its performance in solving the task — 1.000.000 timesteps in the environment for each task. Training the PPO agent in this known manner results in a set of actor-critic network parameters Θ_T that approximate the optimal policy and value function to solve the task. After solving the first task T, we extract the network parameters Θ_T. We generate a new task embedding $e_T \in \mathbb{R}^{32}$ with $e_i \sim \mathcal{N}(0,1)$ drawn from the normal distribution with zero mean and unit variance and add it to the set of task embeddings $\{e\}$.

We then train the hypernetwork f_h to generate the task weights Θ_T based on the task embedding e_T and hypernetwork parameters Θ_h using our novel loss

$$\mathcal{L}_{CL_{task}}(e_T, \Theta_h, \Theta_T) := \sum_{l=1}^{|L|} \left\| f_h(e_T, \Theta_h)_l - \Theta_{T_l} \right\|_H \tag{4.12}$$

calculating the Huber loss $\left\| f_h(e_T, \Theta_h)_l - \Theta_{T_l} \right\|_H$ between each pair of generated actor-critic network layers $f_h(e_T, \Theta_h)_l$ and the actual layers Θ_{T_l} for each of the layers $l \in L$. The resulting parameters Θ_h of the hypernetwork and the task embedding e_T can now be used to generate the actor-critic network weights Θ_T for the task T.

Training solely on the loss $\mathcal{L}_{CL_{task}}$ does not solve the continual learning problem, as it does not include a mechanism to maintain the knowledge about previous tasks. Upon

Parameter	Value
Activation function	ReLU
Bias (Neuron)	True
Task embeddings	$e_T \in \mathbb{R}^{32}$
Input layer	32
Hidden layers	256×256
Output layer	Agent dependent
Optimizer (Θ_h)	Adam
Optimizer ($\{e\}$)	Adam
Initial learning rates	0.003
Learning rate schedule	Step Decay [WMJ21]
Step Decay: Step size	5
Step Decay: Gamma	0.25
Gradient clipping (upper limit)	5

Table 4.6: The common training parameters of the hypernetworks used in continual reinforcement learning.

obtaining a new task T, the hypernetwork would override the knowledge about previous task $t < T$ stored in its parameters Θ_h.

We thus propose the novel learning scheme to train the hypernetwork on sequential tasks. After learning to generate the actor-critic network weights for the previous $T-1$ tasks, we train a new PPO agent on the current task T. In our setting, the second task is the pushing task. We train the hypernetwork to learn the subsequent tasks by optimizing the task loss $\mathcal{L}_{CL_{task}}$ for the second task, while maintaining the knowledge about previous tasks using our novel output regularizer $\mathcal{L}_{output}$.

We define the novel continual RL learning scheme

$$\begin{aligned}
\mathcal{L}_{CL}(\{e\}, \Theta_h, \Theta_T) :=& \mathcal{L}_{CL_{task}}(e_T, \Theta_h, \Theta_T) + \mathcal{L}_{output}(e_{t<T}, \Theta_h, \Theta_h^*) \\
=& \mathcal{L}_{CL_{task}}(e_T, \Theta_h, \Theta_T) + \sum_{t=1}^{T-1} \mathcal{L}_{CL_{task}}(e_t, \Theta_h, f_h(e_t, \Theta_h^*)) \\
=& \mathcal{L}_{CL_{task}}(e_T, \Theta_h, \Theta_T) + \sum_{t=1}^{T-1} \mathcal{L}_{CL_{task}}(e_t, \Theta_h, \Theta_t)
\end{aligned} \tag{4.13}$$

where $\mathcal{L}_{CL_{task}}(e_T, \Theta_h, \Theta_T)$ is the known loss of the hypernetwork to learn to generate the task specific actor-critic network weights Θ_T of the current task. We use the hypernetwork parameters Θ_h^* before learning the new task and the task embeddings of the previous tasks $e_{t<T}$ to regenerate the previous task network weights $\Theta_{t<T}$. We use the regenerated task embeddings $\Theta_{t<T}$ to regularize the hypernetwork so that the output does not deviate from the original previous task network weights. This regularization ensures that the hypernetwork produces the actor-critic network parameters $\Theta_{t<T}$ for the previous tasks $e_{t<T}$ while learning to produce the current task weights Θ_T based on the new task

embedding e_T. Table 4.6 displays an extensive list of the hyperparameters we use during training of the hypernetwork.

By generating the target network weights $\Theta_{t<T}$ using the hypernetwork $f_h(e_{t<T}, \Theta_h^*)$ itself we only store knowledge about previous tasks within the hypernetwork. This approach does not store any actor-critic network weights of previous tasks and needs no access to previous data. Instead, the hypernetwork regenerates its targets only based on the learned task embeddings $\{e\}$. After training the hypernetwork, we only store the hypernetwork parameters Θ_h and the set of learned task embeddings $\{e\}$. We show in the following chapter that our novel approach completely prevents catastrophic forgetting and opens the field of model-free continual RL with hypernetworks.

5 Evaluation

In this thesis we propose novel hypernetwork-based approaches to SRL and model-free RL. We evaluate our approaches by answering the research questions (section 1.2):

5.1 Comparison of Numeric and Vision-Based Agents

Research questions one and two boil down to a comparison between four different types of RL agents:

1. An agent trained on the numeric observations. This agent uses the numeric observations $s \in \mathbb{R}^{32}$ that include the end effector position and direction vector, the joint angles and velocities, the target position and direction vector as well as the object position and direction vector (Table 4.1). We measure the values in the simulations and use these definitive numeric values as the ground truth of observations.

2. An agent trained on the image feature vectors. This agent uses the image feature vectors $f \in \mathbb{R}^{1024}$ that are generated by the DenseNet we use as the feature extraction backbone (subsection 2.2.1). These image feature vectors contain all information that is extractable via the DenseNet feature extraction backbone, but in an unstructured and possibly redundant format.

3. An agent trained on the output of a non-continual SRL oracle. We train two SRL oracles separately on the datasets described in subsection 4.2.2 for the two tasks of reaching and pushing. The SRL-based agent uses the latent state embedding $s \in \mathbb{R}^{128}$ extracted by the SRL encoder model from the image feature vectors. This state representation contains all information that is extractable via the DenseNet, possibly structured and condensed using the robotic priors (subsection 2.2.3), the inverse model (subsubsection 2.2.4) and the reward model (subsubsection 2.2.4).

4. An agent trained on the output of the continual SRL model. We train the CSRL model continually on the same datasets as the SRL oracles, using our novel hypernetwork-based approach described in section 4.2. The CSRL-based agent uses the latent state embedding $s \in \mathbb{R}^{128}$ extracted by the CSRL encoder model from the image feature vectors.

We use the PPO actor-critic architecture described in section 4.3. We train each of the four listed agents separately on the two tasks — starting with the reaching and continuing

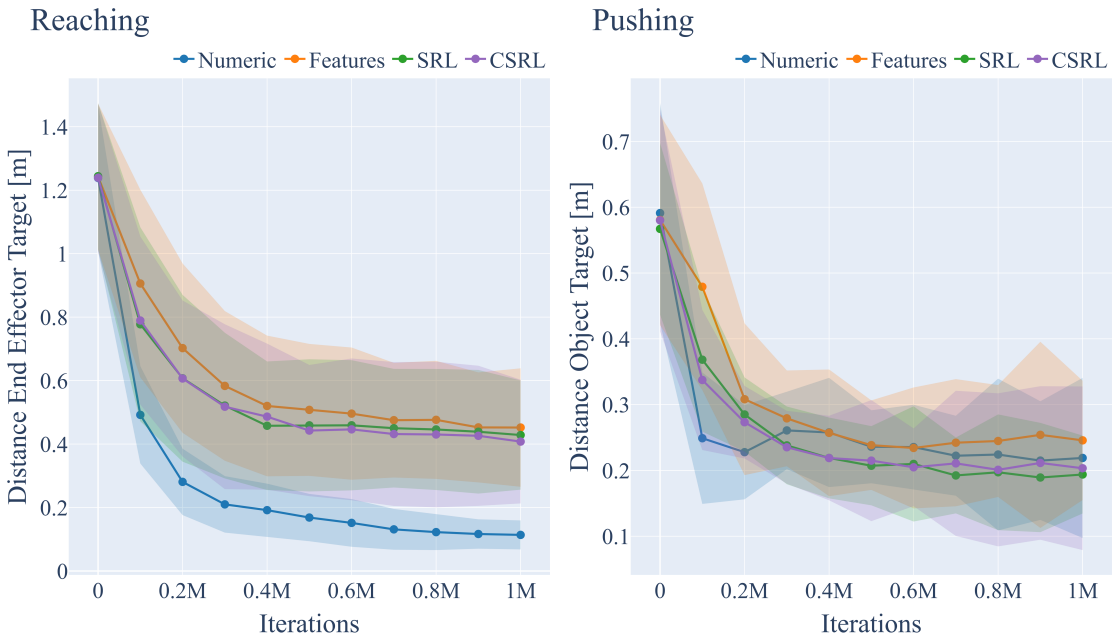


Figure 5.1: The evaluation metrics of the reaching (left) and pushing (right) tasks. We evaluate 4 agents with different input observations, namely numeric state, image features, state encoded by the SRL model and the CSRL model. For each agent we run 100 episodes with 250 steps each. We plot the mean (lines) and one standard deviation (shaded regions) over 5 seeds. — Left: The distance of the end effector to the target position in meters. Lower is better, as the robot tries to touch the target. — Right: The distance of the object position to the target position in meters. Lower is better, as the robot tries to push the object upon the target.

with the pushing task — for 1.000.000 timesteps in the environment. During training, we generate a snapshot of each agent after 100.000 timesteps, resulting in 11 snapshots of the agents per setting. To prevent a dependency of a specific input seed, we train five independent agents per setting with five random but fixed initial seeds. All agents use the same architecture described in section 4.3, except for the input layer adjusted to represent the input observation types.

