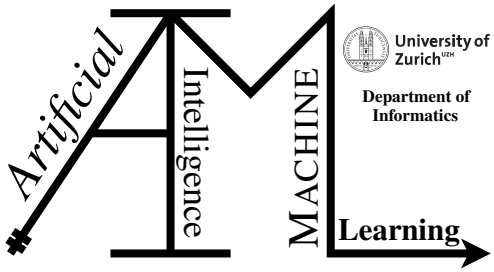
Influence Of Changing Environments On Model-Based Reinforcement Learning Algorithms

Master Thesis

Severin Siffert
14-720-536

Submitted on
October 4 2021

Thesis Supervisor
Prof. Dr. Manuel Günther



Master Thesis

Author: Severin Siffert, severin.siffert@uzh.ch

Project period: April 6 2021 - October 6 2021

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

Abstract

Current work on reinforcement learning algorithms for the reset-free setting focuses heavily on model-based agents because of their strong planning capabilities, especially for previously unseen tasks. Model-based reinforcement learning relies heavily on internal models of the world surrounding the agent, but not much attention is paid to shifting environment dynamics, which is likely to happen in various real-world scenarios. In this work, LiSP (Lifelong Skill Planning) and LSR (Learning Skillful Resets) are evaluated and compared against SAC (Soft Actor-Critic) in situations with different environment behaviour. Additionally, methods of environment shaping and environment dynamism are examined to facilitate the adaptation to new environment dynamics or as an alternative way to slowly introduce complex environment dynamics during training. The results suggest that both LiSP's and LSR's exploration mechanisms fail to explore both environments sufficiently to achieve good performance. SAC needs some help in the form of environment shaping or the help of occasional resets to achieve near-optimal performance in one environment, but fails to learn in the second. The results demonstrate what kinds of environment modifications are most useful to improve performance and which ones are better avoided.

Zusammenfassung

Aktuelle Fortschritte an Reinforcement Learning-Algorithmen in endlosen Umgebungen konzentrieren sich vor allem auf modellbasierte Methoden, da diese gut planen können und besonders bei komplett neuen Aufgaben brillieren. Modellbasiertes Reinforcement Learning ist stark auf interne Modelle der umgebenden Welt angewiesen. Es wird aber kaum beachtet, wie sich diese auf ändernde Umgebungen reagieren, obwohl in der realen Welt oft Veränderungen geschehen. In dieser Arbeit werden LiSP (Lifelong Skill Planning) und LSR (Learning Skillful Resets) evaluiert und mit SAC (Soft Actor-Critic) in Situationen mit unterschiedlichen Umgebungsverhalten verglichen. Zusätzlich zu diesen Vergleichen werden Techniken namens Environment Shaping und Environment Dynamism evaluiert, ob sie sich dazu eignen, während dem Trainingsprozess Komplexität langsam einzuführen. Die Resultate zeigen, dass LiSP und LSR die Umgebungen nicht genügend erkunden, um zufriedenstellende Leistung zu erbringen. SAC braucht Hilfe in Form von Environment Shaping oder gelegentliche Resets um in der einen Umgebung nahezu perfekte Leistung zu bringen, schafft es aber in der zweiten Umgebung nicht, etwas zu lernen. Die Resultate liefern ausserdem Hinweise darauf, welche Arten von Modifikation an den Umgebungen am meisten Nutzen bringen und welche Arten zu vermeiden sind.

Contents

Chapter 1

Introduction

In reinforcement learning (RL), an agent is placed in an environment and is supposed to solve a task. This is done by letting the agent do whatever it wants in trial-and-error style while it learns. After a certain (low) number of steps, the whole problem (minus the agent's logic) is reset to its original state and the agent may try again. This process is then repeated until training ends. The goal to be achieved is often some sort of movement task like walking in a straight line or navigating a maze. Minimal video games are also a relatively popular choice of task.

During model-based reinforcement learning (MBRL), learning how the environment behaves and which actions distribute rewards in what state is a crucial factor in creating a successful agent. Those learned environment dynamics (also called environment models) are used to hypothesize about the consequences of different courses of action, so that the best one can be chosen. In traditional reinforcement learning, the environment dynamics are learned through countless attempts at the same problem, where the environment is reset to a previous state quite often. Lately, the so-called reset-free setting has gotten more attention. The reset-free setting is concerned with cases where going back to the initial state is expensive, impractical, or completely infeasible. This is often the case where agents are deployed in the real world, e.g. a robot that works with large items. In those settings, using a training algorithm that requires little external intervention is highly desirable.

Recent work by Co-Reyes et al. (2020) has shown that RL agents should be viewed more in connection with their environment. They proposed environment shaping (making the environment more agent-friendly and slowly ramping up the difficulty) and environment dynamism (adding some randomness to the environment's behaviour) as quite effective ways to accelerate learning and improve stability of the learned behaviours. Besides those manual interventions in the training process, the environment may also change on its own during the training process. Environmental factors may change visibility or movement patterns because of fog or wind, pathways can get better or worse with attrition, or new agents could join the environment. While their results demonstrate substantial improvements in performance, they only looked at a very simple agents in a toy-scale problem.

Current algorithms for reset-free RL like Lifelong Skill Planning (LiSP) (Lu et al., 2021) or Learning Skillful Resets (LSR) (Xu et al., 2020) focus on learning a diverse and predictable set of skills to use at a later point in conjunction with a model of the environment. This works really well in all their demonstrated MuJoCo benchmarks. All benchmarks these and similar papers use, however, do not include anything that could be considered a 'dynamic' environment. For example the most popular benchmark, Ant-Waypoint, consists only of the agent itself and and a completely flat world. Nothing about this environment changes during or between trials except for the positioning of the waypoints. It is unknown how those algorithms behave in an environment that changes.

Since the proposed environment modifications environment shaping and environment dy-

namism make the environment less static, the algorithms may not work as well anymore. On the other hand, adding the right kind of dynamics may even improve performance in comparison with the more static environments. The purpose of this thesis is to investigate how the algorithms react to such changes and to find out what kind of modifications are beneficial.

To evaluate how the proposed environment modifications affect the algorithms, LiSP, LSR, and Soft Actor-Critic (SAC, the most popular benchmark for RL) (Haarnoja et al., 2019) are evaluated on the Volcano and 2D Minecraft environments introduced by Lu et al. (2021). In addition, their performance under various modifications according to environment shaping or environment dynamism is compared. The results show that LiSP and LSR both struggle surprisingly much with both tasks, even with the most useful modifications.

Not all versions of environment dynamism and environment shaping proved successful. Simply adding noise to movement commands turned out to be less useful than having a chance for random movements. Altering the environment improved performance the most when small modifications were made that alter the optimal path so that the previously optimal strategy needs to be adapted slightly.

Chapter 2

Background

A RL problem can be described as a Markov decision process that visits different states $s_1, \ldots, s_T$ in a state space S over T discrete time steps. The sequence of states visited is determined by the actions $a_1, \ldots, a_T$ the performing agent takes from an action space A. Which actions the agent takes is determined by the agent's policy $\pi(s_t) = a_t$, which is then applied to the environment to transition to the next state: $d(s_t, a_t) = s_{t+1}$. If an agent uses an internal representation of the (unknown) environment dynamics d, it uses an approximation denoted by q. The transition from one state to the next produces a reward r_t for the agent, which can be used to optimize the policy π.

It is possible to extend the policy in order to allow for multiple distinguishable behaviours called *skills*. Skills are denoted by either natural numbers $z \in \mathbb{N}$ for entirely distinct behaviours (as used in LSR in Section 2.3) or they can be used as gradual modifications to the base policy $z \in [-1, 1]^{dim(z)}$ (as used in LiSP in Section 2.4). In either case, the agent's policy not only depends on the current state, but also the skill(s) to be performed $\pi(s_t, z_t) = a_t$. The choice of skill is done by the skill policy $p(s_t) = z_t$ depending on the current state of the environment.

The following algorithms heavily rely on entropy H and mutual information I. (Shannon) entropy as defined in Eq. 2.1 describes how much randomness a random variable X or a distribution contains. The conditional entropy $H(X|Y)$ (Eq. 2.2) also describes the randomness in a random variable X, but with prior knowledge about a (likely related) different random variable Y.

$$H(X) = -\sum_{i}^{n} p(x_i) \log p(x_i) \tag{2.1}$$

$$H(X|Y) = -\sum_{i}^{n} p(x_i, y_i) \log \frac{p(x_i, y_i)}{p(y_i)} \tag{2.2}$$

$$H(X|Y, Z) = -\sum_{i}^{n} p(x_i, y_i, z_i) \log \frac{p(x_i, y_i, z_i)}{p(y_i, z_i)} \tag{2.3}$$

Mutual information $I(X, Y)$ (Eq. 2.4) explains how much information about X we can gain by observing Y. Similarly to the entropy, mutual information can also be extended to conditional mutual information $I(X, Y|Z)$ (Eq. 2.5), which then describes the expected mutual information $I(X, Y)$ given knowledge about random variable Z.

$$I(X; Y) = H(X) - H(X|Y) \tag{2.4}$$

$$I(X; Y|Z) = H(X|Z) - H(X|Y, Z) \tag{2.5}$$

2.1 Challenges of RL Algorithms

One classic problem during RL is the choice of exploration strategy. In order to learn how to solve a specific task, the RL agent needs to attempt a lot of different approaches and see which ones work well and which ones do not. In the well-known ϵ-greedy strategy, this is done by trying out random actions and using the results of those actions to estimate the reward of future actions. While this (mostly) random exploration produces good results in relatively simple problems, it has two major drawbacks: By taking random actions, certain states are unlikely to be reached and therefore never discovered. And by using a reward function to judge the reached states, a lot of a priori knowledge is required. (Eysenbach et al., 2018) Even if the reward function is known and the random exploration goes in the right direction, the agent may not hit a rewarding state, and stop short of finding a new path to rewards.

