Representation Learning for Data-Efficient Deep Reinforcement Learning

by

Trevor McInroe

Thesis Project
Submitted in partial fulfillment of the
Requirements for the degree of

MASTER OF SCIENCE IN DATA SCIENCE

December, 2021

Dr. Alianna J. Maren, First Reader

Dr. Edward Arroyo, Second Reader

Contents

Glossary

BYOL Bootstrap your Own Latent.

CPC Contrastive Predictive Coding.

CURL Contrastive Unsupervised Representations for Reinforcement Learning.

DMC DeepMind Control Suite.

DrQ Data-Regularized Q.

EMA exponential moving average.

KSL k-Step Latent.

MDP Markov decision process.

MoCo Momentum Contrast.

MSE mean squared erred.

MuJoCo Multi-Joint Dynamics with Contact.

RAD Reinforcement Learning with Augmented Data.

RL reinforcement learning.

SAC Soft Actor-Critic.

SAC+AE Soft Actor-Critic + Autoencoder.

SLAC Stochastic Latent Actor-Critic.

SotA state-of-the-art.

TD temporal difference.

Abstract

Deep reinforcement learning (RL) agents that exist in high-dimensional state spaces, such as those composed of images, have interconnected learning burdens. Agents must learn an action-selection policy that completes their given task, which requires them to learn a representation of the state space that discerns between useful and useless information. The reward function is the only supervised feedback that RL agents receive, which causes a representation learning bottleneck that can manifest in poor sample efficiency. In this thesis, we explore representation learning methods for RL in the context of the popular PlaNet benchmark suite, a set of six continuous control tasks with image-based states. We introduce k-Step Latent (KSL), a new representation learning method that provides state-of-the-art (SotA) results in this suite. KSL accomplishes these results by exploiting the environment's underlying Markov decision process via a learned recurrent environment transition model in the latent space. Finally, we spend time analyzing the representations that are learned by KSL and other previous SotA methods. Through this analysis, we uncover a set of desirable characteristics of latent representations as they relate to the RL goal. Using these characteristics, it may be possible to design new representation learning methods in the future that open the door for feasible real-world RL applications.

Introduction

Reinforcement learning (RL) agents that operate in high-dimensional state spaces have two main learning burdens. For one, they must learn a good action-selection policy that allows them to complete their given task. Second, and perhaps most critically, they must learn to discern between useful and useless information in the environment. The standard RL loop ties these two objectives together, which has been shown to cause a representation learning bottleneck (Shelhamer et al. 2017). Learning meaningful information about the state-space in RL is only ever indirect, as the supervised feedback in RL comes from task-oriented reward functions.

The representation learning bottleneck, when combined with the well-known sample inefficiency of deep RL algorithms, makes real-world RL difficult (Kober, Bagnell, and Peters 2013; Zhang and Ma 2020). This inefficiency generally results in the need for an abundance of agent-environment interactions with potentially unsafe or slow-moving components with a high depreciation rate. This property is further exacerbated by the prevalence of sensors on real-world systems that produce high-dimensional data, such as cameras (Breyer et al. 2019; Levine et al. 2017).

Image data are rich with information, and its pixel spaces are high dimensional. Unfortunately, information that is not directly relevant to the task at hand may come in the form of noise to the machine learning system being trained. Ideally, we could convert a full image down to some compressed representation that captures only the axes of variation that are most important for accomplishing our task. To do so, we could learn a function that projects incoming data into a lower-dimensional latent representation. A more information-

dense representation, theoretically, should improve the speed at which a deep RL agent learns. This leads to the following hypothesis:

> *A good **latent** representation of a high-dimensional state space can allow a reinforcement learning agent to learn just as well, or better, as on the **full-dimensional** representation.*

The above hypothesis centers around the word "good". Some modern research in representation learning with images deals with loss functions that encourage accurate data reconstruction. However, it is not clear whether this objective incentivizes the learning of representations that are useful to RL agents. Other streams of representation learning research center around loss functions that operate in the latent space, focusing on encouraging certain properties of the learned projection.

In this thesis, we develop k-step Latent (KSL), a novel model-free approach to representation learning for RL that produces state-of-the-art results in a common benchmark suite of continuous-control tasks. KSL augments the RL process with a representation learning routine that tries to learn a recurrent environment-transition model *in the latent space*. Also, we examine the latent representations that are learned by KSL and compare them to those learned by other modern methods. Our evaluations are both qualitative and quantitative and reveal a series of desirable characteristics of latent representations as they relate to the ultimate RL task.

The remainder of this thesis is laid out as follows. Chapter 1 introduces both reinforcement learning and representation learning. It provides all of the fundamental knowledge that is necessary to understand the remainder of the manuscript.

Chapter 2 presents the body of relevant research work and places KSL in the context of current research streams. First, it presents the area of representation learning independent

from RL. Then, it describes, in detail, Soft Actor-Critic (SAC), an RL algorithm that both KSL and recent SotA methods are built on top of. Finally, it presents the current research work that lies in the intersection of representation learning and RL by introducing the methods that KSL are compared against in Chapter 4.

Chapter 3 introduces KSL. Here, we describe its learning mechanisms and implementation details as well as introduce several ablation studies across design choices. The SAC section of Chapter 2 is key to understanding how KSL fits within the context of the greater RL problem as well as how it interacts with the base SAC learning algorithm.

Chapter 4 provides details on the main experiment in this thesis. It first introduces the PlaNet benchmark suite and describes each of its six tasks. Then, it provides details on the collection and treatment of data for the experiment. Finally, it reveals the performance results that compare KSL to a handful of other prominent methods.

In Chapter 5, we systematically analyze the latent representations produced by various methods, including KSL. Here, we explore their properties and uncover several key characteristics that make for useful representations for the RL task.

In Chapter 6, we summarize our study and its findings. We briefly describe future streams of research and touch on several real-world implications.

Finally, in Chapter 7, we propose several future research directions for KSL.

Chapter 1

Background

Reinforcement Learning

Reinforcement learning is both a sub-field of machine learning and a problem formulation. RL has a rich research history, dating back to its origins in dynamic programming in the 1950s (Bellman 1959), to its emergence as a discipline unto itself (Sutton and Barto 1998; Watkins 1989; Williams 1992), and more recently through its revival as a sub-discipline of deep learning (Mnih et al. 2015; Silver et al. 2018; Haarnoja, Zhou, Abbeel, et al. 2018).

The learning task in RL is to train an autonomous agent to accomplish a given goal over a series of discrete time steps. At each time step, the agent observes its surroundings and chooses an action. Given the agent's chosen action, the environment shifts, and a reward is produced. The reward characterizes the quality of the agent's action as it relates to the agent's goal. Through repeated interaction, the agent learns an action-selection policy that completes its goal via maximizing collected rewards.