We use the task reward functions presented in subsection 4.1.4 for each of the agents. This enables us to compare the performance of the agents independent of the input observation spaces. We evaluate the performance of each snapshot of the five agents per setting on the same environment it was trained on but with a fixed initial seed for the evaluation environment. We calculate the mean and the standard deviation over the five seeds, which gives us an estimate of the general performance and robustness of our approach. Figure 5.1 display plots of the evaluation metrics of the agents per task.

5.1.1 Performance of Individual Agents

How well does a RL agent trained on images perform compared to an agent trained on hand-crafted numeric states? Can the vision-based agent solve the given tasks as well as the numeric-based agent?

Reaching The goal of the robot in the reaching task is to bring the tip of its end effector as close as possible to a target as described in subsection 4.1.4. The main objective of the agents is to minimize the euclidean distance of the end effector position and the target position.

Figure 5.1 shows that all agents improve their performance drastically trained on either of the inputs. As expected, the agent trained on the numeric observations rapidly lowers the mean distance between its end effector and the target. This agent utilizes the real three-dimensional positions of its end effector and of the target and has direct access to its measured joint angles as well as the joint velocities. These values include all the information the agent needs to navigate its end effector precisely. The vision-based agents also rapidly improve their performance, but do not reach a remaining distance as low as the numeric agent. Additionally, the standard deviation of the vision-based agents is much higher compared to the numeric agents. We observe that for some target locations the vision-based agents fail to get close to the target.

We propose a three-fold explanation for this behavior:

1. The required information to predict the end effector and target positions is not well embedded in the image feature vectors extracted by the DenseNet. The feature extraction backbone is pretrained on the ImageNet [Rus+14] dataset. The dataset consists of real photos annotated for classification tasks. Our image observations result from the simulation, thus the images are virtually generated. The shift in the distributions between the real photos and the generated observation image results in a loss of relevant information useful for the robot.

2. We use a singular camera setup with the camera mounted 3 meters above the robots base. The target spawns at a height of 0.5 meters above the ground and at a distance between 0.5 to 1 meter away from the origin at the robots base. Monocular depth estimation is still an open field of research [Bho19] and the SRL module needs to learn to estimate the distance between the camera and the target while encoding all other relevant properties. In section 5.2 we show that especially the height can not be predicted as precisely as the other two dimensions of the target position. Perspective distortion further amplifies the problem of the monocular system approach.

3. We observe that the robot moves slower to the targets position in the vision-based setups compared to the numeric setup. The robot takes more steps in the environment with its end effector being further away from the target position. The slow movement of the robot increases the mean distance over the whole episode drastically.

Pushing The goal of the robot in the pushing task is to bring the green box as close as possible to a target as described in subsection 4.1.4. The main objective of the agents is to minimize the euclidean distance of the position of the box and the target position.

Figure 5.1 shows that all agents improve their performance drastically trained on either of the input feature spaces. In contrast to the reaching tasks, the vision-based agents perform on par with the numeric agent. All agents can reliably move the object within a $0.2 \sim 0.25$ centimeter range to the target averaged over the 250 frames per episode. Surprisingly, the agents trained on the latent state representations extracted by the SRL and CSRL encoder outperform the numeric and feature-based agents.

We explain these observations based on the properties of the approach:

1. The numeric agent performs not as well as the agents trained on the latent state representations. Although the numeric state includes the real position and direction vector of the object and target, as well as the robot configuration, the numeric state might not include enough of the relevant information the robot needs to solve the pushing task. The numeric agent especially has to detect patterns in the state that encode the distance between the target and the object from the absolute positions.

2. We place the camera 3 meters above the base of the robot facing vertically downwards. In this setup the camera sees the floor, the target and object, as well as the robot top down. All the components can only move in the two-dimensional plane parallel to the floor so that there is no need to include a depth estimation as in the reaching task. The task of minimizing the distance between the object and the target thus equals to minimizing the pixel distance between the two.

3. The agent can only move with its upper part nearly parallel to the floor. This relaxation of the pushing task enables the robot to focus more on the relevant dimensions of the input spaces. As the agent does not need to take the height of objects into account, it can focus more on solving the two-dimensional problem in the three-dimensional space instead of estimating the height component of the target and object.

Discussion We show that the vision-based agents are all capable of solving the given tasks. We see that especially for the pushing task, the performance of the vision-based agents is on par with the numeric agent.

For the reaching tasks there is the problem of monocular depth estimation. The vision-based agent would clearly benefit from an additional model extracting a depth map of the current image observation. This depth map can be fed to the SRL and CSRL encoders to include the depth information in the latent state embedding to further improve performance.

Additionally, we use the DenseNet as a feature extraction backbone with fixed pretrained weights. The ImageNet dataset [Rus+14] it was pretrained on might not be of the same distribution as the image observations from the simulator. The extraction of the relevant image features is thus not tailored towards our tasks. The vision-based approaches will

Agent	Agent Performance	Numeric 0.114	Features 0.452	SRL 0.429	CSRL 0.408
Numeric	0.114	100%	396.5%	376.3%	357.9%
Features	0.452	**25.2%**	100%	94.9%	90.3%
SRL	0.429	**26.6%**	**105.4%**	100%	95.1%
CSRL	0.408	**27.9%**	**110.8%**	**105.1%**	100%

Table 5.1: Ratio of performance of the agents on the reaching task. The cells display the ratio of the agent in the row to the agent in the column. — E.g. the CSRL agent reaches a performance of 27.9% compared to the numeric agent, whereas it reaches a performance of 110.8% and 105.1% compared to the feature and SRL-based agents.

benefit of a fine-tuning of the DenseNet to our domain. The DenseNet can be fine-tuned in an unsupervised manner jointly with the SRL module. The fine-tuned model can be protected by hypernetworks equivalent to the CSRL encoder to produce a complete end-to-end approach to vision-based CRL for robotic manipulation tasks.

5.1.2 Difference of Image Feature Vectors and Latent State Embeddings

How well does an agent trained on image feature vectors perform compared to an agent trained upon the latent state embedding of a SRL encoder model? Is there a benefit in using the latent state embedding over the image feature vectors?

Figure 5.1 shows the performance of the four described agents on the two tasks. In both plots it is visible that there is a margin by which the agents trained on the latent state embeddings perform better than the agent trained on the image feature vectors.

Reaching Table 5.1 displays the final performance of the agents on the reaching task. We see in the first column that the vision-based agents perform worse than the numeric agent by a large margin. Regarding the vision-based agents, we observe that the agent based solely on image feature vectors performs worse that the agents trained on the latent state embedding. We observe that the agents trained on the state embedding from the SRL encoder perform 5.4% better than the agent trained on the image feature vectors. The agents trained on the CSRL state embedding perform 10.8% better than the feature-based agents and 5.1% better than the embeddings from the SRL encoder. Additionally, Figure 5.1 displays that the performance of the SRL and CSRL agents increases more quickly compared to the feature-based agent.

Pushing Table 5.2 displays the final performance of the agents on the pushing task. We see in the first column that only the feature-based agent performs worse than the numeric agent by 11.0%. The agents trained on the state representation extracted by the SRL and CSRL encoders outperform the numeric agent by 11.2% and 7.9%. The agents

Agent	Agent Performance	Numeric 0.219	Features 0.246	SRL 0.194	CSRL 0.203
Numeric	0.219	100%	112.3%	90.0%	92.7%
Features	0.246	**89.0%**	100%	78.9%	82.5%
SRL	0.194	**111.2%**	**126.8%**	100%	104.6%
CSRL	0.203	**107.9%**	**121.2%**	**95.6%**	100%

Table 5.2: Ratio of performance of the agents on the pushing task. The cells display the ratio of the agent in the row to the agent in the column. — E.g. the CSRL agent reaches a performance of 107.9% compared to the numeric agent, whereas it reaches a performance of 121.2% and 95.6% compared to the feature and SRL-based agents.

trained on state embeddings perform the best in the vision-based setting with a 26.8% increase in performance when using the SRL encoder and 21.2% using the CSRL encoder compared to the agents trained on the image feature vectors. The agents trained on the state extracted from the CSRL model perform 4.4% worse than the SRL-based agents.

Discussion We observe that the performance of the RL agents trained on the latent state embeddings extracted by either of the SRL or CSRL encoders perform strictly better than the agents trained on the image feature vectors.

We propose that the image feature vectors are not structured meaningfully for RL. The image feature vectors are 1024 dimensional and only structured regarding the classification task of the ImageNet [Rus+14] challenge the DenseNet feature extraction backbone was trained on. This structure is not optimal for RL, and the agents use much of their capacity to detect meaningful patterns within the unstructured image feature vectors.

The latent state embeddings are only 128 dimensional, compared to the 1024 dimensional feature vectors. We train the encoders regarding the physical and temporal properties of the environment using the robotic priors. Additionally, the inverse model structures the latent space so that successive states are related regarding the action responsible for the state transition, in conjunction with the reward network that embeds knowledge about the state-reward relation. These structural, physical and temporal priors shape the latent state embedding space meaningfully for the subsequent RL tasks. The agents solve the tasks in the same physical and causal domain in which the latent states are embedded, and the imposed structure frees capacity in the RL networks to focus on the task.

We additionally propose that a positive backward transfer happens in the CSRL module. We observe this transfer as the performance of the agents trained on the embeddings from the CSRL module perform 5.1% better on the reaching tasks compared to the SRL module. The training of the CSRL module continually on both tasks transfers knowledge from the pushing task back onto the reaching task, where the performance of the RL agent benefits from.

We also observe that we could not achieve positive forward transfer, as the performance

of the agent trained on the SRL embedding are 4.4% better compared to the CSRL embeddings. We propose that there is no positive transfer due to the regularization of the hypernetwork. Our current hyperparameter choice for the encoder generating hypernetwork might not be optimal to solve the pushing task in succession to the reaching task. Nonetheless, we propose that by further hyperparameter tuning — especially of the regularization strength parameter β_{output} (Equation 4.10) — positive forward transfer can be achieved.