To avoid the problems introduced by reward functions, unsupervised exploration can be used. One approach to unsupervised exploration is examined in Eysenbach et al. (2018). Their algorithm DIAYN learns a set of diverse behaviours without using an external reward function. Those behaviours can then be used either in a hierarchical controller or as a basis for further training towards a specific reward function. A hierarchical controller would take the learned behaviours and then learn when to execute which behaviour.

A challenge for real-life application of RL is the requirement for lots of resets to an identical start state. In most RL formulations, the agent gets many thousand chances (episodes) to perform well from the same starting state. In-between episodes the policy is then updated to incorporate the new experiences. This resetting works very well for computer games and similar virtual environments. In real-life, such resets may not be very practical for many reasons: Even very precise machines have some amount of imprecision in their actions, so a perfect reset is impossible. Besides that, resets may take a long time to do, for example when a truck has flipped over because it drove off the road into slanted territory. If resets are fast to perform, they still can be expensive, for example when they require experts to intervene or when parts can break. (Lu et al., 2021)

For those reasons, it is desirable to have algorithms that do not require much external intervention, or preferably none at all. Such algorithms with limited need for resets (for an example see Section 2.3) still need to experience similar states multiple times to learn enough. This requires that the agent can somehow reset the environment to some extent from all states. If this is not possible (as in the flipped truck example), external resets are still needed occasionally.

2.2 Unsupervised Skill Learning Using Mutual Information

One problem many RL algorithms have is that they rely heavily on a (sparse) reward function. In many real-life applications such a reward function can only be supplied by a human, which may be expensive (Christiano et al., 2017). Certain algorithms, however, do not rely on a reward function for large parts of their training, which makes them very appealing in such cases. Algorithms that do not rely on an external reward function are called *unsupervised*.

Mutual Information can be used as a general reward function to encourage an unsupervised RL agent to explore the environment. Typically, it tries to maximize the entropy over all states S, and tries to minimize the entropy of the states given the taken action A:

$$I(S;A) = H(S) - H(S|A) \tag{2.6}$$

Maximizing $H(S)$ means that the agent tries to explore as much of the environment as possible. At the same time, minimizing the conditional entropy $H(S|A)$ makes the taken actions as predictable

as possible. (Sharma et al., 2020)

Certain approaches do not just train the entire policy as a whole on this objective, but also add in skills which should exhibit a palette of distinctive behaviours. Eysenbach et al. (2018) define a skill as a *latent-conditioned policy that alters that state of the environment in a consistent way*, meaning that the policy does not produce an action given a current state ($a_t = \pi(s_t)$), but can be influenced by a skill z which alters the behaviour: $a_t = \pi(s_t, z_t)$. Note that according to this definition it is ultimately irrelevant which actions are taken. Instead, it only cares about the reached state and which skill was used to get to that state.

The skill-based objective function tries to make the skills consistent, but also tries to maximize entropy in behaviour as much as possible to encourage exploration. In mathematical terms, the objective intends that the mutual information $I(S; Z)$ between reached state S and the performed skill Z is maximized. To make sure that the actions are not used to distinguish the skills, the mutual information between skills and actions given the state $I(A; Z|S)$ is minimized at the same time. Having distinguishable skills is, however, not sufficient to ensure that a large part of the state space is explored. Eysenbach et al. (2018) add another objective to do that: Rewarding a higher entropy of actions given a state $H(A|S)$ makes the agent try out more varying actions. Putting those objectives together, we get the objective function in Eq. 2.7

$$r_{intrinsic} = I(S; Z) + H(A|S) - I(A; S|Z) \tag{2.7}$$
$$= H(Z) - H(Z|S) + H(A|S, Z) \tag{2.8}$$

Simplifying, we get Eq. 2.8, which can be interpreted as well: $H(Z)$ intends that the agent develops as many skills as specified. The second term wants the skills to be clearly discriminable by the visited states, completely ignoring the actions taken. This is very much intended: in the end, we care about outcome, and not how the agent got there. As an example, look at a robotic arm grasping a cup: Unless the cup shatters, we do not care how much force is used to grasp the cup, as long as it is actually grasped. The last term $H(A|S, Z)$ encourages further exploration by rewarding more random behaviour. (Eysenbach et al., 2018)

These learned skills do not need to be used as they were trained in the end. Eysenbach et al. (2018) propose to take the learned skills and then keep training towards one specific reward function, since one or two of the learned skills will probably be close to what the actual goal is. Sharma et al. (2020), on the other hand, take the learned skills and put a controller on top of the skills which is then trained to perform the right skill at the right time, while the skills themselves stay the same.

One possible pitfall with hierarchical controllers can occur: If the learned skills are to be used in a hierarchical setting, it is an obvious idea to learn skills and policy to use those skills at the same time. However, if both are trained at the same time, then the skill controller quickly only chooses the most promising few skills and completely forgets about the rest. (Eysenbach et al., 2018)

2.3 Learning Skillful Resets

Learning Skillful Resets (LSR, Algorithm 1) (Xu et al., 2020) is an unsupervised approach to learning skills that requires no hard resets. LSR trains two policies: One that does standard RL, called forward policy, and one that resets the environment into a variety of start states using learned skills, called reset policy. The two policies then take turns controlling the agent. This training process is similar to Asymmetric Self-Play (Sukhbaatar et al., 2018). By having the reset policy challenge the performing policy, it is more likely that a significant part of the domain gets explored and the forward agent learns to perform its task from a variety of initial states. It is, of course, necessary that the environment/task can be reset by the agent in some way.

This reset game is implemented as a positive-sum game between two players. The task policy tries to score well, and the reset policy tries to make it harder for the task policy to succeed. To produce more diverse starting states, the reset policy also gets rewarded for producing a variety of different starting states. The two policies then learn very distinct skills that can later be used together as skill sets for a model-based planner or hierarchical controller.

Once training is done, the forward actor can be used to solve the task in a very robust manner because it has trained from many different starting states. The reset actor can be used like a DIAYN model with a variety of skills that can be used by a hierarchical controller, or to specialize towards a new task. This is especially useful if more models in the same environment should be trained because they can use different skills as a starting point for training.

Similar to DIAYN (Eysenbach et al., 2018), LSR uses the objective function $I(S;Z) - I(A;S|Z)$ as an intrinsic reward for the reset policy. $I(S;Z) = H(S) - H(S|Z)$ maximizes state coverage while maximizing predictability, and $I(A;S|Z) = H(A|S) - H(A|S,Z)$ makes sure the skills are recognized by the resulting states instead of by the actions they take. This results in the intrinsic reward function $r_{skill}(\pi, p, s, z, a) = \log p(z|s) - \log p(z) - \log \pi(a|s,z)$ (Xu et al., 2020). p represents the ground truth and prior assumptions. $p(z)$ is an assumed prior distribution of skills, usually a uniform distribution. $p(z|s)$ describes how likely the used skill can be predicted from the reached state, and $\pi(a|s,z)$ is the likelihood that he policy π takes action a given state s and skill z. Because p is unavailable to the algorithm, it has to be approximated with a learned function q. In contrast to DIAYN, LSR does not include an additional reward $H(A|S)$ for diverse behaviour within a skill for unmentioned reasons. Likely, this is because finding a challenging starting state for the forward policy is already encouragement enough to explore new states and the extra reward is not necessary.

The forward agent tries to optimize the reward $J^{forward} = \sum_t \gamma^t r(a_t, s_t)$, whereas the reset agent tries to optimize $J^{reset} = \sum_t \gamma^t r_{skill}(a'_t, s'_t) - \lambda J^{forward}$. Here, λ describes how important the received reward by the forward agent is compared to the intrinsic reward r_{skill}. For $\lambda = 0$, we get the same formulation as it is used in DIAYN, and $\lambda > 0$ encourages the reset controller to find more challenging starting states for the forward controller since it gets punished if the forward controller earns a high reward. Rewards further in the future are less likely to be accurate and should therefore be discounted a bit. γ controls the value decay over time and is usually set to $\gamma = 0.99$. The resulting reset game can be described as a Stackelberg Game, which means that the problem can be solved optimally. In the RL setting, this would require an infeasible amount of samples, but the solution can be approximated by alternating turns between the two agents, and with the reset agent having a slower learning rate. (Xu et al., 2020)

Some implementation details are worth clarifying:

- The prior skill policy $p(z)$ is not explained in Xu et al. (2020), but it is supposed to be *kept stable for an extended period of time*. DIAYN takes a uniform distribution during the whole training to maximize $H(z)$, which is part of the objective function.
- The learned skill discriminator q is taken from DIAYN and tries to predict the skill from the reached state. It is optimized using cross-entropy loss.
- Soft Actor-Critic (SAC) (Haarnoja et al., 2019) is used to optimize the two actors.