Formally, the reinforcement learning problem can be formulated as a finite Markov decision process (MDP) that is parameterized by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma)$, where $\mathcal{S}$ is the state space produced by an environment $\mathcal{E}$, $\mathcal{A}$ is the action space that the agent may sample from, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function that maps states and actions to rewards r, $\mathcal{T} = p(s_{t+1}, r_{t+1}|s_t, a_t)$[1] is a function that explains the state-transition dynamics of $\mathcal{E}$, and

1. For simplicity, we will drop the need for time step notation. E.g., s_t as s and s_{t+1} as s'.

$\gamma \in [0, 1)$ is a discount rate that helps the agent weigh between short- and long-term gains. The agent's goal is to learn an action-selection policy $\pi(a|s)$ that produces a probability distribution over actions conditioned upon a given state. An optimal policy is one that maximizes the sum of discounted rewards $\pi^* = \arg\max_\pi \mathbb{E}_{a\sim\pi, s\sim\mathcal{T}}[\sum_{i=1}^{T} \gamma^i R(s_i, a_i)]$ over the time horizon T of a given task. Figure 1.1, below, depicts the general RL loop described above.

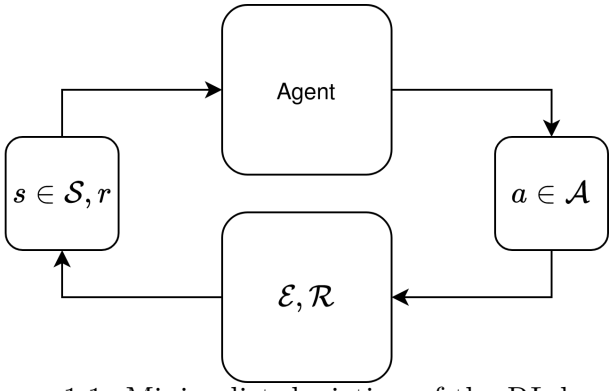


Figure 1.1: Minimalist depiction of the RL loop.

RL algorithms can be characterized by their various attributes such as whether or not they use the transition model of the environment for planning (model-free vs. model-based), whether they use a value function to select actions or directly parameterize a policy (value-based vs. policy-based), and whether they learn from experiences that are generated by the current policy or other policies (on-policy vs. off-policy). Each of these variations is explained in brief in the remainder of this section. A summary of these characteristics can be found in Table 1.1 at the end of this section.

The state-value function $V(s)$ or action-value function $Q(s,a)$ computes the value of a given state or state-action pair[2]. These computed values can then be used to select actions. $V(s)$ can be estimated via the *Bellman equation* $V(s) = \sum_a \pi(a|s) \sum_{s',r'} p(s',r'|s,a)[r + \gamma V(s')]$. We note that this equation is recursive in that the value of the given state s (left-hand term) relies on the value of the next state s' (final right-hand term). It may not be practical to directly solve the Bellman equation, especially when actions or states are continuous, as it would require our agent to visit every possible combination of actions and states. Instead, we oftentimes use temporal difference (TD) methods to estimate value functions. Temporal difference methods estimate the Bellman equation by taking advantage of its backup nature. Specifically, we can repeatedly update the value function for a given state-action pair:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[\overbrace{\underbrace{r + \gamma Q(s',a')}_{\text{TD Target}} - Q(s,a)}^{\text{TD Error}}], \tag{1.1}$$

where α is the usual learning rate. Using this update rule will, little-by-little, move the value of $Q(s,a)$ towards $Q(s',a')$. For a concrete example of why this update rule is desirable, consider the two trajectories $\tau_{success} = \{(s_1, a_1, r_2), ..., (s_{T-1}, a_{T-1}, r_T)\}$ and $\tau_{failure} = \{(s_1, a_1, r_2), ..., (s_{T-1}, a_{T-1}, r_T)\}$. $\tau_{success}$ results in success and therefore has a large reward at r_T while the opposite is true for $\tau_{failure}$. To encourage our agent to act more like it did to generate $\tau_{success}$, we would like our agent to learn $Q(s,a)$ such that high values are given to the tuples (s,a) in $\tau_{success}$ and small values to the tuples in $\tau_{failure}$. Given that the final tuple (s_{T-1}, a_{T-1}) in $\tau_{success}$ has a large reward, we can assign a large value to $Q(s_{T-2}, a_{T-2})$ using

2. $V(s)$ and $Q(s,a)$ are used synonymously in the literature and can trivially be exchanged within algorithms.

the TD Target in Equation 1.1, and continue to backup this discounted value assignment all the way to (s_1, a_1).

This operation need not be done explicitly over a full trajectory. As it turns out, we can estimate this backup operation through sampling various trajectories (Silver 2009; Szepesvári 2010). That is, we can update our policy at the current time step π_t using a policy from a previous time step $(s, a) \sim \pi_{i \neq t}$ or some other policy altogether $(s, a) \sim \pi^{\star}$. Methods that mix trajectories are referred to as *off-policy* while methods that only use trajectories from the current policy are referred to as *on-policy*. The remainder of this thesis will focus on off-policy RL, as it is commonly accepted that their sample efficiency is significantly better than on-policy methods.

In addition, the transition dynamics $p(s', r'|s, a)$ of the environment may not be explicitly known. We can attempt to learn a dynamics function $\hat{p}(\cdot)$ alongside π, or we can formulate the learning problem such that the agent does not need access to $p(\cdot)$. Methods that use $p(\cdot)$ or $\hat{p}(\cdot)$ for planning are referred to as *model-based*. The process of planning uses imagined trajectories (i.e., trajectories that have not happened) from the transition dynamics model to learn $Q(s, a)$. *Model-free* methods are methods that do not perform planning.

The preceding pages have focused on the learning of a state-value function $V(s)$ or an action-value function $Q(s, a)$ as a way of selecting actions. Instead, we may choose to directly parameterize a policy. With this approach, we can learn a function $\pi(a|s)$ that parameterizes a distribution over actions and then sample this distribution using a given state. Methods that learn a value function are referred to as *value-based* and methods that directly learn a policy are referred to as *policy-based*. The division between policy- and value-based methods is less well-defined than the other characteristics described previously. There is a wealth of

modern research in Actor-Critic methods that learn both (Grondman et al. 2012; Haarnoja, Zhou, Hartikainen, et al. 2018).

Attribute	Description
On-policy	π is updated with experience tuples that are generated with the current policy.
Off-policy	π is updated with experience tuples that may not be generated with the current policy.
Model-based	Has access to $p(\cdot)$ or attempts to learn $\hat{p}(\cdot)$ in order to perform planning.
Model-free	Does not perform planning.
Value-based	Parameterizes a value function $V(s)$ or action-value function $Q(s, a)$ to select actions.
Policy-based	Parameterizes a policy directly π to select actions. May also use a value function.

Table 1.1: Summary of defining characteristics of RL algorithms.

Representation Learning

Representation learning deals with the learning of some function Φ that maps data of one dimension to another $\Phi : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$. In neuroscientist David Marr's book, *Vision*, he states that representation learning "makes explicit certain entities and types of information," (Marr 2010). It can allow us to disentangle or uncover information in data that is not obvious in its original form. For example, the classic "kernel trick" through which a toy dataset is made linearly separable once projected from two to three dimensions.