5.2 Comparison of Continual and Non-Continual State Representation Learning

How well can a continual SRL model encode relevant information into the latent state representation when trained on sequential tasks compared to a non-continual SRL module trained on the tasks separately? How well can the continual SRL model avoid catastrophic forgetting?

We evaluate the CL capabilities of our novel continual SRL approach by a comparison of three different models:

1. The oracle SRL model consists of two separate models trained separately on the two tasks. This model does not include CL as a protection from forgetting. This model is not prone to forgetting, as the separate models do not interfere with each other.

2. The forgetting SRL model consists of one single model trained sequentially on the two tasks. This model does not include CL as a protection from forgetting. This model is prone to forgetting, as during training on the second task the knowledge about the first task is not protected and thus may be overridden.

3. The continual SRL model consists of one single hypernetwork-based model trained sequentially on the two tasks. This hypernetwork-based model does include the regularization-based protection from forgetting detailed in section 4.2. This model may be prone to forgetting, but the hypernetwork regularizer protects the previous task knowledge during training on the second task.

We train each of the above-mentioned models sequentially on the two tasks, starting with the reaching and continuing with the pushing task. For each type of model we train 5 models with the same initial conditions differing only in the random seed. Each model is trained for $\sim$ 40.000 iterations (50 epochs) and we stop the training on the reaching task after the first $\sim$ 40.000 iterations. The networks do not have access to any data regarding the reaching task after we switch to the pushing task.

We save the model parameters of the oracle SRL model after the reaching task to avoid forgetting completely. The oracle thus provides a baseline for comparison with the other models. The forgetting SRL model uses the same architecture as the oracle model, but we do not save the parameters and thus do not protect it from forgetting. The continual SRL model uses a hypernetwork to generate the same architecture as the

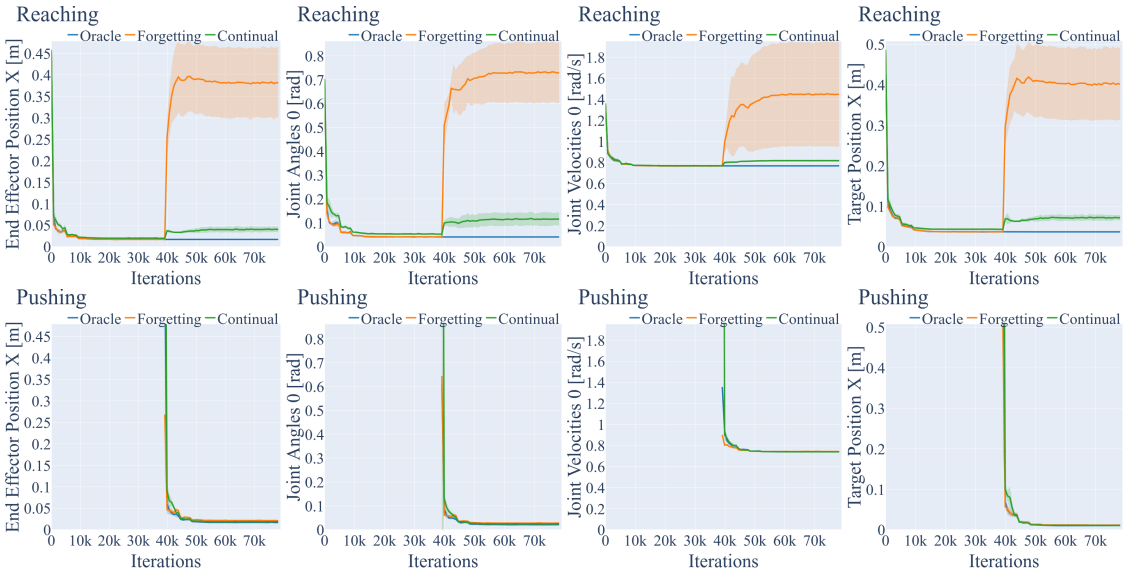


Figure 5.2: Top row: Reconstruction error of the numeric observations based on the latent state embedding of the SRL oracle, the forgetting SRL model, and the continual SRL model evaluated on the reaching task over the whole sequential training over both tasks. — Bottom row: The same entries evaluated on the pushing task when training sequentially on both tasks. — The training on the reaching task starts at iteration 0. The training on the pushing task starts at the end of the reaching task at iteration $\sim 40k$. — Positions are measured in meters, angles in radians, velocities in radians per second.

two above-mentioned networks. Our adaption of the hypernetwork output regularizer described in subsection 4.2.6 protects the hypernetwork from forgetting the first task.

Simultaneously to the training of the SRL models, we train two other NN models to predict the numeric observations from the latent state embeddings. We call these models the ground truth models. We train the ground truth models in a supervised fashion with the latent state embedding as input and with the numeric observations as the target vectors. The ground truth models thus evaluate the amount of the original numeric information that is reconstructable from the state embeddings. We use one separate ground truth model per task to predict the numeric states, whereas we store the model trained on the first task after training to protect it from parameter changes.

This evaluation is to be taken with caution, as there are many factors of variance involved, prohibiting a precise reconstruction of the numeric state. These factors include the lost information after passing the images through the feature extraction backbone or possibly not optimal hyperparameters for either of the SRL or ground truth models. Nonetheless, a qualitative analysis of the CL capabilities of the models is possible regarding this reconstruction metric.

Figure 5.2 displays the remaining error of the reconstruction of a selection of the

entries in the numeric state (We include the error plots of all state entries in chapter 7). We calculate the errors as the absolute difference between the expected numeric value extracted from the simulator and the predicted value from the ground truth model.

During training on the first task (reaching) we observe that either of the SRL models encodes a roughly equal amount of information needed to reconstruct the numeric values from the state embedding. This behavior is as expected, as on the first task none of the models may forget information as there is no interference of task knowledge. The SRL model generated by the hypernetwork also equally well encodes the information, verifying the capability of the hypernetwork to approximate the network weights of the SRL encoder.

At the start of the training on the second task (pushing), the CL capabilities of the hypernetwork become visible. As soon as we start training on the second task at ~ 40.000 iterations, the oracle model keeps a constant performance when evaluated on the first task. This is as expected, as the separate network weights do not change as this model is not trained on the second task.

The forgetting model rapidly decreases in performance as displayed in the first row of Figure 5.2. The training of the model on the second task overrides the network parameters and tailors them towards the second task. The model is not protected from catastrophic forgetting and thus cannot retain the knowledge.

The continual model does only decrease in performance slightly as displayed in the first row of Figure 5.2. The hypernetwork regularizer described in subsection 4.2.6 reliably protects the hypernetwork from overriding the knowledge of the first task. Nonetheless, the second row of Figure 5.2 displays that the hypernetwork does exhibit enough capacity to also learn a state representation equally well for the second task compared to the non-continual models. We expect that the remaining reconstruction error can be fully mitigated with a better set of hyperparameters.

Following the presented observations, we propose that a hypernetwork-based approach for SRL is a viable option for continual SRL. The hypernetwork reliably prevents catastrophic forgetting while being able to achieve comparable performance regarding the reconstruction of the numeric state from the latent state embedding.

Analysis of the Prediction of Numeric Observations In chapter 7 we present the evaluation plots of the reconstruction errors for all 32 entries of the hand-crafted numeric states. The plots show that all SRL encoders reliably encode the position and direction vectors of the end effector, target, and object. The joint angles are predictable within a maximal error of ~ 0.1 radians in either case, although there can be self-occlusion of lower joints of the robot.

The joint velocities are hard to predict by the ground truth model based on the latent state embedding. This implies that the SRL encoder does not embed the velocity information well in the state representation. We propose that the velocity is a time-dependent quantity and that for a reliable prediction multiple successive frames need to be passed to the encoder. As we require that only the observation of the current timestep is input to the

SRL encoder, a reliable prediction is not possible. Nonetheless, the absolute remaining error of $\sim$ 0.8 radians per second corresponds to a relative error of $\sim$ 5% within the $[-10, +10]$ radians per second range of the velocity.

As we do not use the object in the reaching task, the ground truth model reliably predicts the default position and direction vector with zero remaining error. On the pushing task the SRL encoder reliably predicts the object position within a range of $\sim$ 3 cm, but struggles to predict the object direction vector with a remaining error of $\sim$ 0.25. The direction vector is hard to embed by the SRL encoder as the object in the pushing task is a green cube. The cube is symmetrical and has the same green color on all sides and there is no visual difference for the SRL encoder to detect.

5.3 Comparison of Continual and Non-Continual Reinforcement Learning

How well does a continual RL agent trained sequentially on either hand-crafted numeric states or latent state embeddings perform compared to a non-continual RL agent trained separately on each task? How well can the continual RL agent avoid catastrophic forgetting?

In subsection 4.3.3 we present a novel hypernetwork-based approach to continual model-free RL. In this thesis we are especially interested in how well hypernetworks can avoid catastrophic forgetting in model-free RL. In section 5.1 we detailed the training of multiple baseline agents on the reaching and pushing task sequentially. To evaluate the performance of our hypernetwork-based CRL approach we select the baseline agents that performed best over the whole training and over all five seeds. We extract the actor-critic network parameters from each of these models and use them as the targets for a hypernetwork. The hypernetwork learns to generate the target weights for both tasks according to the training scheme we propose in subsection 4.3.3. After the training of the hypernetwork, we use it to regenerate the target weights of the actor-critic networks.

To evaluate the performance of the hypernetwork-based RL agents, we replace the actor-critic network weights with the generated ones and run these agents in the same environment as the baseline agents. We perform this comparison over all four of the input spaces described in section 5.1. For each type of agent we train 5 hypernetworks with the same initial conditions differing only in the random seed. As a reference, we also evaluate a random agent in each of the task environments. The random agent samples actions $a_t \in [-1, 1]^3$ with the entries drawn from the uniform distribution $a_{t,i} \sim \mathcal{U}(-1, 1)$.