2.4 Lifelong Skill Planning

Lifelong Skill Planning (LiSP) (Lu et al., 2021) is an unsupervised skill learning framework meant to be used in MBRL, meaning it does not learn an optimal policy, but a set of useful skills that a skill policy then learns to use. It focuses on a reset-free setting by trying to avoid needing a reset,

Algorithm 1: Learning Skillful Resets (Xu et al., 2020), discriminator $q_\omega(s_t|a_t)$ corrected to $q_\omega(z|s_t)$

Input : Environment
Initialize: policy π, reset policy π_ϕ^{reset}, discriminator $q_\omega(z|s_t)$, prior $p(z)$
for N *iterations* **do**
  Sample skill $z \sim p(z)$
  # set environment
  for $t \leftarrow 0 \dots T_{reset-1}$ **do**
    Sample $a_t \sim \pi_\phi^{reset}(a_t|s_t, z)$
    Step env $s_{t+1} \sim \mathcal{P}_s(s_{t+1}|s_t, a_t)$, compute reward for reset controller r_t^{reset}
  # solve task
  for $t \leftarrow 0 \dots T-1$ **do**
    Sample $a_t \sim \pi(a_t|s_t)$
    Step env $s_{t+1} \sim \mathcal{P}_s(s_{t+1}|s_t, a_t)$, obtain reward r_t
  Update reset policy's final reward $r_T^{reset} = -\sum_{t=0}^{T} \gamma^t r_t(a_t, s_t)$
  Update π_ϕ^{reset}, π to maximize respective return using SAC
  Update discriminator q_ω using Adam

Algorithm 2: Lifelong Skill Planning (LiSP), online mode. In offline mode the last line is skipped. (Lu et al., 2021)

Initialize: true replay buffer D, generated replay buffer $\hat{D}$, dynamics model ensemble $\{f_{\phi_i}\}_{i=1}^N$, policy π, discriminator q, skill-practice distribution ψ
while *alive* **do**
  Update f_ϕ using D
  Update policy models with $UpdatePolicy(D, \hat{D}, f_\phi, \pi, q, \psi)$
  Perform $GetAction(s, f_\phi, \pi)$ and record experience in D

but also permits skill discovery from offline data or a standard environment with frequent resets. When LiSP is runing in online mode (see Algorithm 2), it alternates between taking actions and training its models. In offline mode, only training happens and the last line of Algorithm 2 is skipped.

One major goal of LiSP is to avoid resets. It is impossible to avoid them completely since any algorithm without prior knowledge will not know what actions will lead to a sink state and require a reset, but LiSP is very good at avoiding accidentally triggering sink states. The main mechanism to do this is within the model it uses to predict what happens to the environment: Instead of maintaining a single environment model, the environment dynamics are predicted with an ensemble of predictors $\{f_{\phi_i}\}_{i=1}^I$ that have the same architecture but different initial weights. Predictions for the future are then ran through all models in the ensemble. Only if the models agree enough with each other is the prediction actually used. Otherwise, a heavily negative reward is put in place to deter the policy from exploring that space by accident.

2.4.1 Training

LiSP learns the following four functions during Algorithm 3, $UpdatePolicy$:

- An environment model ensemble $\{f_{\phi_i}\}_{i=1}^I$ that predicts the next state of the environment

Algorithm 3: LiSP's $UpdatePolicy$ (Lu et al., 2021)

Hyperparameters: number of rollouts M, disagreement threshold α
Inputs : true replay buffer D, generated replay buffer $\hat{D}$, dynamics model f_ϕ, policy π, discriminator q, skill-practice distribution ψ
Function *UpdatePolicy(D,$\hat{D}$,f_ϕ,π,q,ψ)*:
  for *i*=1...M **do**
    Sample s_0^i from D
    Sample z^i with $\psi(s_0^i)$
    Generate a^i with $\pi(s_0^i, z^i)$
    Generate s_1^i with $f_\phi(s_0^i, a^i)$
    Add (s_0^i, a^i, z^i, s_1^i) to $\hat{D}$
  Using the M samples, update q to maximize $\log q(s_1|s_0, z)$
  Calculate $r_{adjusted}$ for samples in $\hat{D}$
  Update π, q, ψ using SAC with minibatches from $\hat{D}$

given a state and an action to perform.

- A skill-policy π that produces an action given a state and a skill to perform.
- A discriminator q used to compute the intrinsic reward during unsupervised training. It predicts how likely it is that a state follows a previous state if a specific skill was performed.
- A skill-practice distribution ψ that chooses skills to practice given the current state.

To learn as much as possible from every single experience, LiSP not only trains on the actual experiences (stored in replay buffer D) but also on a set of generated experiences (stored in generated replay buffer $\hat{D}$) that seem plausible given the previous real experience. Only dynamics model f_ϕ is trained on the real transitions. The other functions are trained during $UpdatePolicy$ (Algorithm 3) on data generated using f_ϕ. Because f_ϕ is constantly learning and creating more accurate generated states, generated replay buffer $\hat{D}$ has a rather small size so that it is completely replaced within 10 to 20 training steps, whereas a replay buffer usually stores data from one million exploration steps. The quick replacement of the generated experiences is needed because the environment model is updated continuously. Leaving the generated samples in for too long would mean that the model trains on outdated data for a longer time than is necessary. The true replay buffer on the other hand can contain a large amount of samples because those are true experiences that were collected in the real environment, so they do not need to be replaced.

Defining an intrinsic reward function is a crucial part for every unsupervised RL algorithm. LiSP uses an intrinsic reward $\tilde{r}$ (see Eq. 2.9) proposed by Sharma et al. (2020), which is derived from an approximation of $I(s_{t+1}; z|s_t)$. $q(s_{t+1}|s_t, z)$ says how likely state s_{t+1} is to occur when one step is taken from state s_t with skill z. This value is compared to how likely s_{t+1} is to follow s_t given any random skill z_i. To make this computationally feasible, this expectation is sampled from L random samples (Lu et al. (2021) use $L = 16$, Sharma et al. (2020) use $L = 512$) using a uniform distribution as $p(z)$. As a result, the intrinsic reward wants the different skills to be predictable (rewarded in the first term) and at the same time distinguishable (rewarded in the second term).

$$\tilde{r}(s, z, s_{t+1}) = \log q(s_{t+1}|s, z) - \log \frac{1}{L} \sum_{i=1}^{L} q(s_{t+1}|s, z_i) \text{ where } z_i \sim p(z) \tag{2.9}$$

Algorithm 4: LiSP's *GetAction* (Lu et al., 2021)

Hyperparameters: population size S, planning horizon H, planning iterations P, discount γ
Inputs : current state s_0, dynamics model f_ϕ, policy π
Function *GetAction*(s_0, f_ϕ,π)**:**
  for *p=1...P* **do**
    Sample skills $\{z^i\}_{i=1}^{S} \sim [-1,1]^{dim(z)\times H}$ based on distribution of previous iteration.
    Estimate returns $R = \{\sum_{t=0}^{H-1} \gamma^t r(s_t^i, \pi(s_t^i, z_t^i), s_{t+1}^i)\}_{i=1}^{S}$ using trajectory sampling, with states s^i sampled from f_ϕ for skills z^i.
    Use MPPI update rule on R and z to generate new distribution of skills $\{z_t\}_{t=0}^{H-1}$.
  return $a \sim \pi(s_0, z_0)$

To avoid training on imprecise generated experiences, the intrinsic reward $\tilde{r}$ is only awarded when the dynamics model ensemble $\{f_{\phi_i}\}_{i=1}^{I}$ is reasonably certain of its predictions. To see how certain f_ϕ is of its predicition, a disagreement score $dis(s,a)$ is calculated between the different models in the ensemble:

$$dis(s,a) = \mathbb{E}_{i\neq j}[\|f_{\phi_i}(s,a) - f_{\phi_j}(s,a)\|_2^2] \tag{2.10}$$

When $dis(s,a)$ is too large (larger than hyperparameter α_{thresh}), a constant penalty $-\kappa$ is awarded instead of $\tilde{r}$. This results in $\tilde{r}_{adjusted}$, which is used as an intrinsic reward function:

$$\tilde{r}_{adjusted} = \begin{cases} \tilde{r} & \text{if } dis(s,a) \leq \alpha_{thresh} \\ -\kappa & \text{if } dis(s,a) > \alpha_{thresh} \end{cases} \tag{2.11}$$

After generating new samples for generated replay buffer $\hat{D}$, policy π, discriminator q, and skill practice distribution ψ are updated using Soft Actor Critic Haarnoja et al. (2019) with mini-batches sampled from $\hat{D}$.

2.4.2 Generating Actions

The goal of *GetAction* (Algorithm 4) is to develop a sequence of skills to perform and to return the next action to take. LiSP does this with a Model-Predictive Control (MPC) approach Nagabandi et al. (2019), meaning it tries to predict the changes in state and reward if it takes different courses of action. The MPC implementation used in LiSP to optimize the plan is Model-Path Predictive Integral (MPPI) (Williams et al., 2015), which uses trajectory sampling. Trajectory sampling is a technique where a number of slightly random courses of action are simulated and averaged to find the best base plan.

Typically, MPC approaches search through the entire action space to find the best course of action for a planning horizon of H steps. Similar to Dynamics-Aware Discovery Of Skills (DADS) (Sharma et al., 2020), however, LiSP plans its course of action in terms of skills the agent has learned, which reduces the planning space significantly in comparison to action-based planning. This allows for a substantially larger planning horizon. They do, however, examine the individual actions taken by the skills so that they avoid sink states with very high certainty. Because the policy is encouraged to only perform actions where the dynamics model is confident in its predictions (see $\tilde{r}_{adjusted}$ in Eq. 2.11), it is even more unlikely that a sink state is accidentally reached.

To avoid repeating optimization work, the plan of skills to use (distribution of skills $\{z_t\}_{t=0}^{H-1}$ in Algorithm 4) is stored between the different calls of $GetAction$. Whenever an action is taken, the first skill to perform is cut from the plan, the rest of the planned skills are advanced one time slot, and a neutral skill is added at the end of the sequence. This way the planning iterations build upon each other instead of discarding the whole optimization and starting from scratch every time.