Representation learning is a useful lens through which we can approach deep learning problems. It can be used for the goal of model generalization or simply as a way to transform incoming data to improve the performance of downstream tasks. For example, in the field of computer vision, image augmentation is a common method of data transformation to encourage neural networks to learn a representation of the dataset that is translation invariant.

That is, the models are encouraged to learn an internal representation that allows them to understand objects regardless of their positioning and orientation within the image frame.

Many modern representation learning architectures are made of three main components: an encoder Φ, a module for the main task $\mathcal{T}_{\mathcal{M}}$, and a module for the representation learning task $\mathcal{T}_{\mathcal{R}}$. In the case of image classification, Φ could be a set of convolutional layers that projects a full-color image tensor $x \in \mathbb{R}^{3\times n\times n}$ into a latent-space vector $z \in \mathbb{R}^{1\times j}$. z is then passed to both $\mathcal{T}_{\mathcal{M}}$ and $\mathcal{T}_{\mathcal{R}}$. $\mathcal{T}_{\mathcal{M}}$ could be some classifier such as a dense neural network whose task is to sort z into a set of possible classes. The choice of $\mathcal{T}_{\mathcal{R}}$ is flexible. For example, it could be an auxiliary task that is trained to project zs from the same class close together in j-dimensional space, and vice versa for zs of dissimilar classes. This architecture allows us to accumulate gradients from both tasks in a single forward pass. We can then send both of these gradients backward into the encoder via backpropagation. Doing so allows us to exploit the representation learning task such that z has certain characteristics while also encouraging Φ to create latent representations that are useful for the main task. Figure 1.2, below, depicts this process.

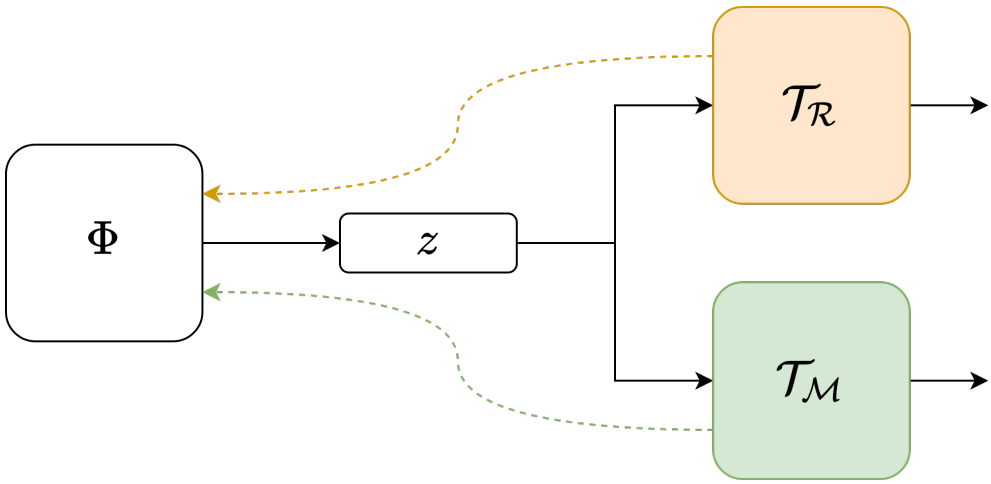


Figure 1.2: Depiction of encoder Φ, representation learning task $\mathcal{T}_{\mathcal{R}}$, main task $\mathcal{T}_{\mathcal{M}}$, and their respective gradients (dotted lines).

When applying this architecture to reinforcement learning, it is not obvious what the representation learning routine should be. The RL task is unique from other machine learning tasks. The feedback signal in RL is continuous and time-based such that our models should understand delayed reward. This begs the question: *"What makes for a good representation, in general?"*. Bengio, Courville, and Vincent (2013) offer several theories on this question.

First, the learned representation function should be smooth. That is, $x \approx y$ implies $\Phi(x) \approx \Phi(y)$. Although we transform data with Φ, we should retain a general consistency of data organization in our new-dimensional space that exists in the original-dimensional space.

Second, the learned representations should be sparse. For any given data point, only a small number of factors are relevant to the task at hand. Bengio, Courville, and Vincent (2013) posit that this characteristic materializes with representations whose entries are often zero-valued.

Third, the learned representations should have temporal and spatial coherence. In general, it is thought that important factors move slowly through time (Becker and Hinton 1992; Mobahi, Collobert, and Weston 2009). Therefore, in accordance with smoothness, the entries in the representations should also move slowly. For example, $\Phi(s_t) - \Phi(s_{t+1}) \leq \Phi(s_t) - \Phi(s_{t+10})$.

Fourth, the learned representations should have a natural clustering that centers around characteristics of interest. For example, in image classification, latent representations should be clearly divided by image class.

Building on the ideas offered by Bengio, Courville, and Vincent (2013), we argue that learned representations are only useful in the context of RL if they relate to the RL

task. RL has two components that differentiate it from other forms of machine learning: the underlying MDP and the reward function. The characteristics of the MDP may be gleaned from temporal coherence, and we postulate that we should have the ability to decipher reward from the learned representations. We explore these ideas and more in Chapter 5.

Chapter 2

Related Works

This chapter contains three sections. First, we introduce relevant literature and ideas from representation learning. Then, we introduce, in detail, Soft Actor-Critic (SAC), the current SotA RL algorithm for continuous control. Understanding the main components of SAC is important as most modern methods for representation learning in RL, including KSL, are built on top of SAC. Finally, we explore the area where the two disciplines of RL and representation learning collide. Here, we spend time on several specific algorithms that provided SotA results in the PlaNet benchmark suite at the time of their publishing.

Representation Learning

Representation learning is a general machine learning concept that spans all data-type domains. In this thesis, we focus solely on representation learning methods that are applicable to image data.

In the absence of large, labeled datasets, representation learning methods for downstream tasks can generally be grouped into two categories: contrastive and generative. Contrastive approaches can be thought of as methods of instance discrimination, whereby representations of positive pairs should be similar while representations of negative pairs should be dissimilar. Generative methods, on the other hand, are focused on the task of learning the underlying latent factors of a given dataset that, when used alone, can allow for the reconstruction of the original data.

One common example of a generative model is autoencoders, of which they are many varieties. These models work by first projecting an image down into a latent representation with an encoder $\Phi : x \rightarrow z$ and then up-scaling the latent representation back into the original image-space with a decoder $\Omega : z \rightarrow \hat{x}$. These two functions are learned jointly by minimizing the pixel-wise distance between the original image and the reconstructed image. Autoencoders can be deterministic (Ghosh et al. 2020) or they may include randomness through variational inference via Bayesian methods (Kingma and Welling 2014) or by being trained on a de-noising task (Vincent et al. 2010). The reconstruction task of the decoder is made easier when the encoder is able to imprint the most important generative factors into the latent representation.