Reaching In Figure 5.3 we display the performance of the agents on the reaching task. It shows that the baseline and hypernetwork-based RL agents clearly outperform the random agent by a large margin, implying that any of the approaches is well suited to learn the tasks. The RL agents learn a meaningful policy to solve the tasks and are capable to reach the target with the end effector within a small range. As described in section 5.1, the agents trained on the numeric state perform best, followed by the CSRL-based agents.

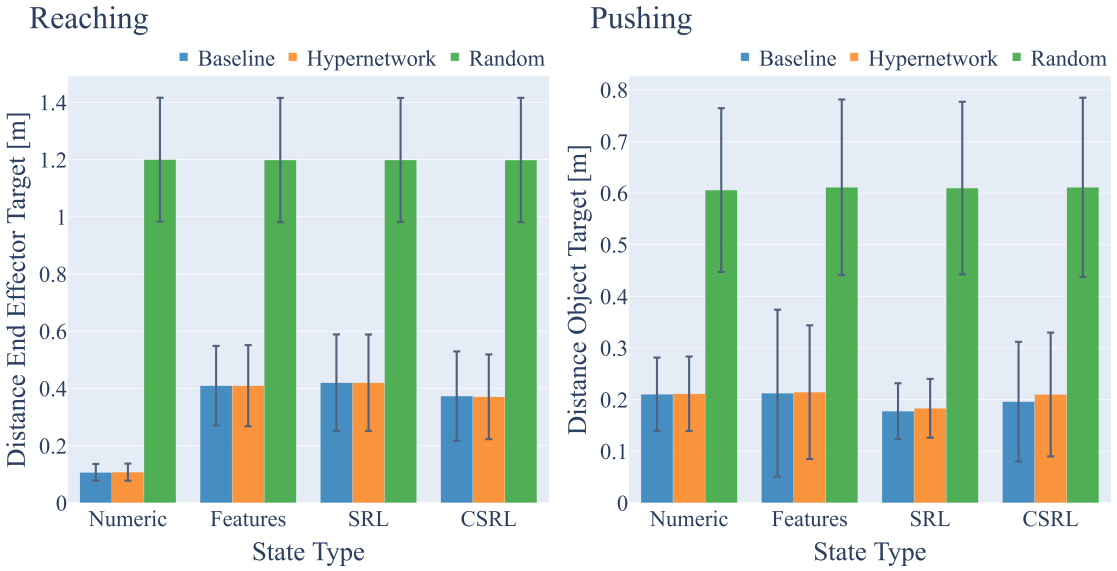


Figure 5.3: Comparison of the evaluation metrics between a baseline RL agent trained on the tasks separately, a hypernetwork-based RL agent trained continually and an agent taking random actions. The agents are grouped by input states used to train the RL agents on. For each agent we run 100 episodes with 250 steps each. We plot the mean (bar) and one standard deviation (error) over 5 training seeds. — Left: The distance of the end effector to the target position in meters. Lower is better, as the robot tries to touch the target. — Right: The distance of the object position to the target position in meters. Lower is better, as the robot tries to push the object upon the target.

Figure 5.3 shows that the performance of the hypernetwork-based agents are on par with the baseline agents. This implies that the hypernetwork is capable of regenerating the RL agents after training on both tasks. We observe that the performance of the hypernetwork-based agents is independent of the input state type.

Pushing In Figure 5.3 we display the performance of the agents on the pushing task. We observe the same significant performance increase in comparison to the random agent, what implies that the baseline and hypernetwork-based agents learn a meaningful policy for the pushing task. As detailed in section 5.1, the agents trained on the image feature vectors perform worse, whereas the agents trained on the state embeddings extracted from the SRL and CSRL models perform better that the numeric agent.

Figure 5.3 shows that the performance of the hypernetwork-based agents are on par with the baseline agents. This implies that the hypernetwork is capable of regenerating the RL agents after training on both tasks. We observe that the performance of the hypernetwork-based agents is independent of the input state type.

Discussion We observe that the performance of the agents generated by our novel hypernetwork-based approach is on par with the baseline agents trained separately on the tasks. In our novel proposed learning scheme, the hypernetwork learns to regenerate the network weights for both tasks, independent of the input space. This is as expected and proposed by [Osw+19], as hypernetworks operate solely in target network space and do not depend on input nor output features. The neglectable differences in the performance between the baseline and hypernetwork-based RL agents implies that model-free RL can be approached using hypernetworks.

Our novel learning scheme relies on a two-step process of first training the PPO agent and then the hypernetwork to generate the agents network weights and is thus not strict online learning. Furthermore, our approach prevents forward and backward transfer, as the hypernetwork does not participate in the training of the PPO agent itself. As the hypernetwork parameters do not gradually update during the training of the RL agent on the task, no new task knowledge can be embedded. Additionally, our robot setup described in subsection 4.1.4 also prevents knowledge transfer, as the network in the reaching task controls a significantly different set of joints compared to the network in the pushing task. Due to this setup we decided to focus on preventing catastrophic forgetting of the RL agents, and we leave extending our approach to a setup possibly allowing knowledge transfer as future work.

Nonetheless, these significantly different robot setups per task also display a core strength of the hypernetwork-based approach. As the hypernetwork operates solely in target network space, it does not depend on any similarity of tasks or robot setups. Even in the case of possibly opposing gradients in the RL agent networks while learn the single tasks, the hypernetwork learns to regenerate the weights reliably. These possibly opposing gradients are a core problem in regularization based approaches, and hypernetworks naturally sidestep the problem by operating only in target network space.

6 Future Work

In science folklore, the merit of a research question is compounded by the number of interesting follow-up research questions it raises.

So to show the merit of the problem you worked on, you list these questions here.

If you don't care about research folklore (I did not as a student), this chapter is still useful: whenever you stumble across something that you should do if you had unlimited time, but cannot do since you don't, you describe it here.

Typical candidates are evaluation on more study objects, investigation of potential threats to validity,

The point here is to inform the reader (and your supervisor) that you were aware of these limitatons.

Limit this chapter to very few pages.

Two is entirely fine, even for a Master's thesis.

6.1 Model-Based Continual RL

6.2 Task-Agnostic SRL

6.2.1 Task Detection Model

6.3 More Complex Tasks

6.4 Use Vectorized Environment for Continual Stream of Data

7 Conclusion

Using neural networks and reinforcement learning for robotic manipulation is still in its infancy and practical applications of such controllers are rare. One major problem that prevents wider application is that such controllers can only perform a very specific task they were trained on and adding new capabilities requires training of the controller from scratch. Another major problem is the need for hand-crafted state representations relevant for the manipulation task in question in order to successively train a controller. Focusing on these two problems, we explore solutions that allow continual learning of tasks from raw image data.

In this thesis we proposed — to the best of our knowledge — for the first time to solve the problem of vision-based continual reinforcement learning for robotic manipulation tasks based on hypernetworks. The hypernetwork is itself a neural network that learns to generate the weights of another target network, conditioned on the task. The task-conditioning enables the hypernetwork to retain knowledge over a sequence of tasks, while it preserves the plasticity to learn a multitude of tasks. Our main contributions are related to continual state representation learning and continual reinforcement learning:

Continual State Representation Learning We proposed a hypernetwork-based approach to prevent a state representation encoder model from catastrophically forgetting when trained on multiple tasks. We trained the encoder model based on a combination of robotic priors, an inverse model, and a reward prediction model and protected the encoder from forgetting with our proposed hypernetwork regularizer. We compared the performance of multiple reinforcement learning agents trained either on hand-crafted numeric states or on state embeddings extracted by the state representation encoder from images in a continual and non-continual way. Our results show that the hypernetwork-based continual state representation learning approach achieves comparable results to a non-continual state representation learning approach while retaining knowledge over a sequence of tasks.

Continual Reinforcement Learning We proposed a hypernetwork-based approach for model-free continual reinforcement learning to prevent catastrophic forgetting when a learning agent is presented with multiple tasks sequentially. We trained multiple separate actor-critic agents on the tasks in question and extracted the learned network weights as input to a hypernetwork. The hypernetwork continually learns to generate the weights of the task-specific agents based on our novel learning scheme. Again, we compared the performance of multiple agents trained either on hand-crafted numeric

states or images in a continual and non-continual way. The results show that the performance of the continual agents is comparable to the non-continual agents trained separately on the robotic manipulation tasks. The continual agent trained on the continual state representation showed that backward transfer in the encoder model improves performance of the agent compared to the agent trained on the non-continual encoder.

In this thesis we provided the foundation for hypernetwork-based state representation learning and hypernetwork-based model-free reinforcement learning. Future research can build upon the ideas we presented to enable robots to perceive and interact with their environment without catastrophically forgetting what they learned even on lengthy sequences of tasks. There are several directions that are interesting to explore:

Sim2Real transfer We presented a simulation environment for our robot and trained the control networks only in this artificial setup. The goal in reinforcement learning for robotic manipulation tasks is to transfer the learned models to a real-world setup. We did not perform the transfer from simulation to reality as we did not include techniques to close the reality-gap. Future research needs to be conducted to evaluate and improve the capabilities of our hypernetwork-based approach in conjunction with methods for sim-to-real transfer.

Domain Adaption We preprocessed the images from our simulated robot captured by our virtual cameras by passing them through a feature extraction model. This model is pretrained on an image dataset designed for classification and segmentation of real-world photos. We propose that the difference in the domains of virtually captured images and real photos leads to a decreased performance. In future research, a fine-tuning of the feature extraction model to our domain and a hypernetwork-based approach to protect the fine-tuned model from catastrophically forgetting should be evaluated. Additionally, a promising direction for future research is to extend our monocular setting to a multi-view setup to improve the prediction of three-dimensional properties.