An additional technique (not visualized in Algorithm 4 but implemented in Lu et al. (2021)) to expand the planning horizon is to perform multiple actions for every planned skill. In doing so, the algorithm looks farther into the future without making the planning space larger. When this technique is used, the planning loop is only performed every n steps, where n is the number of times the same skill is used to sample an action.

The MPPI update rule (Eq. 2.12) is *simply a reward-weighted average of random variations in the [sequence of skills]* (Williams et al., 2015). The environment model f_ϕ and policy π are used to predict what happens when a sequence of skills $\{z_t\}_{t=0}^{H-1}$ is performed from the current state. A lot of slightly different sequences of skills are generated by adding some Gaussian noise to the currently planned sequence (reminder: in LiSP, skills are defined as $z \in [-1, 1]^{dim(z)}$ and not as a natural number like in LSR), which then are all estimated and averaged to produce an updated sequence of skills according to Eq. 2.12. The update takes more inspiration from the better random deviations while taking less from the worse ones. It relies on a temperature parameter λ and weighs the inputs based on the cost (or negated reward) $c_{t,m} = \sum_{n=1}^{t}(-r_{n,m})$, which is the cumulative cost of all steps up to step t of the predicted state of sample m.

$$z_t \leftarrow z_t + \sum_{n=1}^{M} \frac{\exp(-\frac{1}{\lambda}c_{t,n})(z_{t,n} - z_t)}{\sum_{m=1}^{M}\exp(-\frac{1}{\lambda}c_{t,m})} \tag{2.12}$$

2.5 Entropy-based Learning

Currently, the most popular family of supervised learning algorithms is the group of algorithms based on entropy maximization. Soft Actor-Critic (SAC) (Haarnoja et al., 2019) is *the* baseline everyone competes against, together with Proximal Policy Optimization (PPO) (Schulman et al., 2017), which is SAC's precursor. This family of algorithms maximizes the objective function

$$L = R + \alpha H(S) \tag{2.13}$$

where R is the earned reward and $\alpha H(S)$ is the entropy over the visited states multiplied by a scaling factor (also called temperature) to account for reward scales and relative importance of the terms. Just like in LiSP and LSR, adding the entropy to the objective function is here to encourage exploration of the entire state space.

The main improvement of SAC over PPO is its off-policy formulation. PPO is a so-called on-policy algorithm, which means that only data collected with the current version of the policy can be used for training and after just one training step all data needs to be discarded. Haarnoja et al. (2018) managed to create an algorithm to optimize for the same objective from Eq. 2.13 but in an off-policy formulation. Off-policy means that all collected data can be used to train, even if it was collected with an entirely different algorithm. As a further improvement, Haarnoja et al. (2019) then also found a small adjustment that allows the temperature parameter α to be adjusted automatically, so that almost no parameter tuning needs to be done. Thanks to its off-policy formulation and the automatic temperature adjustment, SAC has become a crucial component of many algorithms, including LiSP, LSR, and their precursors.

2.6 Environment Shaping

Traditionally, in environments with very sparse rewards, reward shaping is used to make it easier for policies to learn. For example, an agent is not just rewarded when it completes the task, but also a small reward is given when an intermediate step is completed, and maybe even an even smaller reward when it moves towards the correct position for the next step. While reward shaping can solve a lot of issues, it is not without problems: It often requires access to privileged information (such as the distance to the next goal) which may not be available to the agent through its normal sensors. This is rarely a problem in simulations, but for the real world collecting that information may be very expensive. In addition, it is also very hard to create a good reward function. Intermediate rewards may discourage the agent from searching for the ultimate reward because it's not large enough, or a path to the solution may exist but is not recognized with intermediate rewards, so it will never be explored.

Environment shaping is a technique proposed by Co-Reyes et al. (2020) as a more natural alternative to reward shaping to help agents get over the initial hurdles of sparse rewards much more quickly. When using environment shaping, the agent is placed in a much more cooperative environment at the beginning of training, just like human parents support their kid at the beginning of its life. For example, in a deer hunting scenario, where the deer will flee in the evaluation setting, the deer starts off by walking towards the agent. After some success the deer might stop moving towards the agent and starts to move randomly instead. Once the agent has learned to move towards the randomly moving deer, the agent can be placed in the real setting where the deer starts to move away from the agent. If the agent was placed in the real setting from the start, it may have never even reached one deer and would be stuck randomly exploring for a very long time. Of course, the evaluation environment is not changed.

Co-Reyes et al. (2020) attribute their improved training speed to the smaller disparity between the current and the optimal policy. RL learns best if the optimal policy is not too different from the current agent's behaviour. The problem in the deer hunting scenario is that the optimal behaviour is very different from the random actions a RL agent typically takes at the start. With the deer moving towards or very close to the agent, even a random policy has a good chance at succeeding because it is easy to stumble upon the correct action sequence. It is also possible that the starting positions are changed instead of (or in addition to) the environment's behaviour: If the deer starts very close to the untrained agent, it is likely for the agent to catch the deer even if it tries to flee.

2.7 Environment Dynamism

Environment dynamism is the practice of adding some randomness to the behaviour of the environment, also proposed by Co-Reyes et al. (2020). It is very similar to environment shaping, even in the benefits it provides, but more like a stochastic version of it. In their experiments they found that giving certain elements in the environment (such as the deer in the previous example) a probability (in the example 10-20%) of completely random behaviour instead of their normal behaviour improves performance significantly, both in training and evaluation. The more dynamic environment $\tilde{d}$ can be described as having more entropy in its transitions than the real environment d:

$$H_{\tilde{d}}(s_{t+1}|s,a) \geq H_d(s_{t+1}|s,a) \tag{2.14}$$

In their discussion of the cause of those performance gains, Co-Reyes et al. (2020) attribute the gains to the fact that having more entropy during training will result in a more uniform distribution of all the visited states. Because more states are visited, the agent's policy is nudged towards a more generally useful one. Especially at the beginning (when the policy is completely untrained

and essentially random) a more random environment increases the likelihood of encountering a rewarding state. And when the agent gets stuck in a place in a reset-free environment, the randomness can act like a soft reset function. An experiment with sparse rewards showed a 28% decrease in the average time until the first rewarding state was reached. When the entropy is increased too far however, the state distribution becomes too uniform and the actions performed by the agent become irrelevant, making training useless.

Co-Reyes et al. (2020) also claim that traditional RL theory forgets about the fact that the real world is quite dynamic already and that the typical benchmarks (like the MuJoCo (Brockman et al., 2016) environment) are too clinical. In the real world, even if an agent does nothing, it's likely that it can learn a lot already just from observing what happens around it. They say that adding randomness to pre-training in a simulator is likely beneficial because the learned policy is then a bit more stable to small errors in measurement. The impact on environment models for MBRL algorithms is unclear and should be investigated more.

Chapter 3

Experimental Evaluation

The goal of this thesis is to examine the performance of current MBRL agents for the reset-free setting under changing environment dynamics as described by Co-Reyes et al. (2020). The first step is getting LiSP[1] and LSR[2] to run on LiSP's Volcano and 2D Minecraft environments, called *volcano-baseline* and *minecraft-baseline* in this thesis. Besides the reproduction, potential issues in the implementations are examined for their relevance to the algorithms working.

Following that, LiSP and LSR are evaluated under novel conditions that employ environment shaping and environment dynamism. Those new experiments are all adapted from the environments provided in LiSP's implementation. In addition to examining LiSP and LSR, a comparison is also made against Soft Actor-Critic (SAC) (Haarnoja et al., 2019), which is one of the most-used baseline algorithms. Hyperparameters are set to the values recommended in the papers.

The most common baseline environments for RL papers are the MuJoCo environments (Brockman et al., 2016). They are, however, not suited for this work because the license forbids modifications, so environment shaping and environment dynamism are illegal to try. Another set of candidate environments to use in the experiments were the ones used in Co-Reyes et al. (2020), but incompatible libraries made the effort to get them to work with the algorithms too large.

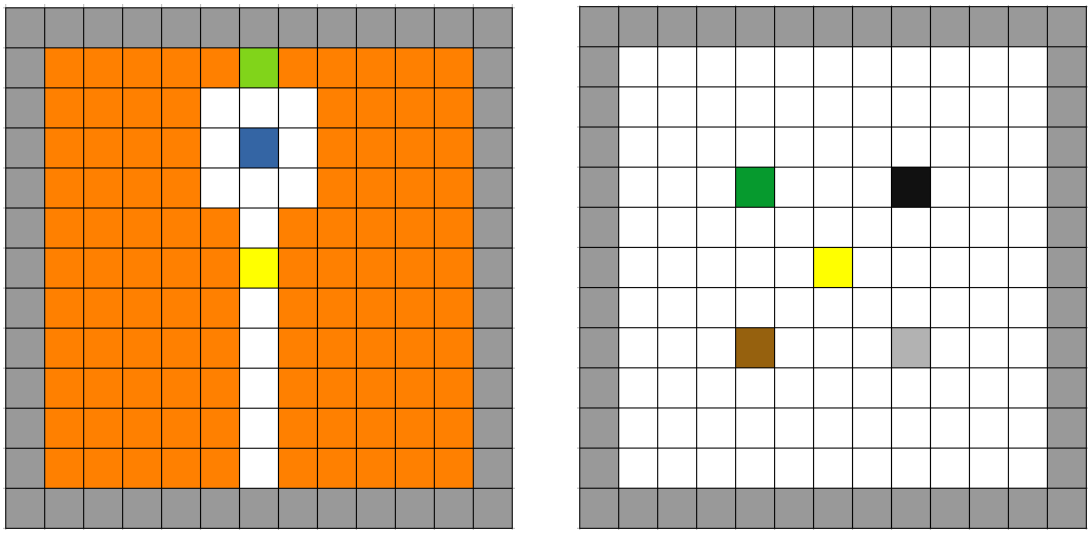
3.1 Experiment 1: Volcano

The Volcano environment is a simple 2D navigation task where the agent needs to move towards a goal while navigating around a hole. Besides the goal and the inescapable hole, there is also lava, which should be avoided. The hole is located between the starting position of the agent and the goal. The most direct path (around the hole) to the goal is safe; everything else is filled with lava, as is displayed in Figure 3.1(a). The picture makes it look like a discrete environment, but it actually is continuous. Possible actions $a \in [-1, 1]^2$ consist of movement in horizontal and vertical direction, with the maximum step covering almost one tile in any direction. The state visible to the agent consists of the agent's position, the position of the hole, and the position of the goal.