Contrastive methods take many forms and tend to be more popular than generative methods in the current research literature. Oord, Li, and Vinyals (2018) introduced Contrastive Predictive Coding (CPC), an unsupervised representation learning method that uses *InfoNCE*, a loss that operates in the latent space. InfoNCE estimates the mutual information between vectors. Given a target vector q, a positive example vector k_+, and a set of negative example vectors $\{k_{-,i}\}_{i=1}^{N}$, we can learn a projection matrix W that minimizes:

$$\mathcal{L}(W) = \frac{exp(q^\top W k_+)}{exp(q^\top W k_+) + \sum_{i=1}^{N} exp(q^\top W k_{-,i})}. \tag{2.1}$$

CPC creates q and k_+ by encoding augmented versions of a single image and k_- are formed by encoding augmented versions of other images.

Momentum Contrast (MoCo) (He et al. 2020; Chen, Fan, et al. 2020) builds off of CPC by introducing the idea of a momentum encoder. In this paradigm, two versions of the

given encoder are used: an online version Φ_o and a momentum version Φ_m. At each learning step, the same mini-batch of data $\{x_i\}_{i=1}^N$ is passed through both encoders to produce two sets of latent representations $\{z_o^{(i)}\}_{i=1}^N, \{z_m^{(i)}\}_{i=1}^N$. The loss function is formulated such that latent representations of the same image $(z_o^{(i)}, z_m^{(i)})_{pos}$ $\forall i \in N$ form positive pairs, else a negative pair is had $(z_o^{(i)}, z_m^{(j)})_{neg}$ $\forall i \neq j \in N$. The loss is only allowed to propagate through the online encoder and the weights in the momentum encoder are updated as an exponential moving average (EMA) of the weights from the online encoder: $\Phi_m \leftarrow \zeta\Phi_m + (1-\zeta)\Phi_o$ where ζ controls how quickly Φ_m follows Φ_o. This EMA, sometimes called a "soft" update, is a critical component of most representation learning methods used in RL today.

It has long been believed that the negative pairs in contrastive representation learning methods prevent *representational collapse* (Chen, Kornblith, Norouzi, et al. 2020; Chen, Kornblith, Swersky, et al. 2020). A collapse happens when a network converges to a solution where the encoder outputs the same vector for all inputs. However, Grill et al. (2020) introduced Bootstrap your Own Latent (BYOL), a contrastive representation learning method that avoids collapse via other mechanisms. BYOL uses a series of extra modules in the form of small neural networks that continue to project the latent representations into other forms. After z_o and z_m are encoded, both are projected into another form $\bar{z}_o, \bar{z}_m$ via a secondary set of encoders. Finally, a prediction matrix W forms $\hat{z}_o = \bar{z}_o^\top W$. The entire series of networks in the online path are trained to minimize the distance between $\hat{z}_0$ and $\bar{z}_m$: $||\hat{z}_0 - \bar{z}_m||_2^2$. KSL's representation learning module is based on BYOL and uses a similar loss function and is explained in depth in Chatper 3.

Soft Actor-Critic

Haarnoja, Zhou, Abbeel, et al. (2018) introduced Soft Actor-Critic (SAC), an off-policy, model-free RL algorithm for continuous control that leverages both value functions and a parameterized policy. The theory of the model is based on the notion of *maximum entropy RL* (Ziebart et al. 2008), wherein the optimization objective jointly maximizes reward and randomness:

$$\pi^* = \arg\max_{\pi} \sum_{t=1}^{T} \mathbb{E}_{a\sim\pi, s\sim\mathcal{T}}[\overbrace{\mathcal{R}(s_t, a_t)}^{\text{RL}} + \alpha \underbrace{\mathcal{H}(\pi(\cdot|s_t))}_{\text{Entropy}}]. \tag{2.2}$$

$\mathcal{H}(\cdot)$ is a function that measures a given policy's entropy in a given state and α is a temperature parameter that controls the relative importance between the standard RL objective (lefthand term) and the entropy objective (righthand term)[1]. The main interpretation of the above objective is that the agent will find a policy that maximizes reward while acting as randomly as possible.

The maximum entropy framework of RL is well-studied and, in summary, provides four main benefits. First, the entropy term encourages randomness in action selection, which should aid in agent exploration. Second, entropy can act as a regularization term that ultimately can improve convergence and stability during training (Geist, Scherrer, and Pietquin 2019; Viellard et al. 2020). Third, some work argues that the entropy term actually makes the loss landscape "smoother" and, therefore, easier to optimize through (Ahmed et al. 2018). Finally, some recent work has empirically shown that entropy-based RL provides

1. We note that the standard RL objective can be recovered in the limit as $\alpha \to 0$.

robustness to disturbances in reward functions, environment dynamics, and even in the presence of adversarial forces (Eysenback and Levine 2021).

SAC is made of five separate neural networks. While this may seem like a staggering over-parameterization of the problem, each of the networks performs a very specific task and performance would suffer without them. As the name suggests, SAC has two main components: the actor and the critic. The actor's policy π_θ is parameterized by a single neural network θ. The critic's value function Q_{ϕ_1,ϕ_2} is made of two neural networks ϕ_1, ϕ_2 that are initialized independently but trained jointly. The purpose of the ϕ_2 is to mitigate the well-known over-estimation of values in both discrete-action settings (Hasselt, Guez, and Silver 2016) and continuous-action settings (Fujimoto, Hoof, and Meger 2018; Sinver et al. 2014). This is handled by using the minimum of the two during loss computation. Finally, there exists a target critic, parameterized by two more neural networks $Q_{\bar{\phi}_1,\bar{\phi}_2}$. The target critic is used to compute the TD target, as shown in Equation 1.1.

The training of SAC happens in several stages. First, a mini-batch of experience tuples are sampled from a memory bank of past experiences $\tau = \{(s, a, s')_i\}_{i=1}^N \sim \mathcal{D}$. Then, the critic is updated via the squared Bellman error:

$$\mathcal{L}_{critic} = \mathbb{E}_{a'\sim\pi_\theta,(s,a,s')\sim\tau}[(Q_{\phi_1,\phi_2}(s,a) - \mathcal{R}(s,a) + \gamma[\overbrace{Q_{\bar{\phi}_1,\bar{\phi}_2}(s',a') - \alpha \log \pi_\theta(a'|s')}^{\text{// sg}}])^2]. \quad (2.3)$$

The gradient is not allowed to backpropagate through any of the networks underneath the stop gradient (// sg) in Equation 2.3. After the critic is updated, the actor is updated:

$$\mathcal{L}_{actor} = \mathbb{E}_{s\sim\tau,a\sim\pi_\theta}[\alpha\ log\ \pi_\theta(a|s) - \overbrace{Q_{\phi_1,\phi_2}(s,a)}^{\text{// sg}}] \quad (2.4)$$

In a follow-up paper, Haarnoja, Zhou, Hartikainen, et al. (2018) introduced a learnable α:

$$\mathcal{L}_\alpha = \mathbb{E}_{(s,a)\sim\tau}[\alpha - (-\log \overbrace{\pi_\theta(a|s)}^{\text{// sg}} - \mathcal{H}_{target})] \tag{2.5}$$

where $\mathcal{H}_{target}$ is a target entropy hyperparameter that, in practice, is usually set to $-dim(\mathcal{A})$. Finally, after all of the network parameters have been updated, the target critic's weights undergo a soft update: $Q_{\bar{\phi}_1,\bar{\phi}_2} \leftarrow \zeta Q_{\bar{\phi}_1,\bar{\phi}_2} + (1-\zeta)Q_{\phi_1,\phi_2}$.