Task-Agnostic Learning The task-conditioned hypernetworks we used throughout this thesis depend on a learnable task embedding to distinguish between the tasks at hand. This task dependent approach enforces clear boundaries between tasks and that the current task is known at all times. In continual and lifelong learning these assumptions are neither always given nor desirable. A combination of an automated task detection mechanism and our hypernetwork-based approach to state representation learning and reinforcement learning can enable robots to learn task-agnostic and without supervision.

List of Figures

List of Tables

Bibliography

[Agr+16] P. Agrawal, A. Nair, P. Abbeel, J. Malik, and S. Levine. "Learning to Poke by Poking: Experiential Learning of Intuitive Physics" (June 2016).

[Ass+15] J.-A. M. Assael, N. Wahlstrom, T. B. Schon, and M. P. Deisenroth. "Data-Efficient Learning of Feedback Policies from Image Pixels using Deep Dynamical Models" (Oct. 2015).

[Atk+20] C. Atkinson, B. McCane, L. Szymanski, and A. Robins. "Pseudo-Rehearsal: Achieving Deep Reinforcement Learning without Catastrophic Forgetting" (Dec. 2020). DOI: 10.1016/j.neucom.2020.11.050. URL: http://arxiv.org/abs/1812.02464.

[AW21] A. Ayub and A. R. Wagner. "Continual Learning of Visual Concepts for Robots through Limited Supervision" (2021). DOI: 10.1145/3434074.3446357. URL: https://doi.org/10.1145/3434074.3446357.

[Bel57] R. Bellman. *Dynamic Programming*. Dover Publications, 1957.

[Bho19] A. Bhoi. "Monocular Depth Estimation: A Survey" (Jan. 2019).

[Boh+15] W. Bohmer, J. T. Springenberg, J. Boedecker, M. Riedmiller, and K. Obermayer. "Autonomous Learning of State Representations for Control: An Emerging Field Aims to Autonomously Learn State Representations for Reinforcement Learning Agents from Their Real-World Sensor Observations." *KI - Kunstliche Intelligenz* 29 (4 Nov. 2015), pp. 353–362. ISSN: 0933-1875. DOI: 10.1007/s13218-015-0356-1.

[BRK18] A. S. Benjamin, D. Rolnick, and K. Kording. "Measuring and regularizing networks in function space" (May 2018).

[Bro+16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. "OpenAI Gym" (June 2016).

[Cha+19] A. Chaudhry, M. Ranzato, M. Rohrbach, and M. Elhoseiny. "Efficient Lifelong Learning with A-GEM." *7th International Conference on Learning Representations, ICLR 2019* (Dec. 2019). URL: http://arxiv.org/abs/1812.00420.

[Chu+18] K. Chua, R. Calandra, R. McAllister, and S. Levine. "Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models" (May 2018).

[Col+15] P. Collaboration, P. A. R. Ade, N. Aghanim, et al. "Planck 2015 results. XIII. Cosmological parameters" (Feb. 2015). DOI: 10.1051/0004-6361/201525830.

[CZA17] H. Chougrad, H. Zouaki, and O. Alheyane. "Convolutional Neural Networks for Breast Cancer Screening: Transfer Learning with Exponential Decay" (Nov. 2017).

[DGF19] H. C. Dupre, M. Garcia-Ortiz, and D. Filliat. "S-TRIGGER: Continual State Representation Learning via Self-Triggered Generative Replay" (Feb. 2019). URL: http://arxiv.org/abs/1902.09434.

[DR11] M. P. Deisenroth and C. E. Rasmussen. *PILCO: A Model-Based and Data-Efficient Approach to Policy Search*. 2011.

[Esp+18] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures" (Feb. 2018). URL: http://arxiv.org/abs/1802.01561.

[FAL17] C. Finn, P. Abbeel, and S. Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks" (Mar. 2017). URL: http://arxiv.org/abs/1703.03400.

[Fer+17] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. "PathNet: Evolution Channels Gradient Descent in Super Neural Networks" (Jan. 2017). URL: http://arxiv.org/abs/1701.08734.

[FHM18] S. Fujimoto, H. van Hoof, and D. Meger. "Addressing Function Approximation Error in Actor-Critic Methods" (Feb. 2018).

[Fin+15] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. "Deep Spatial Autoencoders for Visuomotor Learning" (Sept. 2015). URL: http://arxiv.org/abs/1509.06113.

[Fle+21] Y. Flet-Berliac, J. Ferret, O. Pietquin, P. Preux, and M. Geist. "Adversarially Guided Actor-Critic" (Feb. 2021). URL: http://arxiv.org/abs/2102.04376.

[Fur+16] T. Furlanello, J. Zhao, A. M. Saxe, L. Itti, and B. S. Tjan. "Active Long Term Memory Networks" (June 2016). URL: http://arxiv.org/abs/1606.02355.

[Gif+17] M. Giftthaler, M. Neunert, M. Stauble, M. Frigerio, C. Semini, and J. Buchli. "Closing the Sim2Real Gap using Invertible Simulators." *Advanced Robotics* 31 (22 Nov. 2017), pp. 1225–1237. ISSN: 15685535. DOI: 10.1080/01691864.2017.1395361.

[GML15] R. Goroshin, M. Mathieu, and Y. LeCun. "Learning to Linearize Under Uncertainty" (June 2015).

[Goo] Google. *Google Remote Procedure Call (gRPC)*. URL: https://grpc.io/.

[Goo+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative Adversarial Networks" (June 2014). URL: http://arxiv.org/abs/1406.2661.

[Gri+19] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. "A Survey of Deep Learning Techniques for Autonomous Driving" (Oct. 2019). DOI: 10.1002/rob.21918.

[Haa+18] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor" (Jan. 2018). URL: http://arxiv.org/abs/1801.01290.

[Haa14] J. K. Haas. "A history of the unity game engine." *Diss. WORCESTER POLYTECHNIC INSTITUTE* (2014), p. 32.

[Had+20] R. Hadsell, D. Rao, A. A. Rusu, and R. Pascanu. *Embracing Change: Continual Learning in Deep Neural Networks*. Dec. 2020, pp. 1028–1040. DOI: 10.1016/j.tics.2020.09.004.

[Haf+18] D. Hafner, T. Lillicrap, I. Fischer, R. Villegas, D. Ha, H. Lee, and J. Davidson. "Learning Latent Dynamics for Planning from Pixels" (Nov. 2018).

[Haf+19] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi. "Dream to Control: Learning Behaviors by Latent Imagination" (Dec. 2019).

[Har] Harvard. *The History of Artificial Intelligence*. URL: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/.

[HCM15] A. Hallak, D. D. Castro, and S. Mannor. "Contextual Markov Decision Processes" (Feb. 2015).

[HDL16] D. Ha, A. Dai, and Q. V. Le. "HyperNetworks" (Sept. 2016).

[Hoc91] S. Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen." *Diploma, Technische Universitat Munchen* 91 (1 1991).

[Hoo+19] H. V. Hoof, N. Chen, M. Karl, P. V. D. Smagt, and J. Peters. *Stable Reinforcement Learning with Autoencoders for Tactile and Visual Data*. 2019.

[Hua+16] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks" (Aug. 2016).

[Hua+21] Y. Huang, K. Xie, H. Bharadhwaj, and F. Shkurti. "Continual Model-Based Reinforcement Learning with Hypernetworks." *Proceedings: ICRA. IEEE International Conference on Robotics and Automation* (Sept. 2021). URL: http://arxiv.org/abs/2009.11997.

[HVD15] G. Hinton, O. Vinyals, and J. Dean. "Distilling the Knowledge in a Neural Network" (Mar. 2015). URL: http://arxiv.org/abs/1503.02531.

[JB15] R. Jonschkowski and O. Brock. "Learning state representations with robotic priors." *Autonomous Robots* 39 (3 Oct. 2015), pp. 407–428. ISSN: 15737527. DOI: 10.1007/s10514-015-9459-7.

[Jos+20] J. Josifovski, M. Malmir, N. Klarmann, and A. Knoll. "Continual learning on incremental simulations for real-world robotic manipulation tasks." 2020, Nicht–veroffentlichter.

[Kar+16] M. Karl, M. Soelch, J. Bayer, and P. van der Smagt. "Deep Variational Bayes Filters: Unsupervised Learning of State Space Models from Raw Data" (May 2016).

[Kar16] A. Karpathy. *Deep Reinforcement Learning: Pong from Pixels*. [Online; accessed 12-February-2022]. May 2016. URL: https://karpathy.github.io/2016/05/31/rl/.

[KB14] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization" (Dec. 2014).

[KCD19] S. Kim, A. Coninx, and S. Doncieux. "From exploration to control: learning object manipulation skills through novelty search and local adaptation" (Jan. 2019). DOI: 10.1016/j.robot.2020.103710. URL: http://arxiv.org/abs/1901.00811.

[Khe+20] K. Khetarpal, M. Riemer, I. Rish, and D. Precup. "Towards Continual Reinforcement Learning: A Review and Perspectives" (Dec. 2020). URL: http://arxiv.org/abs/2012.13490.

[Kir+17] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell. *Overcoming catastrophic forgetting in neural networks*. 2017.

[Kir+20] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Perez. "Deep Reinforcement Learning for Autonomous Driving: A Survey" (Feb. 2020).

[KK17] R. Kemker and C. Kanan. "FearNet: Brain-Inspired Model for Incremental Learning." *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (Nov. 2017). URL: http://arxiv.org/abs/1711.10563.

[Kop00] M. Koppen. "The curse of dimensionality." Vol. 1. 2000, pp. 4–8.

[KT99] V. R. Konda and J. N. Tsitsiklis. *Actor-Critic Algorithms*. 1999.