When the agent lands in the hole, it is stuck there and the episode ends. Touching lava is not as bad as falling down the hole: it only distributes an additional punishment and the agent can continue on its path. To compute the reward the agent receives every step, the distance to the goal and the kind of tile the agent is standing on are taken into consideration: As a base reward, the negative L2-distance is used, and if the agent is standing in lava or in the hole, an additional

[1] Code available at https://github.com/kzl/lifelong_rl

[2] Code available at https://github.com/siddharthverma314/clcp-neurips-2020



(a) Volcano default layout. Green: Goal. Blue: Hole. Orange: Lava.

(b) Minecraft default layout. Brown: Crafting table. Green: Wood. Black: Stone. Light Grey: Iron.

Figure 3.1: ENVIRONMENT LAYOUTS. Pictured are the baseline layouts for the Volcano (a) and 2D Minecraft (b) environments. Yellow: Agent spawn/reset tile. Dark grey border: Wall.

punishment is awarded. Since the experiments are not about reward shaping and this is a not an extensively shaped reward function, it is used as provided.

This scenario is mainly about avoiding danger, which Lu et al. (2021) explicitly laude as a strength of LiSP. According to them, LiSP excels at not accidentally triggering sink states, which in this case would be falling in the hole. But, the Volcano scenario is also about balancing different threats: If the agent avoids the hole too much, then it will get punished for standing in lava.

3.1.1 Baseline

The task *volcano-baseline* is just what it says: a baseline to compare modifications to. The environment layout is the one displayed in Figure 3.1(a) and does not change. Whenever a terminal state is reached, the agent is reset to its starting position. Terminal states are reached when the agent reaches the goal or falls into the hole. Otherwise no resets happen, even if the agent gets stuck in a corner for hundreds of steps. Standing in lava simply gets a lower reward but is not terminal.

This baseline environment is the environment that is used for performance evaluation of the following tasks.

3.1.2 Baseline with Resets

The experiment *volcano-baseline-reset* is almost the same as *volcano-baseline*, but with the small difference that resets happen even if the agent has not reached a terminal state. When the agent reaches a terminal state OR if 500 steps since the last reset have passed, then the task is reset. This completely removes the reset-free aspect of the problems and is therefore much more similar to the episodic problems typical in RL.

In a setting where frequent resets happen, more aggressive exploration policies are favoured because they do not have to find a terminal state to try again. If they get stuck at some location,

they will get reset anyway. This also makes the experiment setup easier because there is less need for complex logic to find out when a reset may be useful. In the worst case, the agent will be stuck for the 500 steps until the reset happens.

One way the frequent resets may harm training is when the reset frequency is set too high. Then, the agent will experience the start of the problem and does not get enough time to explore the later parts of the problem. This is, however, certainly not a problem in this case as travelling from corner to corner of the environment is possible in 20 steps.

3.1.3 Randomized Actions

In the *volcano-random-actions* environment, the movement does not work as precise as in the others. While the layout is the same as the baseline, some random noise is added to the actions: To every action $a \in [-1, 1]^2$ a random value in $[-0.1, 0.1]$ is added in every dimension. Afterwards the values are clipped to be within $[-1, 1]$ again. The clipping could be left out, but that would allow the possibility of jumping over the hole, which would possibly alter the optimal strategy significantly, which is not the intention of adding noise.

Because of the random noise added to the actions the movement becomes less precise. While the noise is not very large in comparison to the commands, an additional margin of safety is needed since the noise can vary the final position by almost a third of a tile. Additionally, planning is harder because not everything is deterministic anymore.

3.1.4 Shifting Environment

In the environment *volcano-shifting* the floor layout shifts around slightly. Every 500 steps a new predefined layout is loaded. The new layouts are all mostly copied from the baseline layout, but the hole and the safe path around the hole are moved left, right, up, or down by a tile. Occasionally, the safe path can also be a bit bigger, and in one layout there are even two holes directly left and right of the optimal path.

These differences in the layout are supposed to challenge the pathfinding. By changing the location of the hole, the algorithm now needs to respond to the hole's location, whereas beforehand it was enough to follow the same path every time. This makes it harder to train, but should improve performance on unseen problems significantly.

3.1.5 Combination

The task *volcano-shifting-random* is a combination of the modifications from both *volcano-random-action* and *volcano-shifting*. This makes the problem even more challenging but should make the resulting algorithm even more robust.

3.2 Experiment 2: 2D Minecraft

The 2D Minecraft environment is a hierarchical 2D navigational task with the basic movement working exactly the same as in the Volcano environment. As in the real Minecraft game, better items can be gathered by spending lower-level items at the right location. The more complex the items are, the more reward is distributed to the agent. The limiting factor in this task is the small inventory size: Because the agent can only carry one item of every type at the same time, it has to go back to the same resource over and over to gather more resources for the next crafting steps. This setup lends itself especially well for skill-based algorithms because of those repeating tasks.

There is no overall inventory size limit since this is not supposed to be an inventory management component to the problem. The state visible to the agent consists of the agent's position, the position of the interesting tiles (resources and crafting table), as well as the inventory content.

The environment consists of mostly empty space with some resources and a crafting table located somewhere, as pictured in Figure 3.1(b). Since there is no danger to avoid, the experiment *minecraft-random* (analogous to *volcano-random*) was not performed. When touching the resource tiles, the agent collects one of the corresponding item (wood, stone, or iron) as long as it has the requisite tool available (wood: no item required, stone: wood pickaxe required, iron: stone pickaxe required), consuming the requisite tool. When the crafting table is visited, possible crafting recipes are used automatically: Wood can be turned into a stick, and a wood or stone block can be combined with a stick to create a wood or stone pickaxe. When a resource or item is acquired by stepping on the resource tile or the crafting table, the agent receives a reward. Items further in the task order award larger rewards (e.g. collecting a piece of wood awards a score of one, but crafting the wood into a stick awards a score of two). This leads to the following task progression (some steps do not have to be taken in this exact order):

1. Collect wood
2. Craft stick with wood
3. Collect wood
4. Craft wood pickaxe with wood and stick
5. Collect stone using wood pickaxe
6. Collect wood
7. Craft stick with wood
8. Craft stone pickaxe with stone and stick
9. Collect iron with stone pickaxe

3.2.1 Baseline

Just like with *volcano-baseline*, the task *minecraft-baseline* is simply a baseline to compare the following experiments with. It has a very similar configuration as *volcano-baseline*: Again, the agent is only reset when it reaches a terminal state. In the minecraft environment, this only is the case when the last step, getting iron, is achieved. Because this requires a lot of discrete tasks to be completed, learning in the minecraft environment is much slower than in the volcano environment. According to Lu et al. (2021), LiSP takes around ten times as long to learn the minecraft task as it takes to learn the volcano task.

This baseline environment is the environment that is used for performance evaluation of the following tasks. Its layout is displayed in listing 3.1(b).

3.2.2 Baseline with Resets

The task *minecraft-baseline-reset* has the exact same modifications in comparison with *minecraft-baseline* as *volcano-baseline-reset* has compared to *volcano-baseline*. The problem is reset if a terminal state is reached OR 500 steps have passed since the last reset.

For this task, the frequent resets are more aggressive in comparison to the volcano setting because the task takes much longer to achieve. The agent has to have a somewhat optimized

technique in order to get to the point of crafting more advanced tools before the time runs out. On the other hand, the first few basic tasks (gathering wood and making a stick out of it) can be learned much more quickly because the resets put the agent back into the position to try those tasks again. Eysenbach et al. (2018), the work LSR is built upon, explicitly state that their algorithm DIAYN is good at solving this kind of challenge where the start of the task is very repetitive and the possibilities only open up once the first step is completed.

3.2.3 Shifting Environment

In the *minecraft-shifting* environment, the interesting tiles (resources, crafting table) do not stay where they are. With a small probability (5% per time step used in all experiments) a resource tile or the crafting table move their location one tile up, down, left, or right. This forces the algorithm to figure out where the resources are at the moment. Having to learn that recognition skill instead of hard-coded locations makes the problem harder, of course, but should make the agent much more resilient to being thrown into completely unseen setups or an imperfect reset, which could happen in real-world scenarios.

3.2.4 Shaped Environment

Instead of randomly moving the tiles of interest, they can be manually placed. In the environments *minecraft-shaping* and *minecraft-line*, the tiles are arranged in different shapes so that the algorithm can collect more useful data early on. In the *minecraft-line* layout, the resources start in a line beginning from the starting position, so the agent only needs to move up a number of times to start solving the task. Afterwards, the tiles slowly start moving away from each other until they are entirely back to their positions as they are in the evaluation environment.