The actor's output parameterizes a Gaussian $N(\mu, \sigma)$ where μ is a vector of means and σ is the diagonal of the convariance matrix. During interaction with the environment, the agent passes the current state through θ and samples from this distribution for action selection.

Figure 2.1 depicts the data-flow of the five networks involved with SAC. For example, for Equation 2.3, $Q_{\phi_1,\phi_2}(s,a)$ is formed by feeding (s,a) pairs drawn from $\mathcal{D}$ into the networks shown in blue, a' is sampled from a distribution parameterized by feeding s' through the actor's network θ shown in orange, which is then used to create $Q_{\bar{\phi}_1,\bar{\phi}_2}(s',a')$ with the networks shown in green. Equations 2.4 and 2.5 follow a similar pattern.

Representation Learning in Reinforcement Learning

In recent years, representation learning has become a popular approach for improving the performance of deep RL algorithms in image-based state spaces. Most of these approaches draw heavily from representation learning methods covered in the previous section. In general, agents that receive a supervised signal from both an RL loss function and

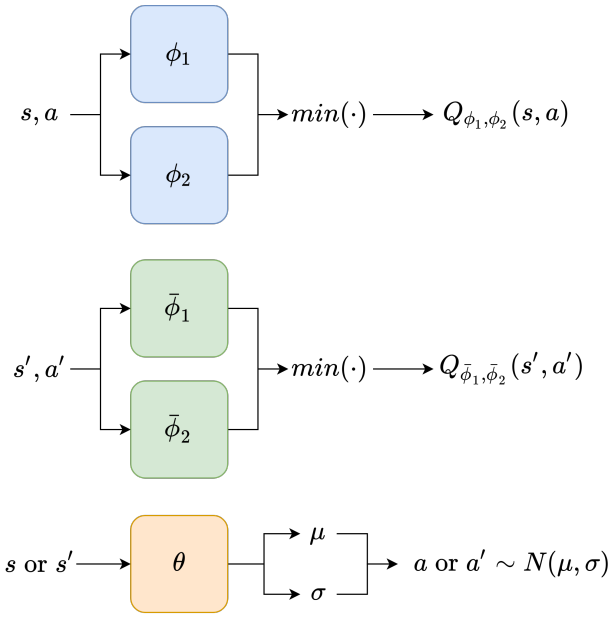


Figure 2.1: Diagram of data-flow through SAC's networks. Critic (blue), critic target (green), and actor (orange).

a representation learning loss function use an alternating training pattern. In the simplest case, the pattern is as follows: at each time step, the agent chooses and performs an action in the environment, draws experience tuples from its memory bank, performs an RL update, and finally performs a representation learning update before returning to the beginning of the loop to execute another action in the environment. The remainder of this section describes several such methods that were, at the time of their publishing, SotA in the PlaNet benchmark.

Contrastive Unsupervised Representations for Reinforcement Learning (CURL) (Laskin, Srinivas, and Abbeel 2020) applies the representation learning modules from MoCo on top of SAC. CURL uses InfoNCE, shown in Equation 2.1. Positive vectors are formed with augmented versions of a given state and negative vectors are formed with augmented versions of other states. After CURL, the same research group released Reinforcement Learning with

Augmented Data (RAD) (Laskin et al. 2020). In this work, they determine the momentum encoder of CURL is unnecessary and that simply training the base SAC algorithm with augmented images is enough to significantly boost performance.

Data-Regularized Q (DrQ) (Yarats, Kostrikov, and Fergus 2021) is similar to RAD with the notable addition of multiple passes through SAC's Q-function. At each step, multiple augmented versions of each state are passed through SAC's critic and averaged together. Both RAD and DrQ have no explicit representation learning mechanism, but instead rely on image augmentation to encourage the learning of useful representations.

Yarats et al. (2020) introduced Soft Actor-Critic + Autoencoder (SAC+AE), which combines SAC with a generative representation learning module via an autoencoder. The reconstruction loss forces the agent's network to organize its internal representation around generative latent factors of the environment. Ideally, this allows the agent to learn its control policy on top of representations that contain more environment-specific information than they would without the auxiliary loss. The main drawback of SAC+AE is the same for all representation learning methods that use a loss function that operates in high-dimensional space: potential large amounts of compute.

Stochastic Latent Actor-Critic (SLAC) (Lee et al. 2020) is a model-free deep RL method that attempts to accelerate sample efficiency through the use of an autoregressive latent variable model (Razavi, Oord, and Vinyals 2019; Maaløe et al. 2019). SLAC's latent variable model is composed of several Gaussian priors and relies on minimizing both a reconstruction loss as well as a loss on an environment transition model. One of the main drawbacks of SLAC is that it requires a significant amount of pre-training. The main results in the SLAC paper are produced by a version of the model that is pre-trained for 50k steps

before any RL training begins. Without any pre-training, SLAC's results drop anywhere from 10% to 55% in the PlaNet benchmark suite.

Chapter 3

k-Step Latent

In this chapter, we introduce k-Step Latent (KSL), our novel approach to representation learning for data-efficient reinforcement learning. KSL builds off of the success of current SotA methods DrQ and RAD. While these two methods rely solely on tricks from computer vision, KSL exploits the main unique characteristic of the RL problem: the underlying MDP. We do so by introducing a representation learning module with a recurrent component that learns temporally-coherent information in the latent space.

KSL's representation learning module is based on BYOL. It uses a self-supervised auxiliary loss that learns to recursively predict the latent representations of future states using the latent representations of current states. At each time step, we sample M trajectories τ of k consecutive-state tuples from a memory bank[1] that include the actions that caused the transition: $\tau = \{(s_1, a_1, s_2, a_2, ..., a_{k-1}, s_k)_i\}_{i=1}^{M}$. For each state in each trajectory, we apply a random translation augmentation (Laskin et al. 2020). This augmentation works by first padding the height and width dimensions of the input image with zeros and then performing a random crop. The ultimate result is a subtle shift of the image. A visual depiction of this process can be seen in Figure 3.1, below. We note that this is the same augmentation used in RAD and DrQ.

After the augmentation occurs, the KSL learning process begins. For a visual depiction of KSL's auxilliary models, see Figure 3.2, below. At step $j = 1$ of trajectory i[2],

1. Trajectories are carefully sampled such that they do not overlap episodes.

2. We omit batch notation in this paragraph for clarity.

(a) 3×84×84 (b) 3×84+Pad×84+Pad (c) 3×84×84

Figure 3.1: Depiction of random translation augmentation steps along with dimensions of each image. Original state (left), padded state (middle), and cropped state (right).