[Kuk21] Kuka. "Sensitive robotics LBR iiwa." 2021. URL: https://www.kuka.com/-/media/kuka-downloads/imported/9cb8e311bfd744b4b0eab25ca883f6d3/kuka_lbr_iiwa_brochure_en.pdf.

[KW13] D. P. Kingma and M. Welling. "Auto-Encoding Variational Bayes" (Dec. 2013). URL: http://arxiv.org/abs/1312.6114.

[Lee05] M. Lee. *Actor-Critic Methods*. [Online; accessed 12-February-2022]. Apr. 2005. URL: http://incompleteideas.net/book/ebook/node66.html#fig:actor-critic.

[Les+18] T. Lesort, N. Diaz-Rodriguez, J.-F. Goudou, and D. Filliat. "State Representation Learning for Control: An Overview" (Feb. 2018). DOI: 10.1016/j.neunet.2018.07.006. URL: http://arxiv.org/abs/1802.04181.

[Les+19a] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. Diaz-Rodriguez. "Continual Learning for Robotics: Definition, Framework, Learning Strategies, Opportunities and Challenges" (June 2019). URL: http://arxiv.org/abs/1907.00182.

[Les+19b] T. Lesort, M. Seurin, X. Li, N. Diaz-Rodriguez, and D. Filliat. *Deep unsupervised state representation learning with robotic priors: a robustness analysis*. 2019. URL: http://www.ieee.org/publications.

[LH16] Z. Li and D. Hoiem. "Learning without Forgetting." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40 (12 June 2016), pp. 2935–2947. URL: http://arxiv.org/abs/1606.09282.

[Lil+15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. "Continuous control with deep reinforcement learning" (Sept. 2015).

[LL20] Y. Lu and J. Lu. "A universal approximation theorem of deep neural networks for expressing probability distributions." *Advances in neural information processing systems* 33 (2020), pp. 3094–3105.

[LR17] D. Lopez-Paz and M. Ranzato. "Gradient Episodic Memory for Continual Learning." *Advances in Neural Information Processing Systems* 2017-December (June 2017), pp. 6468–6477. URL: http://arxiv.org/abs/1706.08840.

[MDL18] A. Mallya, D. Davis, and S. Lazebnik. "Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights" (Jan. 2018). URL: http://arxiv.org/abs/1801.06519.

[Mir+19] S.-I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. "Improved Knowledge Distillation via Teacher Assistant" (Feb. 2019). URL: http://arxiv.org/abs/1902.03393.

[MKB16] J. Munk, J. Kober, and R. Babuska. "Learning state representation for deep actor-critic control." IEEE, Dec. 2016, pp. 4667–4673. DOI: 10.1109/CDC.2016.7798980.

[ML17] A. Mallya and S. Lazebnik. "PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning" (Nov. 2017). URL: http://arxiv.org/abs/1711.05769.

[Mni+13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing Atari with Deep Reinforcement Learning" (Dec. 2013).

[Mni+16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning" (Feb. 2016).

[OSL17] J. Oh, S. Singh, and H. Lee. "Value Prediction Network" (July 2017).

[Osw+19] J. von Oswald, C. Henning, J. Sacramento, and B. F. Grewe. "Continual learning with hypernetworks" (June 2019). URL: http://arxiv.org/abs/1906.00695.

[Pat+17] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. "Curiosity-driven Exploration by Self-supervised Prediction" (May 2017).

[PJB19] S. Padakandla, P. K. J, and S. Bhatnagar. "Reinforcement Learning in Non-Stationary Environments" (May 2019). DOI: 10.1007/s10489-020-01758-5.

[Raf+21] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations." *Journal of Machine Learning Research* 22 (268 2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[Reb+16] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. "iCaRL: Incremental Classifier and Representation Learning." *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-January (Nov. 2016), pp. 5533–5542. URL: http://arxiv.org/abs/1611.07725.

[Rob95] A. Robins. "Catastrophic Forgetting, Rehearsal and Pseudorehearsal." *Connection Science* 7 (2 1995), pp. 123–146. ISSN: 1360-0494. DOI: 10.1080/09540099550039318. URL: https://www.tandfonline.com/action/journalInformation?journalCode=ccos20.

[RT09] T. Raiko and M. Tornio. "Variational Bayesian learning of nonlinear hidden state-space models for model predictive control." *Neurocomputing* 72 (16-18 Oct. 2009), pp. 3704–3712. ISSN: 09252312. DOI: 10.1016/j.neucom.2009.06.009.

[Rus+14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge" (Sept. 2014).

[Rus+15] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. "Policy Distillation" (Nov. 2015). URL: http://arxiv.org/abs/1511.06295.

[Rus+16] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell. "Progressive Neural Networks" (June 2016). URL: http://arxiv.org/abs/1606.04671.

[Rus+17] A. A. Rusu, M. Vecerik, T. Rothorl, N. Heess, R. Pascanu, and R. Hadsell. *Sim-to-Real Robot Learning from Pixels with Progressive Nets*. 2017.

[San+15] J. Sanchez-Duran, J. Hidalgo-Lopez, J. Castellanos-Ramos, O. Oballe-Peinado, and F. Vidal-Verdu. "Influence of Errors in Tactile Sensors on Some High Level Parameters Used for Manipulation with Robotic Hands." *Sensors* 15 (8 Aug. 2015), pp. 20409–20435. ISSN: 1424-8220. DOI: 10.3390/s150820409.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sch+15] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. "Trust Region Policy Optimization" (Feb. 2015).

[Sch+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. "Proximal Policy Optimization Algorithms" (July 2017). URL: http://arxiv.org/abs/1707.06347.

[Sch92] J. Schmidhuber. "Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks." *Neural Computation* 4 (1 Jan. 1992), pp. 131–139. ISSN: 0899-7667. DOI: 10.1162/neco.1992.4.1.131.

[Ser+17] P. Sermanet, C. Lynch, Y. Chebotar, J. Hsu, E. Jang, S. Schaal, and S. Levine. "Time-Contrastive Networks: Self-Supervised Learning from Video" (Apr. 2017). URL: http://arxiv.org/abs/1704.06888.

[Ser+18] J. Serra, D. Suris, M. Miron, and A. Karatzoglou. *Overcoming Catastrophic Forgetting with Hard Attention to the Task*. 2018. URL: https://github.com/joansj/hat.

[She+16] E. Shelhamer, P. Mahmoudieh, M. Argus, and T. Darrell. "Loss is its own Reward: Self-Supervision for Reinforcement Learning" (Dec. 2016). URL: http://arxiv.org/abs/1612.07307.

[Sil+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529 (7587 Jan. 2016), pp. 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[Sil15] D. Silver. *Lectures on Reinforcement Learning*. [Online; accessed 12-January-2022]. 2015. URL: https://www.davidsilver.uk/teaching/.

[Tra+19] R. Traore, H. Caselles-Dupre, T. Lesort, T. Sun, G. Cai, N. Diaz-Rodriguez, and D. Filliat. "DisCoRL: Continual Reinforcement Learning via Policy Distillation" (July 2019). URL: http://arxiv.org/abs/1907.05855.

[VT18] G. M. van de Ven and A. S. Tolias. "Generative replay with feedback connections as a general strategy for continual learning" (Sept. 2018).

[Wan+16] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. "Sample Efficient Actor-Critic with Experience Replay" (Nov. 2016).

[Wat+15] M. Watter, J. T. Springenberg, J. Boedecker, and M. Riedmiller. "Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images" (June 2015).

[Wik21] Wikipedia. *Convolutional Neural Network — Wikipedia, die freie Enzyklopadie*. [Online; Stand 24. Januar 2022]. 2021. URL: https://de.wikipedia.org/w/index.php?title=Convolutional_Neural_Network%5C&oldid=217106402.

[WMJ21] X. Wang, S. Magnusson, and M. Johansson. *On the Convergence of Step Decay Step-Size for Stochastic Optimization*. 2021.

[WSD15] N. Wahlstrom, T. B. Schon, and M. P. Deisenroth. "From Pixels to Torques: Policy Learning with Deep Dynamical Models" (Feb. 2015).

[Yoo+17] J. Yoon, E. Yang, J. Lee, and S. J. Hwang. "Lifelong Learning with Dynamically Expandable Networks." *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (Aug. 2017). URL: http://arxiv.org/abs/1708.01547.

[Zha+20] J. Zhang, J. Kim, B. O'Donoghue, and S. Boyd. "Sample Efficient Reinforcement Learning with REINFORCE" (Oct. 2020).

[ZPG17] F. Zenke, B. Poole, and S. Ganguli. *Continual Learning Through Synaptic Intelligence*. July 2017, pp. 3987–3995. URL: http://proceedings.mlr.press/v70/zenke17a.html.

[ZSP18] A. Zhang, H. Satija, and J. Pineau. "Decoupling Dynamics and Reward for Transfer Learning" (Apr. 2018).

Appendix

To complement the approach and evaluation results we add an extensive list of plots to the appendix.

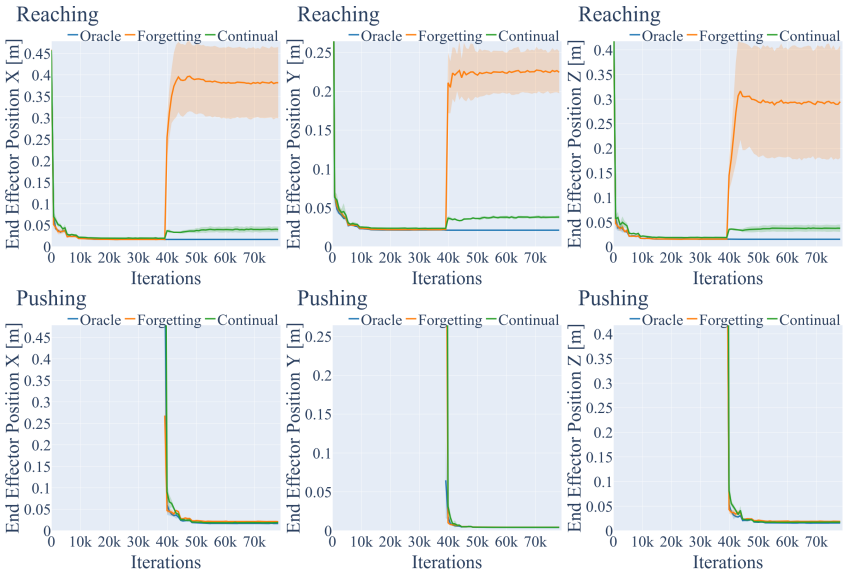
Reconstruction Error of Numeric Observations

The ground truth model predicts the numeric observation from the latent state embedding extracted by our SRL encoders as described in section 5.2. We call the remaining difference of the predicted numeric observation entry and the actual entry the reconstruction error.