In the *minecraft-shaping* experiment, the tiles are in the same direction from the agent, but pulled much closer to the starting point. At the beginning, the agent starts on the crafting table and only needs to move half a tile to touch any of the resources. The tiles then slowly start moving outwards until they reach the same position as used in the baseline.

The goal of the shaped environments is to reduce the time to accidental completion of the task as much as possible. In the baseline environment, it takes a long time until a completely random agent happens to solve even the first few steps because they are so far from the starting point, so the collected data is almost useless for a long time. In the shaped environments, accidental completion is much more likely to happen early on and therefore more useful data is available at an earlier point in time.

3.3 Investigating Implementation Problems

Both LiSP and LSR have potential problems with their provided implementations (to access the code follow the footnotes at the beginning of Section 3). In order to investigate if they are valid problems and to find out how significant those problems are, the baseline is run once with the original code and once with the issues cleaned up.

LiSP's potential problem is with the data collectors. Its implementation can use different data collectors. The step collector and the path collector perform one (for the step collector) or more (path collector) steps and reset the environment after either the environment reports that the episode is over or after a maximum number of steps was taken in the same episode, so it can work like the **-reset* environments. But, for some experiments, the reset free collector is used. This collector ignores every way to indicate that the environment should be reset and instead just keeps on collecting data. While this is totally fine for the MuJoCo environments (Brockman

et al., 2016) which most reported results in the paper stem from, in the volcano and 2D Minecraft environment, this will mean that the agent cannot continue trying to solve the task once the agent falls into the hole or the entire inventory is full. Then, nothing the agent can do will give useful information anymore. Therefore the corrected version of LiSP uses the path collector and the original implementation uses the reset free collector.

LSR's potential problem is with the performance evaluation. After every episode of training, the current performance of the agent is evaluated, potentially in a different environment than the one the training happens in. The data collected during evaluation is only supposed to be used for evaluation, but the supplied code also uses that data to train the agent further. This means that the agent trains on data it is not supposed to train on and therefore probably will have better results than it is supposed to be. By allowing the agent to train on the evaluation data, it is possible that the results are overfitted to the evaluation environment. The corrected version of LSR does not train on the evaluation data.

Chapter 4

Results

This chapter presents the results of the various experiments described in the previous section. For most LSR experiments one run was performed. For most LiSP and SAC experiments three runs were executed. For those, the most average run was chosen to represent the performance.

Some additional notes regarding the graphs presented in this chapter: All curves for LSR were so erratic that they covered up every other curve and were illegible themselves. Therefore all performance curves for LSR are smoothened with a moving average over 5 training units. Additionally, the performance numbers displayed in the graphs are not in the typical format of one reset per episode since the resets happen very seldom in the described setups. How a training unit looks and how long they take to calculate per algorithm are shown in Table 4.1.

4.1 Volcano Experiments

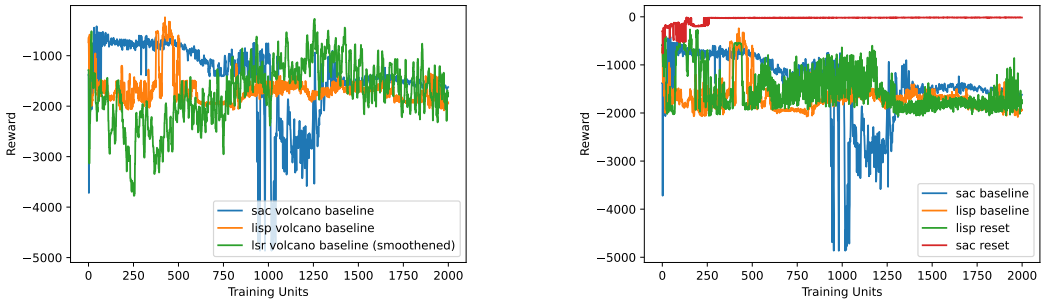
4.1.1 Baseline

As can be seen in Fig. 4.1a, the performances of the three algorithms are very close to each other. While at the very beginning some differences are visible, ultimately all algorithms reach similar levels of performance. Even the peaks are very close to each other.

In terms of stability there are some differences. Regarding the worst achieved performance, LiSP definitely is the best because it never dropped below -2100 performance, not even in the other experiments. LSR is very erratic in its curve, as it is in all experiments. Except for a short period of 300 training units, SAC shows by far the most consistent performance. While it jitters a little bit more than LiSP on occasion, it contains much fewer sudden jumps.

Algorithm	Environment steps per training unit	Training steps per training unit	Computation effort per training unit
SAC	100	10	1 second
LiSP	100	10	19 seconds
LSR	500	25	3.5 seconds

Table 4.1: Comparison of what one training unit means for the different algorithms. Steps per training unit does not contain steps made for evaluation. A training step refers to computing losses on one batch of training data and applying an update step to the weights. Computation effort is measured on one GPU.



(a) Baseline performance on the volcano layout 3.1a.

(b) Performance on the volcano baseline but with frequent resets.

Figure 4.1: VOLCANO BASELINE RESULTS. Pictured are the baseline results in the volcano environment with minimal resets (a) as well as the results with frequent resets (b).

4.1.2 Baseline with Resets

The results for the baseline with frequent resets can be seen in Fig. 4.1b. With resets, SAC rather quickly reaches a level of performance that reflects what a human can achieve. LiSP fails to improve in comparison with the baseline and becomes more unstable.

LiSP's instability is actually quite surprising: In no other experiment does LiSP show such high variance in results, but there are no serious outliers. It still does not go below the -2100 barrier. While the instability is still not that large in comparison to LSR (the magnitude is up to a third as big as unsmoothened LSR), no improvement at all in average performance was definitely not expected.

4.1.3 Randomized Actions

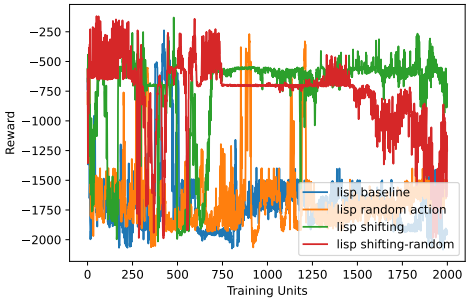
Adding some noise to the actions turned out to have barely any impact at all, as can be seen in the orange curves in Fig. 4.2. While there are some substantial deviations for quite long periods (e.g. for SAC in Fig. 4.2c), ultimately the performance ends up not far from the baseline for all algorithms. For LSR, adding some noise to the actions made the graph by far the most stable of all the experiments.

4.1.4 Shifting Environment

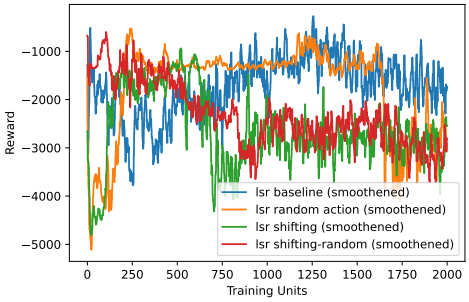
Of all the modifications to the volcano environment, the shifting modification had the most impact (see the green curves in Fig. 4.2). For LiSP and SAC, shifting the hole around resulted in a much better and very stable performance compared to the baseline. LSR, however, was not able to learn from the shifting environment and ultimately had a worse performance than the baseline.

4.1.5 Combination

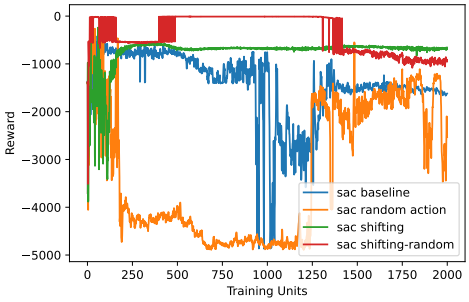
The combination of adding noise to the movement as well as shifting the environment around on average seems to be a little bit worse than only the shifting environment(see red curves in Fig. 4.2). For LSR, the combination has basically no effect in comparison to pure shifting. For both LiSP



(a) Impact of the modifications on LiSP performance.



(b) Impact of the modifications on LSR performance. The curves are smoothened with a 10 training unit wide moving average to make the graph legible.



(c) Impact of the modifications on SAC performance.

Figure 4.2: VOLCANO MODIFICATION RESULTS. Impact on the performance of the modifications to the volcano environment.

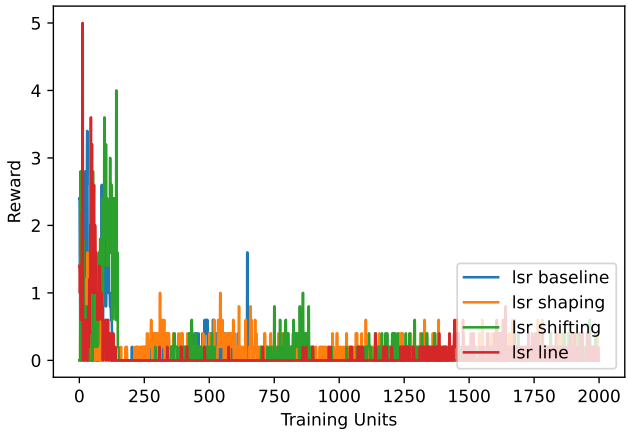


Figure 4.3: MINECRAFT MODIFICATION RESULTS. Pictured are the modification results for LSR in the minecraft environment. SAC and LiSP failed to go beyond a reward of 1 within 80.000 training units of training.