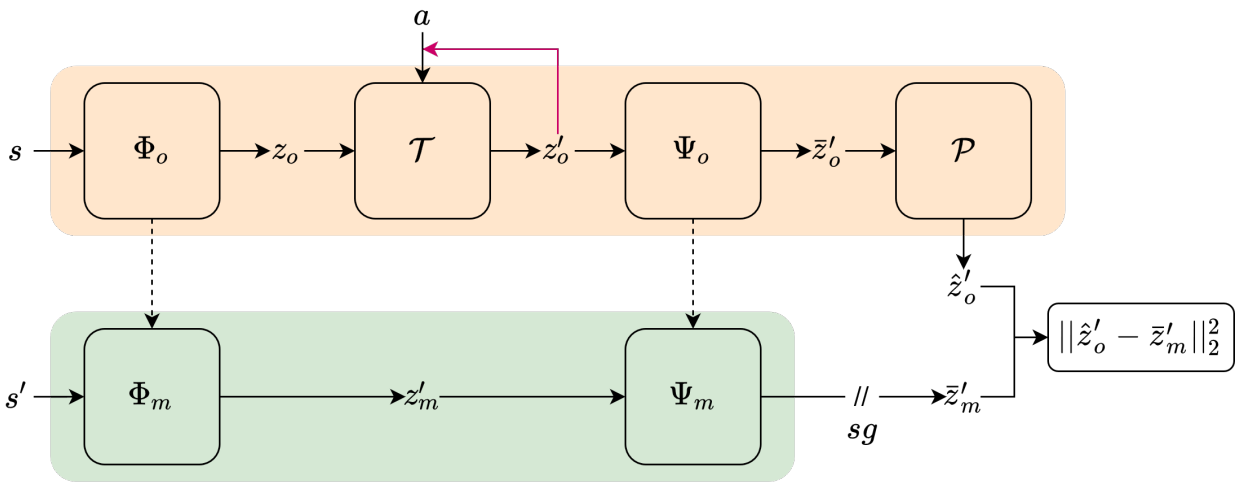


Figure 3.2: Depiction of KSL's modules. Recursive data-flow shown in red. EMA weight updates shown with dotted lines. Current-time-step state s and next-time-step state s' are fed through the online (orange) and momentum (green) path, respectively. Current-time-step action a is concatenated with latent vector z_o for transition module $\mathcal{T}$.

s_j is fed through an online encoder $z_o = \Phi_o(s_j)$ and s_{j+1} is fed through a momentum encoder $z'_m = \Phi_m(s_{j+1})$. Next, z_o is concatenated with a_j and fed through a learned transition model $z'_o = \mathcal{T}([z_o|a_j])$. Then, z'_o and z'_m are fed through their respective projection modules $\bar{z}'_0 = \Psi_o(z'_o)$, $\bar{z}'_m = \Psi_m(z'_m)$. Finally, $\bar{z}'_o$ is fed through a prediction head $\hat{z}'_o = \mathcal{P}(\bar{z}'_o)$. KSL's

loss for step k is computed as the distance between the final two vectors from both paths: $||\hat{z}'_o - \bar{z}'_m||_2^2$.

This operation is performed in a loop over all k steps in τ, with one notable difference at each step after $j = 1$. Instead of encoding every s along the online path, we recursively feed z'_o from the previous step $j - 1$ into the learned transition model $\mathcal{T}$. This is shown with the red arrow in Figure 3.2. The recursive operation forces KSL to learn a temporally consistent representation of the state space *in the latent space*. We form the full loss function as an average over the computed distance for every state for every trajectory:

$$\mathcal{L}_{KSL} = \frac{1}{Mk} \sum_{i=1}^{M} \sum_{j=1}^{k} ||\hat{z}'_{o,(i,j)} - \bar{z}'_{m,(i,j)}||_2^2 \tag{3.1}$$

where the subscript (i, j) indicates trajectory i and step j.

We highlight that a stop-gradient operation is performed on the portion of the computation graph after Ψ_m, as we do not wish for the momentum modules to be updated during backpropagation. Occasionally, the weights of the two modules in the momentum portion of KSL are updated as an exponential moving average (EMA), as depicted by the dotted arrows in Figure 3.2. We note that our method is model-free. The distinction, in this case, is subtle. We do not explicitly use $\mathcal{T}$ for planning but instead use it to encourage our encoders to learn a latent representation that contains temporally-consistent information.

Implementation Details

KSL, just like SAC, has many moving parts and implementation specifics are important. In this section, we describe these key components. RL algorithms are known to

be brittle to hyperparameter settings. As such, we provide a table of hyperparameters in Appendix C.

KSL's online encoder Φ_o and momentum encoder Φ_m are small convolutional neural networks with four convolutional layers and one dense layer with ReLU nonlinearities. The first convolutional layer has a 3×3 kernel and a stride of two. The three following convolutions have a 3×3 kernel and a stride of one. Each convolution outputs 32 feature maps. After the final convolution, the feature maps are flattened and fed through the dense layer, which outputs a vector of length 50. We highlight that this is the exact same encoder architecture used in RAD, DrQ, SAC+AE, and CURL.

These encoders interact with the SAC algorithm. When the agent perceives its environment for action selection, the state s is encoded. The critic and actor networks share an encoder Φ_o and the target critic uses Φ_m. During the RL learning step, the critic's gradient passes through Φ_o but the actor's gradient is stopped before it reaches the encoder. This design has been shown to significantly improve performance (Yarats et al. 2020).

The learned transition module $\mathcal{T}$ is a three-layer dense neural network with 1024, 512, and 50 hidden units, respectively. These layers are separated with ReLU nonlinearities and a LayerNorm operation after the second layer. The projection modules Ψ_0, Ψ_m are two-layer dense neural networks with 512 and 50 hidden units, ReLU nonlinearities, and a LayerNorm operation after the first layer. Finally, the prediction head $\mathcal{P}$ is a single 50×50 matrix.

Just as we use the same encoder architecture as the methods we compare KSL against, we use the same architecture for the SAC networks. All four critic networks are dense neural networks with three ReLU-separated hidden layers with 1024, 1024, and 1 hidden unit, respectively. The agent's network has the same architecture, except the final layer has

$2 \times dim(\mathcal{A})$ hidden units so that it can create the mean vector μ and diagonal σ of the covariance matrix that parameterizes the action-selection Gaussian.

Due to the congruency in architecture, KSL's $\mathcal{T}$ provides an interesting opportunity for knowledge sharing between the RL and representation learning tasks. The KSL learning process is specifically designed to exploit the RL task's underlying MDP, thereby encouraging the learning of representations that directly relate to the RL side. It may be beneficial to allow this information to be shared between the representation learning networks and the networks that directly learn on the RL goal (e.g., the Critic's Q-networks). To do so, we could split $\mathcal{T}$ into two components: knowledge sharing layers and a transition predictor.