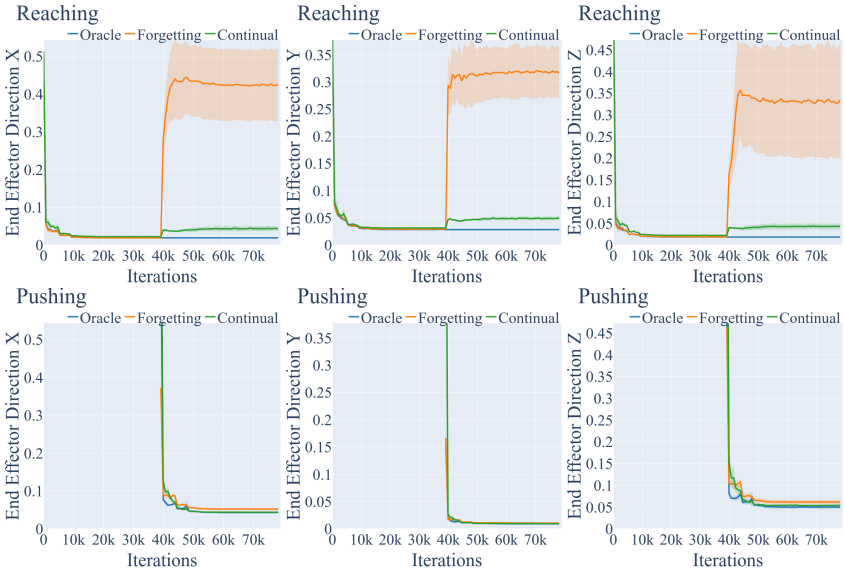
The following plots all display the reconstruction error of some entries in the numeric state. The top row in the plots displays the reconstruction error on the reaching task dataset over the whole sequential training. The bottom row in the plots displays the reconstruction error on the pushing task dataset over the whole sequential training.

We train on the reaching task dataset until we reach $\sim 40k$ iterations and switch to the pushing task for the rest of the training. The SRL models strictly only have access to training data of the current task. During training on the pushing task dataset we keep evaluating the models on the reaching task dataset.

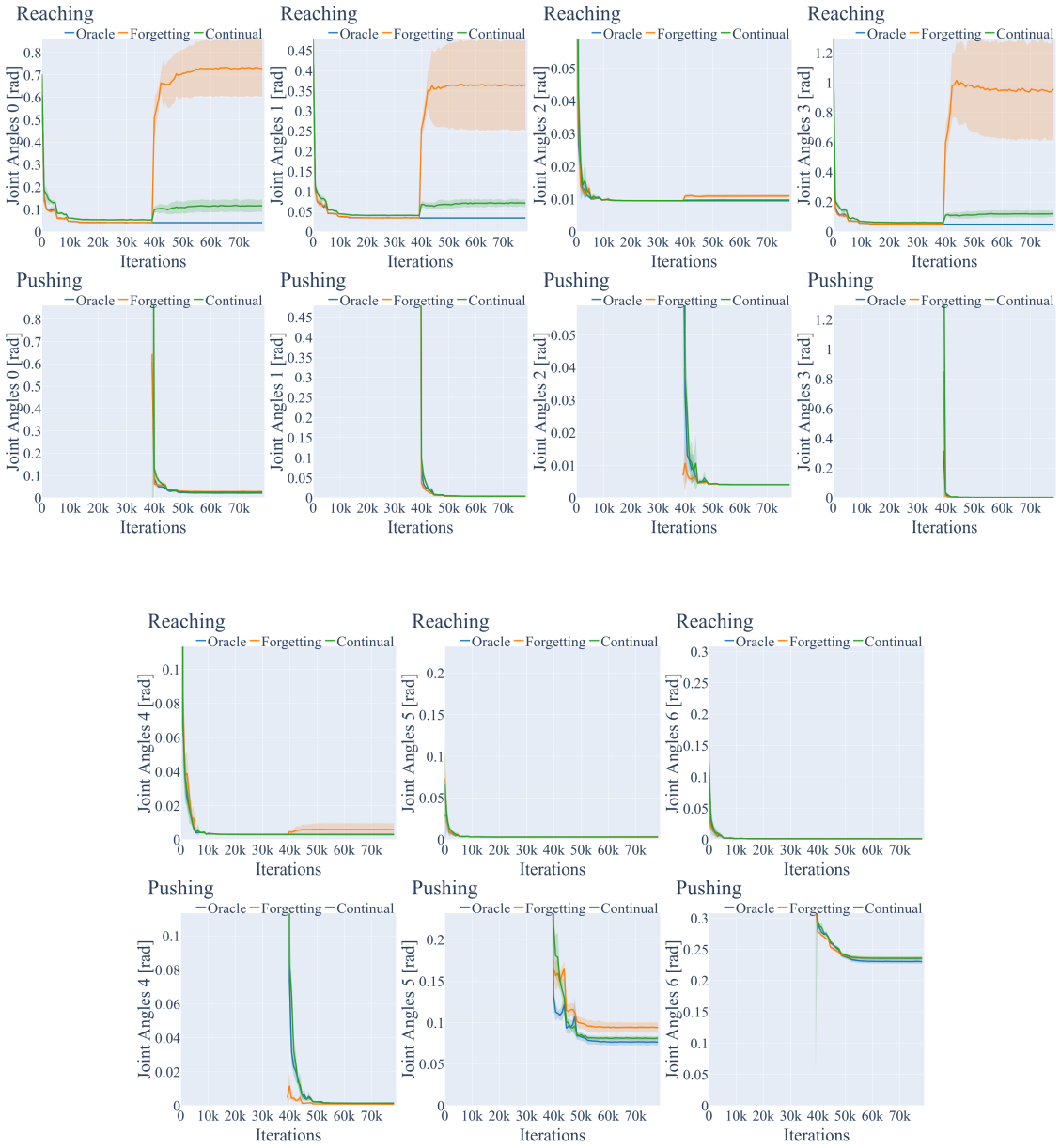
All the plots display that the forgetting SRL model cannot maintain the knowledge of the reaching task as it does not implement a protection against catastrophic forgetting. The continual SRL model maintains most of the knowledge of the reaching task as the hypernetwork prevents catastrophic forgetting. The oracle model maintains all the knowledge of the reaching task as the separate models avoid forgetting completely.



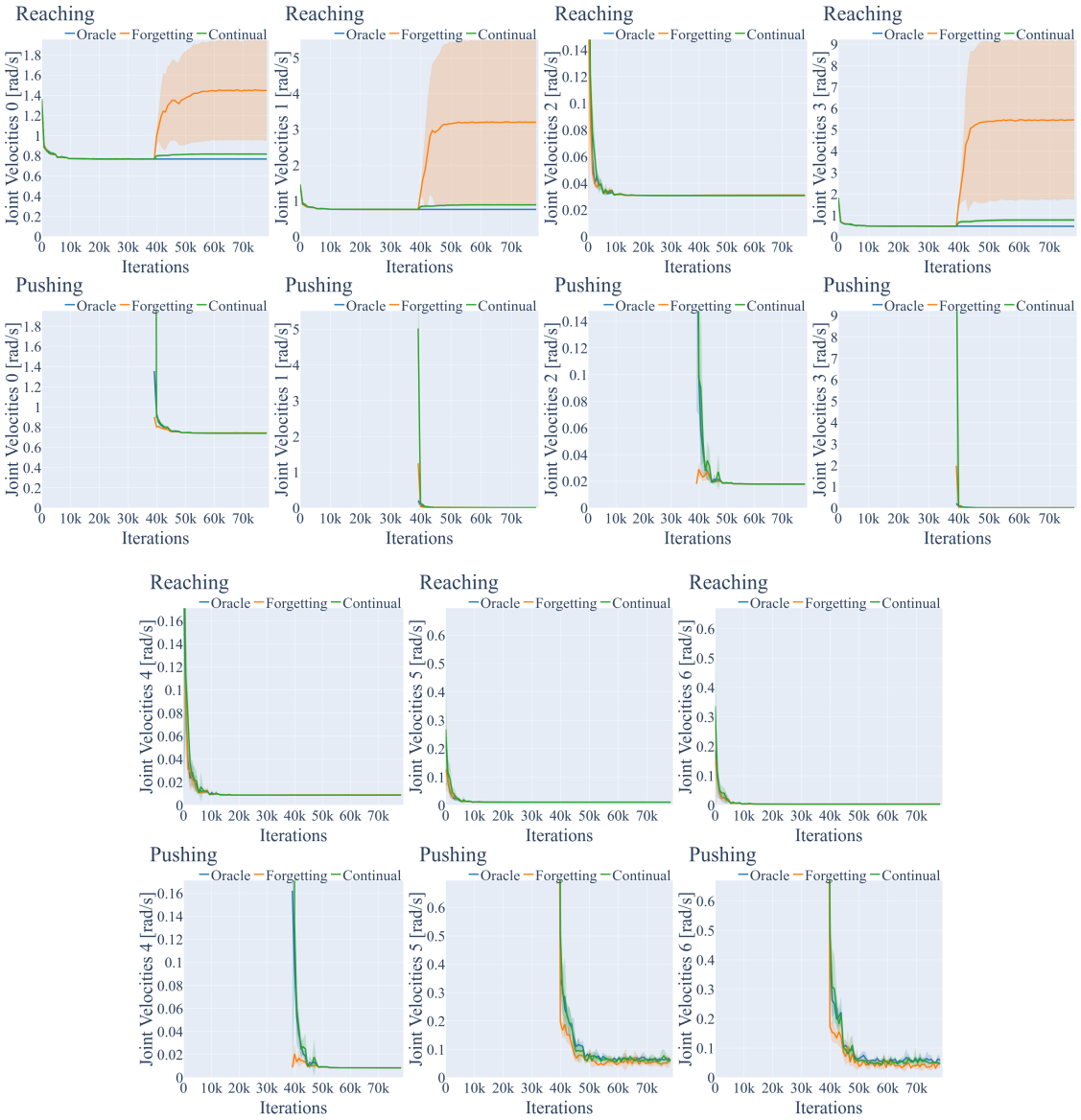
Reconstruction error of the end effector position in meters. All entries of the positions are reconstructable within a range of ~ 3 cm in either the reaching or pushing task.