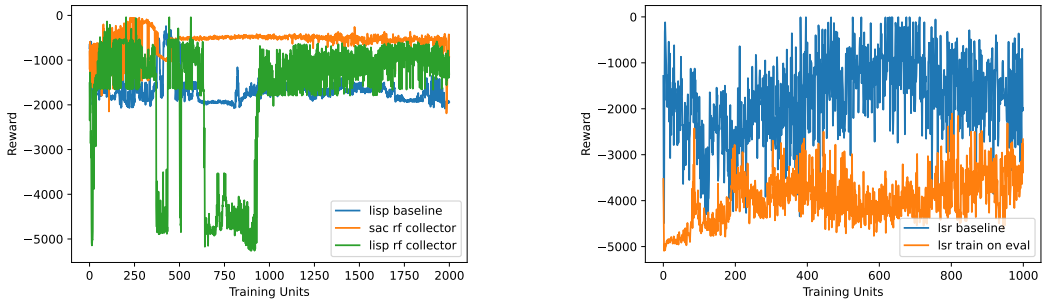
and SAC, performance looks better at the beginning than with any other set of modifications, but at the end drops below the curve with pure shifting. But, they still end up better than the baseline. Also worth noting is that SAC with the combination of shifting tiles and noisy movement is the only algorithm that was able to match the optimal performance that SAC achieves in the setting with resets for a while.

4.2 2D Minecraft Experiments

All algorithms had severe trouble with the 2D Minecraft environment. SAC and LiSP completely failed to learn anything. Occasionally, they accidentally collected one piece of wood (collecting a reward of 1), but never got over that stage. In a very long experiment spanning 80.000 training units they both did not get any further (corrected and original implementation), even after grid-searching over many different parameter combinations. Even when using the exact parameters mentioned in the paper, LiSP never learned anything. Because of that their graphs are not pictured at all.

In the (ultimately unsuccessful) search for parameters with better performance most experiments concerned the learning rates and MPPI configuration since those appeared to be most promising. Even though a learning rate of 3×10^{-4} is used in almost every implementation of SAC going up or down an order of magnitude was tried because learning rates are known to have a large impact on performance. The MPPI configuration (such as planning horizon, sensitivity to model disagreements, or planning iterations) also seemed like a promising place for experiments because MPPI is where the actions to be performed are ultimately chosen. Also worth mentioning is the reward scale: even though LiSP completely ignores rewards during training in favour of the described entropy measures, SAC and the MPPI part of LiSP do consider the rewards. But even scaling the rewards by a factor of 100 was not enough to make the algorithms do what they were supposed to.

LSR had a substantially better performance than SAC and LiSP, as can be seen in Figure 4.3. The performance still can not be described as satisfying (a reward of 30 would mean that the task



(a) Impact of the corrections on LiSP and SAC performance.

(b) Impact of the corrections on LSR performance.

Figure 4.4: VOLCANO CORRECTION RESULTS. Impact on the performance of the corrections to the implementations.

sequence was solved once), but at least some trends are visible. The modifications all offered some small improvements over the baseline for a short time, but performance dropped to the level of the baseline quickly afterwards. Some additional experiments with LSR show that adjusting how quickly the shaping progresses did not improve performance.

4.3 Implementation Corrections

Figure 4.4 shows the effect of the corrections in LiSP's and LSR's code. In both cases, the results are opposite to what was expected.

For LiSP and SAC, the expectation was that performance would improve since more useful information is collected in the baseline. By not using the reset free (rf) collector, the environment is reset when the episode is over (the agent reached the goal or got stuck in the hole) and therefore the agent should be able to collect useful information again. Instead, performance decreased by the correction, and stability was also affected: SAC became much less stable with much better performance, but LiSP's performance decreased slightly but became much more stable. The instability exhibited by LiSP with the rf collector looks similar to the one exhibited in the experiment with frequent resets.

A possible explanation for these results is in how data is collected about stuck agents. With the rf collector, the agents will be trained on lots of data that shows how badly the agent can be stuck in the hole. With the path collector this data is missing because upon falling in the hole the episode is reset. The experience of falling in the hole and receiving a punishment is there, but not for multiple time steps at a time. Therefore the hole will be avoided more in the rf setting. In addition to that, LiSP's implementation of MPPI does not expect episodes ever to be over, so planning can go wrong when MPPI looks too far into the future where no experiences are.

For LSR, the expectation was that performance would decrease with the correction because there is an entire evaluation period of data not being trained on, and that for every training unit. This should result in roughly half as much training happening in the corrected version. By having so much less training, the performance should have decreased. Instead, the corrected version performs much better, even when the amount of training steps is accounted for. A possible interpretation is that the resets cause the policies to have so much exposure to the beginning of the episodes that it fails to learn beyond the beginning since everything else gets drowned out by the

sheer volume of training on starting data. Lowering the learning rates may help with that.

Chapter 5

Discussion

Given the graphs displayed in their original papers, LSR's and LiSP's performance is quite disappointing, especially considering the time they took to train (see Table 4.1). This section discusses the individual experiments, the algorithms, and what can be concluded from the modifications' effect on performance.

5.1 Volcano

The volcano problem turned out to be surprisingly difficult for all agents to learn. Ultimately, only SAC with resets was able to entirely solve the task and SAC with the combined modifications came close to solving it. Such difficulty in solving a task that seems trivial to humans was not expected.

One possible explanation is that the algorithms struggle to explore the domain sufficiently. When resets happen frequently, it is possible to try many different strategies from similar starting positions. In the reset-free setting, this (at least in RL) very common assumption does not hold and if the agents get stuck they simply do not have enough randomness to get unstuck again. This could also explain why SAC with shifting-random was able to do so much better: Because the input to the neural net varies occasionally and because the actions themselves have some added randomness, those almost-sink states are not as hard to break out of.

This theory of exploration through added randomness could help explain what kind of randomness is helpful for learning: If the randomness does not substantially affect what happens, then it is not useful. The added noise in the random action modification is under no condition sufficient to free an agent that is determined to be stuck in a corner. By shifting the environment, however, a large portion of the inputs to the neural network are changed, which is much more likely to provoke a different reaction. This also matches what Co-Reyes et al. (2020) demonstrated: Their randomness gives parts of the environment a probability to take an entirely random action instead of adding some randomness to every action. Their form of randomness improved performance much more.

LSR performed completely differently, however. Its performance actually degraded under the more random conditions. This could mean that LSR contains enough internal randomness generation and is actually hurt by additional randomness from the outside. In contrast with LiSP and SAC, LSR actually contains some internal randomness: The choice of skill to perform is chosen randomly in LSR, whereas SAC does not use skills and LiSP uses its skill-practice distribution, which depends on the state visible to the agent without any source of randomness.

The experiments in the volcano environment show how much RL algorithms can profit from frequent resets. If the explanation of improved exploration through internal randomness holds, it may be necessary to add some of it to algorithms for environments with limited resets. This is a

technique even human brains employ: a part of the brain is responsible for creating noise, which is then used to draw 'inspiration' from (Briggs et al., 2001, p. 255ff). A very simplistic way to do this in RL could be to simply append a few random numbers to the environment state. But, since the algorithms can learn quickly to ignore such an input, probably a more substantial part of the algorithms needs to be controlled by randomness. For LSR, this could mean that not only the reset policy employs randomly chosen skills but also the forward policy needs to respond to some randomly controlled input. LiSP could easily be adapted to use a (partially) random choice of skills instead of the entirely deterministic skill-practice distribution.

5.1.1 2D Minecraft

Because of the terrible performance of all algorithms it is difficult to draw any meaningful conclusions from the 2D Minecraft experiments. What can be seen from Figure 4.3 is that the modifications were useful at the beginning. Especially the shifting environment (green curve) looks promising because it sustained the better performance for a longer time.

Both LSR and LiSP failed to learn useful skills in this environment. LSR occasionally plots what the individual skills do, and after 2000 training units none of the skills moves into the lower half of the grid. Apparently moving into the bottom half was even *unlearned* since earlier plots show that this skill existed already at 100 units. LiSP's skills also only move to the corner without really paying attention to the resources. It is possible that the change in location simply generated too much entropy/intrinsic reward in comparison to picking up an item, which would explain why collecting items was neglected, but still does not explain why LSR only stayed in the upper half of the grid.

It is unlikely that the reward structure interfered with learning skills. Even though the rewards are very sparse in comparison with the volcano environment the rewards only play a minor role for the development of skills. LiSP completely ignores rewards during training and for LSR the rewards only play a minor role.

A potentially critical issue with the intrinsic rewards lies in the nature of the 2D Minecraft environment: The intrinsic reward expects that the resulting state changes from performing a certain skill can be detected anytime the skill was executed. This is, however, not the case in 2D Minecraft: Take the skill of moving to the stone location as an example. A piece of stone can only be acquired when the inventory contains a wooden pickaxe. Otherwise nothing happens when the stone location is reached. Therefore the result of successfully moving to the stone location may only be detected if the inventory contained a wooden pickaxe before the skill was performed. With the current formulation for intrinsic rewards, such conditional skills cannot properly be developed.

5.1.2 LiSP

LiSP showed the most stable performance in comparison with the other algorithms over all experiments together. For example, Figure 4.2a is the only one of the three graphs in Figure 4.2 that does not go down to -5000 reward as a lowest point. On the other hand, it takes a *very* long time to train in comparison with the other algorithms (see Table 4.1). In most situations that additional training time is probably not worth it.