After z_o and a have been concatenated, we pass this vector through the knowledge-sharing layers. To allow for the transfer of information, these layers can be made of some of the beginning layers of both of the SAC's online critic's Q-networks. By including some of the Critic's layers, KSL's backpropagation passes through some layers in the Q-networks, thereby causing their weights to be updated via representation learning. This architecture opens the door for a critical design choice. Specifically, how many layers from the Q-networks should be included in the knowledge-sharing procedure?

In addition to choices on knowledge sharing, the choice of k gives practitioners a hyperparameter to tune. We provide a grid search over several values of k and the number of knowledge-sharing layers in Appendix A.

Chapter 4

Experiments

The PlaNet (Hafner et al. 2019) benchmarking suite has become the defacto experiment grounds for comparing deep RL methods for continuous control. This suite includes six tasks that are provided by the DeepMind Control Suite (DMC) (Tassa et al. 2020; Tassa et al. 2018). It uses the Multi-Joint Dynamics with Contact (MuJoCo)[1] physics engine (Todorov, Erez, and Tassa 2012) as a backend and exposes Python-bindings, which makes it extremely convenient to use. Each environment contains an agent made of a series of joints. The agent's goal is to provide some d-dimensional movement vector $a \in [-1, 1]^d$ at each step that moves the agent to complete a task. Each environment provides several different tasks to choose from. It is common to refer to experiments within the DMC by their *environment, task* pair.

Two learning checkpoints of interest have emerged in the research literature. The 100k steps mark is deemed the "data efficiency" checkpoint and it is used as a way to determine how quickly algorithms learn. Also, the 500k steps mark is the "asymptotic performance" checkpoint and it is used as a way to determine the highest level of performance the algorithm can achieve. One common point of confusion in these studies is the difference between algorithm-learning steps and environment steps. Within each environment, task combo, there is a pre-determined number of action repeats per action selection. For example, if a given task has an action repeat of four, each action that is chosen by the agent is repeated

1. http://mujoco.org/

four times, thereby advancing the environment forward four steps. This results in the 100k and 500k checkpoints occurring directly after 25k and 125k action selections, respectively.

Environments

Each task is formulated such that there is a maximum of 1,000 environment steps per episode. To avoid the possibility of the agent memorizing a sequence of actions, each episode begins with the agent-controlled object in a random state, such as randomizing joint angles. Below, we describe the six environment, task pairs in the PlaNet benchmark suite. Table 4.1, below, shows the dimension of the action space as well as the action repeat for each task. Figure 4.1, below, shows a visual depiction of each task in the suite.

The Finger, spin task is made of two components: the "finger" and a rigid plane "spinner" that is fixed in the center. The finger contains a single joint between two rigid planes (e.g., an elbow joint) and is fixed at one end (e.g., a shoulder joint). The agent's goal is to control the joints on the finger to continually spin the spinner. The reward function gives rewards proportional to the length of time of uninterrupted spinning.

The Cartpole, swingup task involves a cart that slides along a horizontal axis and that has an unactuated pole attached to its center. The pole starts each episode pointing downwards and the agent must move the cart such that its momentum swings the pole and balances it in an upward position. The agent's chosen action applies force on the cart in either the left or right direction. The reward function gives rewards proportional to the length of time the pole is balanced upwards.

The Reacher, easy task is made of a two-link arm on a two-dimensional plane. At the beginning of each episode, a red ball is placed randomly on the plane and the agent's goal

is to move the endpoint of the arm to overlap the ball. The reward function gives rewards proportional to the length of time that the ball is overlapped.

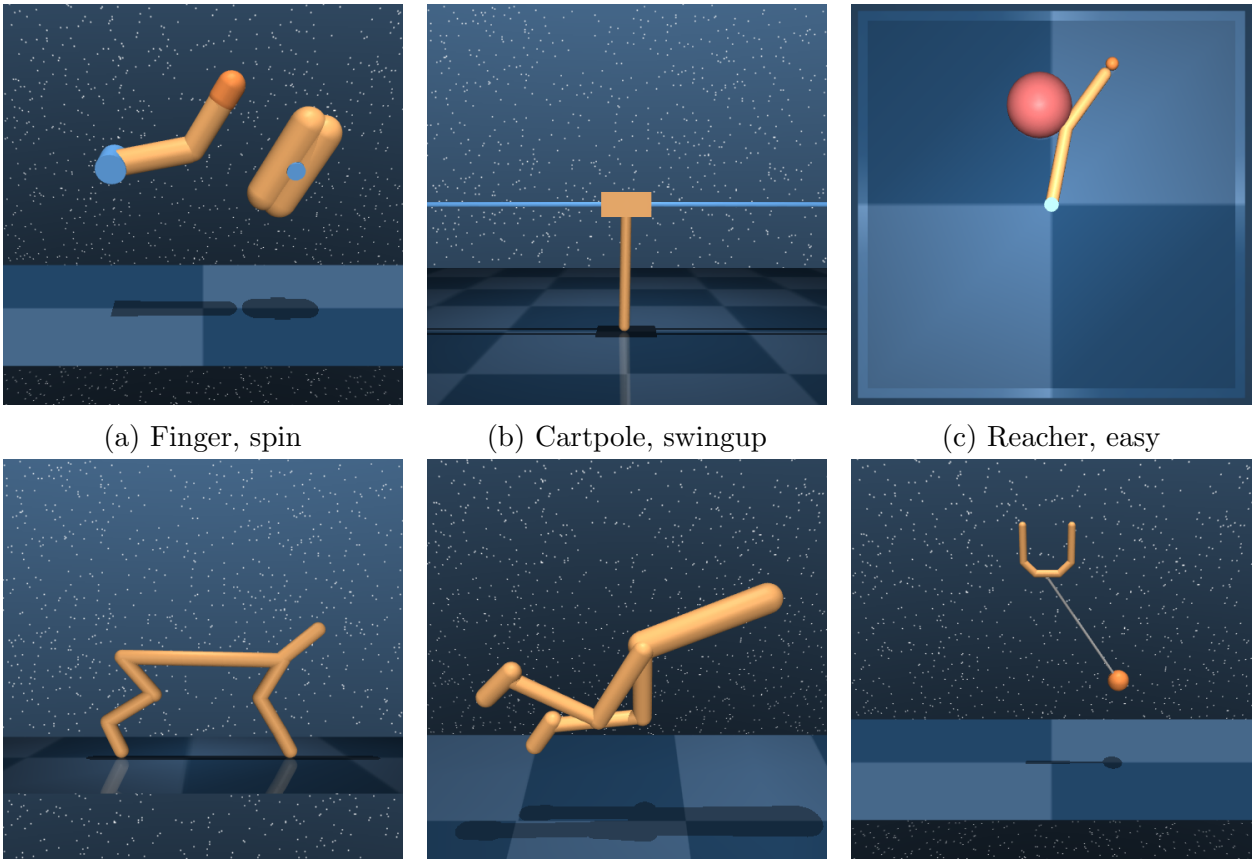
The Cheetah, run task is made of a bipedal creature with a front- and back-leg and six joints. The agent's goal is to control the leg joints in such as way that the body runs to the right as fast as it possibly can. The reward function is proportional to the agent's forward velocity.