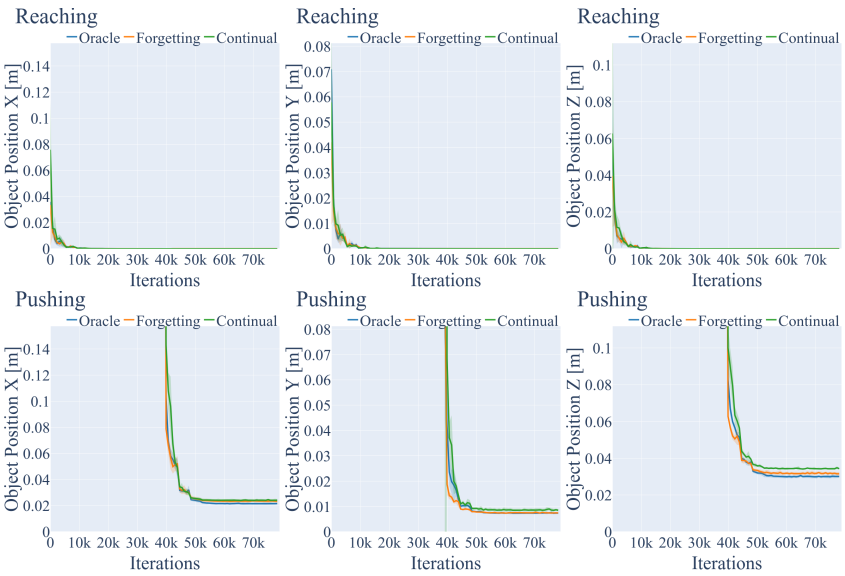
Reconstruction error of the end effector direction vector. All entries of the direction vector are reconstructable within a range of ~ 0.05 in either the reaching or pushing task.



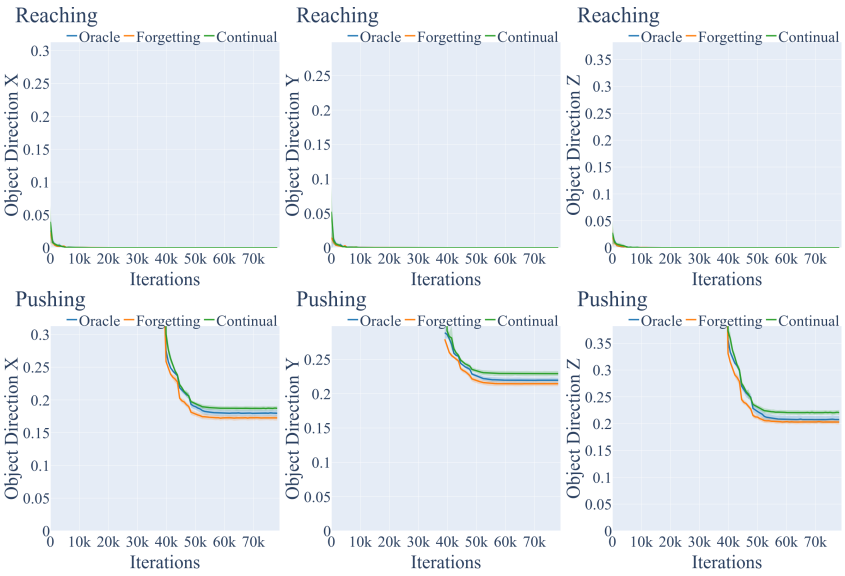
Reconstruction error of the joint angles in radians. All entries of the joint angles are reconstructable within a range of ~ 0.1 radians in either the reaching or pushing task. The joints $\{2, 4, 5, 6\}$ in the reaching task and the joints $\{1, 2, 3, 4\}$ in the pushing task are fixed and the joint angles can only vary due to collisions. Joint 6 is an exception and hard to predict from the latent state, as this is the rotation of the symmetric end effector around itself and never visible in the image.



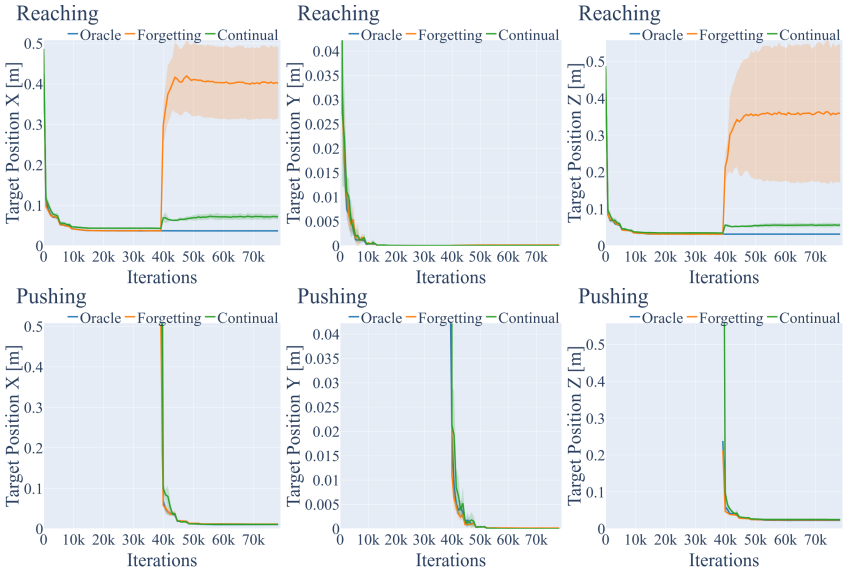
Reconstruction error of the joint velocities in radians per second. All entries of the joint velocities are reconstructable within a range of ~ 0.8 radians in either the reaching or pushing task. The velocities are hard to predict for the SRL encoder models as velocities are a time-dependent quantity, and the encoder only has access to the feature vectors at a single timestep and the previous action. Nonetheless, these large absolute values correspond to a relative error of $\sim 5\%$ within the $[-10, +10]$ range of possible velocities.



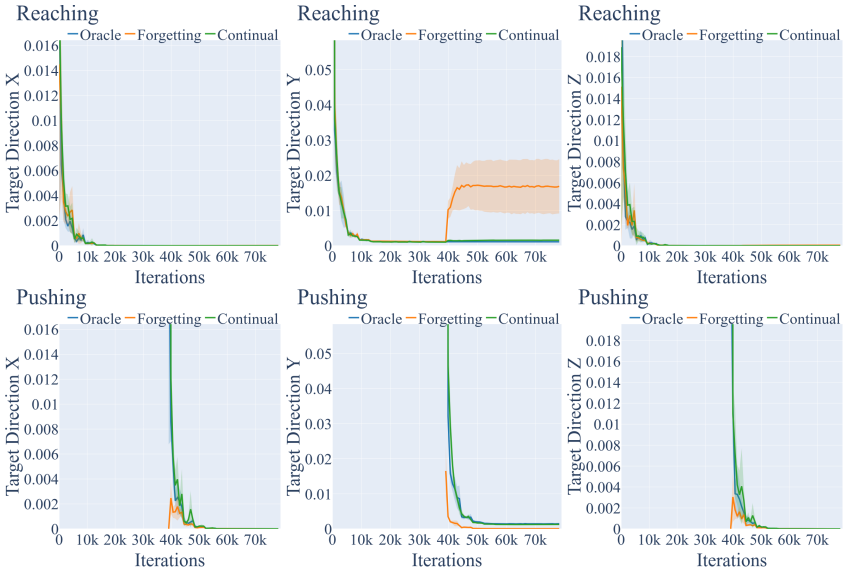
Reconstruction error of the object position in meters. All entries of the positions are reconstructable within a range of ~ 3 cm in either the reaching or pushing task. We do not use the object in the reaching task and set the position in the numeric observation to a default value. The ground truth model only predicts this default value and is thus independent of the input. This implies that even if there is forgetting in the SRL encoder, the output of the ground truth model for the reaching task is independent of it.



Reconstruction error of the object direction vector. All entries of the direction vector are reconstructable within a range of ~ 0.25 in either the reaching or pushing task. We do not use the object in the reaching task, what results in the same independence of the input for the reaching task as described above. The direction vector in the pushing task is hard to predict by the SRL encoder, as the green box is a symmetrical cube and looks the same from all sides.



Reconstruction error of the target position in meters. All entries of the positions are reconstructable within a range of ~ 3 cm in either the reaching or pushing task. The y entry of the target postion (the height) is fixed in the reaching task. The ground truth model predicts the fixed height and is thus independent of the input latent state.



Reconstruction error of the target direction vector. All entries of the direction vector are reliably reconstructable in either the reaching or pushing task. The direction vector is fixed in either task and the models should learn to predict no rotation — zero for all entries — independent of the input.