Additionally, LiSP was not even remotely able to reproduce the 2D Minecraft performance indicated in Lu et al. (2021), even though the parameters were as close to the paper's codebase and the paper's appendix as possible. Given that LiSP fails to convince even under those ideal circumstances it is hard to defend LiSP as a solid choice for real-world applications, even though the reasoning behind the code is promising.

Another major drawback of LiSP is the very large number of hyperparameters. The full experiment configurations contain roughly 65 hyperparameters to configure for LiSP, whereas SAC experiments only takes seven of them. Maybe it would be possible to hide some of them in preference of unchangeable defaults like LSR does it, but that would require a lot of work to figure out what values perform well over many different problems.

Besides the many hyperparameters, the LiSP codebase also contains lots of small optimizations that seem vital to the algorithm. For example, the environment model does not predict the next state, but the difference between the current and next state. Given the complex implementation and large number of hyperparameters, it is likely that the hyperparameters are also dependent on such optimizations and therefore even more brittle. According to Figure 4.1, simply changing the frequency of resets already has a strong impact on the stability of LiSP. A completely separate task environment could make the effect even stronger, so it is unlikely that generally near-optimal hyperparameters can be found.

5.1.3 LSR

The most surprising characteristic of LSR in the experiments is its huge variability in the evaluation results. Even though the evaluation of every training unit is averaged over five trials and the other algorithms are evaluated only in one trial, LSR's performance graphs still are by far the most unstable ones. Even the smoothened graphs show an impressive volatility.

This unexplained volatility is a big concern for real-world application of LSR. If just one unit of training can have such a massive impact, any application that needs to be somewhat stable can hardly be justified from a safety perspective.

Another concern for real-world application of LSR is the implementation: The current implementation for the experiments is not one easily configurable module like LiSP but is instead stitched together from DIAYN and SAC in every experiment separately, and lacks obvious points for configuration for rather critical parameters like λ in $J^{forward}$. Even such minor configurations require relatively deep understanding of the underlying code and risk breaking a lot of the implementation.

The reaction of LSR to the modifications is almost opposite to the reactions of the other algorithms. Where SAC and LiSP were able to learn more, LSR mostly did worse and where SAC and LiSP showed no improvement, LSR improved. Because of that, for applications where SAC and LiSP do not perform well it is worth trying how LSR does. The difference most likely comes from the difference in exploration strategies and skill choice since both LiSP and LSR use SAC under the hood.

5.1.4 SAC

While it was not a goal of this thesis to examine the performance of SAC, it was surprising how well SAC compares against LiSP and LSR. SAC is clearly able to hold its ground against the more complex algorithms and it is obvious that SAC is a good choice for a general baseline in reinforcement learning problems.

When taking training time into account (see Table 4.1), SAC definitely performed best. In addition to the solid performance it is also the easiest to configure and available in all codebases that are concerned with RL.

5.1.5 Environment Shaping and Dynamism

Environment shaping turned out to perform not very well in the experiments. Maybe this is because the environment shaping experiments were done in 2D Minecraft and the algorithms

struggled in the entire experimental setup, but it is also possible that the environment shaping simply was not very useful. Multiple experiments with the speed of environment shaping show that the issue with the experiments is not in how quickly the shaping happened.

As already explained above, not all randomness is equally useful. The noise added to the movement commands was not useful, but the slight changes to the problem layout made a major difference in performance. This can be explained with a central piece of theory from Co-Reyes et al. (2020): the difference between the current policy and the optimal policy. The noise added to the movement commands has no substantial impact on the optimal policy. All it does is forcing a small bit of exploration. The shifting environment, on the other hand, modifies the optimal policy by a small, but still significant amount. This means that the policy has to learn to solve the problem in general, but also needs to take the small variations into account to adjust to the correct version that is currently active, therefore becoming more responsive to what is important (analogous to data augmentation in computer vision).

From that, we can draw some conclusions on what kind of environment dynamism improves the results:

- Adding noise to the output of the policy leads only to very minor additional exploration.
- To improve accidental exploration, having a small percentage to take a completely random action is likely more useful to explore the environment. For examples of that, see the experiments in Co-Reyes et al. (2020).
- The most useful dynamism leaves the task mostly the same, but makes some small adjustments to the problem so that the policy needs to respond to them.
- The dynamic elements do not need to be very sophisticated to be useful. *volcano-shifting* consists of a handful of tiny, hard-coded shifts that are cycled through and still produces substantial improvements.

Chapter 6

Future Work

Overall, the experiments show that current RL algorithms struggle with the reset-free setting, which is an important factor for real-world applications. Especially the exploration strategies for environments with little feedback (in terms of both rewards and state change) seem to be a place where substantial improvements could be made. In their current version SAC and LiSP easily get stuck in corners since they have no mechanism like LSR's internal randomness to substantially change behaviours occasionally.

The current entropy-based approach to exploration was clearly shown to be insufficient for certain problems. Both LiSP and LSR were not capable of correctly balancing the entropy awarded from rare changes in inventory content against the more frequent changes in position. It would certainly be possible to adjust how entropy is calculated to encourage certain kinds of exploration, but that would already be very close to reward shaping, which is what unsupervised approaches try to avoid entirely. A compromise could be the ability to mark certain fields in the state as important so that the algorithm can focus primarily on manipulating those.

Another extension the current skill-based algorithms require to become more proficient at general-purpose problem solving is the ability to develop skills that only are executed under certain conditions. In their current state, the intrinsic rewards do not reward the learning of skills that have certain preconditions for them to be impactful.

Lu et al. (2021) claim that using the skill-practice distribution ψ to determine the skill to be performed improves performance of LiSP compared to a random choice. Eysenbach et al. (2018), on the other hand, say that they choose the skill to be performed in DIAYN (which is a part of LSR) randomly because using a different mechanism regularly leads to only very few skills being learned and the others are completely ignored. While the authors do not speak about exactly the same issue (In LiSP it is about which skill has the largest learning potential whereas in LSR it is about which skill has the best performance), resolving the differing positions about random or non-random choice of skills during training could offer substantial improvements to some skill-based approaches.

As it is, environment shaping and environment dynamism look like very promising and relatively cheap improvements to training performance, but the process is still very informal. The list in Section 5.1.5 is a first step towards developing reliable patterns for environment manipulation. Developing a standard set of techniques for environment manipulation could have a very positive impact for real-world application of RL algorithms.

Chapter 7

Conclusion

Two reinforcement learning algorithms for the reset-free setting were looked at in detail: Lifelong Skill Planning (LiSP) and Learning Skillful Resets (LSR). LiSP is a skill-based algorithm which uses an environment model to decide on the skill that will produce the most reward. LSR performs a reset game in which two policies challenge each other to increasingly more difficult challenges in order to learn various useful behaviours.

LiSP and LSR were then evaluated on different scenarios in 2D navigational tasks where they competed against Soft Actor-Critic (SAC). While SAC clearly won in performance achieved per training time, LiSP showed much better worst-case performance, which may be useful in real-world scenarios where certain actions should be avoided. That improved worst-case performance comes at the cost of substantially longer training time, however. LSR exhibited substantially different reactions to the environments than SAC and LiSP did and therefore should be considered as an alternative for tasks where SAC or LiSP do not perform well.

Experiments conducted with different environment behaviours indicate what kind of modifications to the environment may be beneficial for training. Simply adding noise to the agents' outputs was not very beneficial for training. Instead, it is more useful to have a small chance for a random action. The most worthwhile modification in the experiments performed was to move the environment around such that the task stayed mostly the same, but responding to the modification is critical for success.

Chapter 8

Appendix

List of Figures

List of Tables

Bibliography

Briggs, J., Peat, F. D., and Briggs, J. (2001). *Die Entdeckung des Chaos: Eine Reise durch die Chaos-Theorie*. Number 33047 in dtv. Dt. Taschenbuch-Verl, München, seventh edition.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *arXiv:1606.01540 [cs]*.

Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., and Amodei, D. (2017). Deep reinforcement learning from human preferences. *arXiv:1706.03741 [cs, stat]*.

Co-Reyes, J. D., Sanjeev, S., Berseth, G., Gupta, A., and Levine, S. (2020). Ecological Reinforcement Learning. *arXiv:2006.12478 [cs, stat]*.

Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. (2018). Diversity is All You Need: Learning Skills without a Reward Function. *arXiv:1802.06070 [cs]*.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., and Levine, S. (2019). Soft Actor-Critic Algorithms and Applications. *arXiv:1812.05905 [cs, stat]*.

Lu, K., Grover, A., Abbeel, P., and Mordatch, I. (2021). Reset-Free Lifelong Learning with Skill-Space Planning. *arXiv:2012.03548 [cs]*.

Nagabandi, A., Konolige, K., Levine, S., and Kumar, V. (2019). Deep Dynamics Models for Learning Dexterous Manipulation. page 12.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*.

Sharma, A., Gu, S., Levine, S., Kumar, V., and Hausman, K. (2020). Dynamics-Aware Unsupervised Discovery of Skills. *arXiv:1907.01657 [cs, stat]*.

Sukhbaatar, S., Lin, Z., Kostrikov, I., Synnaeve, G., Szlam, A., and Fergus, R. (2018). Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play. *arXiv:1703.05407 [cs]*.

Williams, G., Aldrich, A., and Theodorou, E. (2015). Model Predictive Path Integral Control using Covariance Variable Importance Sampling. *arXiv:1509.01149 [cs]*.

Xu, K., Verma, S., Finn, C., and Levine, S. (2020). Continual Learning of Control Primitives: Skill Discovery via Reset-Games. *arXiv:2011.05286 [cs]*.