The Walker, walk task, first introduced by Lillicrap et al. (2016), is a simplified humanoid with six degrees of freedom. At the beginning of each episode, the Walker starts in a random position, falling to the ground. The agent must control the Walker's joints to first get up and then walk to the right. The agent's goal is to walk as far right as possible. The reward function gives rewards that encourage distance and speed.

The Ball in Cup, catch task is made of two components: a "U" shaped receptacle with a string attached to its bottom and a weighted ball tied to the other end of the string. The agent's goal is to move the cup in such a way that the ball swings up and lands into the cup. The reward function here is sparse in that $r = 1$ when the ball is within the cup, otherwise $r = 0$.

Environment, task	$dim(\mathcal{A})$	Action Repeat
Finger, spin	2	2
Cartpole, swingup	1	8
Reacher, easy	2	4
Cheetah, run	6	4
Walker, walk	6	2
Ball in Cup, catch	2	4

Table 4.1: Dimensions of action spaces and action repeat values and for all six environments in the PlaNet benchmark suite.



(a) Finger, spin (b) Cartpole, swingup (c) Reacher, easy

(d) Cheetah, run (e) Walker, walk (f) Ball in Cup, catch

Figure 4.1: Visual depiction of all six environments in the PlaNet benchmark suite.

Experimental Setup

We adapt the same experimental design from the DrQ, RAD, CURL, and SAC+AE papers. Every 20k environment steps, the agent performs 10 evaluation episodes (no learning occurs) and the final score is the average sum of rewards across all 10 episodes. We run this process five times for KSL, using different random seeds and average together results for each checkpoint. For performance comparison, we include results from models that were SotA at the time of their publishing: RAD, DrQ, CURL, SAC+AE, SLAC, and PlaNet. We also include two control models as a basis for comparison. Pixel SAC is trained on the image-based states without any representation learning routines nor augmentations. State SAC is

trained on the *proprioceptive* states from the environment, which is a vector of ground-truth attributes such as joint angles and velocity. Instead of a convolutional encoder, State SAC simply uses a dense neural network. All algorithms that use images receive states that are made of a stack of three images from the most recent three states.

Results

Table 4.2, below, shows the main results in this thesis. For all currently-existing methods, we use results published in their respective papers. The main result of note is that KSL outperforms all current methods on all tasks for both the 100k and 500k checkpoint while also using smaller batch sizes. RAD, DrQ, CURL, and SAC+AE all report their results with a batch size of 512. In contrast, KSL achieves state of the art results with a smaller batch size of 128. This is of significant note as previous SotA methods suffer large drops in performance with smaller batch sizes. Appendix B shows plots with KSL, DrQ, and RAD with batch sizes of 128 for the Cheetah, run and Walker, walk task. Both RAD and DrQ see significant drops in performance: around 10% in Walker, walk and a 30-45% drop in Cheetah, run.

500k Step Scores	**KSL**	RAD	DrQ	CURL	SAC+AE	SLAC	PlaNet	State SAC	Pixel SAC
Finger, spin	**982 ± 3**	947 ± 101	938 ± 103	926 ± 45	914 ± 107	771 ± 203	718 ± 40	927 ± 43	192 ± 166
Cartpole, swingup	**870 ± 11**	863 ± 9	868 ± 10	841 ± 45	730 ± 152	-	787 ± 46	870 ±7	419 ± 40
Reacher, easy	**978 ± 3**	955 ± 71	942 ± 71	929 ± 44	601 ± 135	-	588 ± 471	975 ± 5	145 ± 30
Cheetah, run	**813 ± 13**	728 ± 71	660 ± 96	518 ± 28	544 ± 50	629 ± 74	568 ± 21	772 ± 60	197 ± 15
Walker, walk	**952 ± 4**	918 ± 16	921 ± 45	902 ± 43	858 ± 82	865 ± 97	478 ± 164	964 ± 8	42 ± 12
Ball in Cup, catch	**975 ± 10**	974 ± 12	963 ± 9	959 ± 27	810 ± 121	959 ± 4	939 ± 43	976 ± 6	312 ± 63
100k Step Scores									
Finger, spin	**939 ± 46**	856 ± 73	901 ± 104	767 ± 56	747 ± 130	680 ± 130	560 ± 77	672 ± 76	224 ± 101
Cartpole, swingup	**839 ± 13**	828 ± 27	759 ± 92	582 ± 146	276 ± 38	-	563 ± 73	812 ± 45	200 ± 72
Reacher, easy	**846 ± 55**	826 ± 219	601 ± 213	538 ± 233	225 ± 164	-	83 ± 174	919 ± 123	136 ± 15
Cheetah, run	**605 ± 28**	447 ± 88	334 ± 67	299 ± 48	252 ± 173	391 ± 47	165 ± 123	228 ± 95	130 ± 12
Walker, walk	**726 ± 111**	504 ± 191	612 ± 164	403 ± 24	395 ± 58	428 ± 74	221 ± 43	604 ± 317	127 ± 24
Ball in Cup, catch	**945 ± 14**	840 ± 179	913 ± 53	769 ± 43	338 ± 196	607 ± 173	710 ± 217	957 ± 26	97 ± 27

Table 4.2: Evaluation results (mean ± one standard deviation) for PlaNet benchmark. Highest mean results shows in bold.

Chapter 5

Evaluating Latent Representations

It is not enough to simply achieve state of the art results. We should also seek to understand *why* certain methods work better than others. If we can develop intuitions around characteristics of good latent representations, it may be possible to design representation learning methods that exploit these characteristics, thereby leading to more principled approaches.

In this chapter, we analyze a handful of qualitative and quantitative characteristics of learned representations. We focus mainly on comparing DrQ and KSL. For some metrics, we also include an SAC agent that is trained directly on the pixels from the environment without any representation learning routines. We refer to this agent as Pixel.

We train each type of agent in the Walker, walk task for 100k environment steps. Every 5k steps, we save a checkpoint of the weights of each agents' encoder. Doing so allows us to observe how the characteristics of the learned latent representations change throughout the training process. We train each of these agents across five random seeds. To create the measurements in the following sections, we collect 10 episodes of state-reward pairs using an already-trained agent. Using a trained agent allows us to gather a wide variety of state-reward pairs such that the rewards span the entire spectrum of possible values.

Organization

One of the mechanisms that make RL different from other machine learning tasks is the use of the reward function. As the reward function is the sole feedback signal in RL, it

follows that our agent's understanding of the environment should be shaped around reward. Figure 5.1, below, depicts this organization of latent representations for one of the runs for both KSL and DrQ.

We first pass each of the collected states through the agents' encoders. Then, we use PCA (Pearson 1901) to project the latent representations into the second dimension for visualization purposes. Finally, these newly projected points are colored according to the reward received in the original version of the states. We perform this process for the encoders from the 5k, 10k, and 15k steps checkpoints.

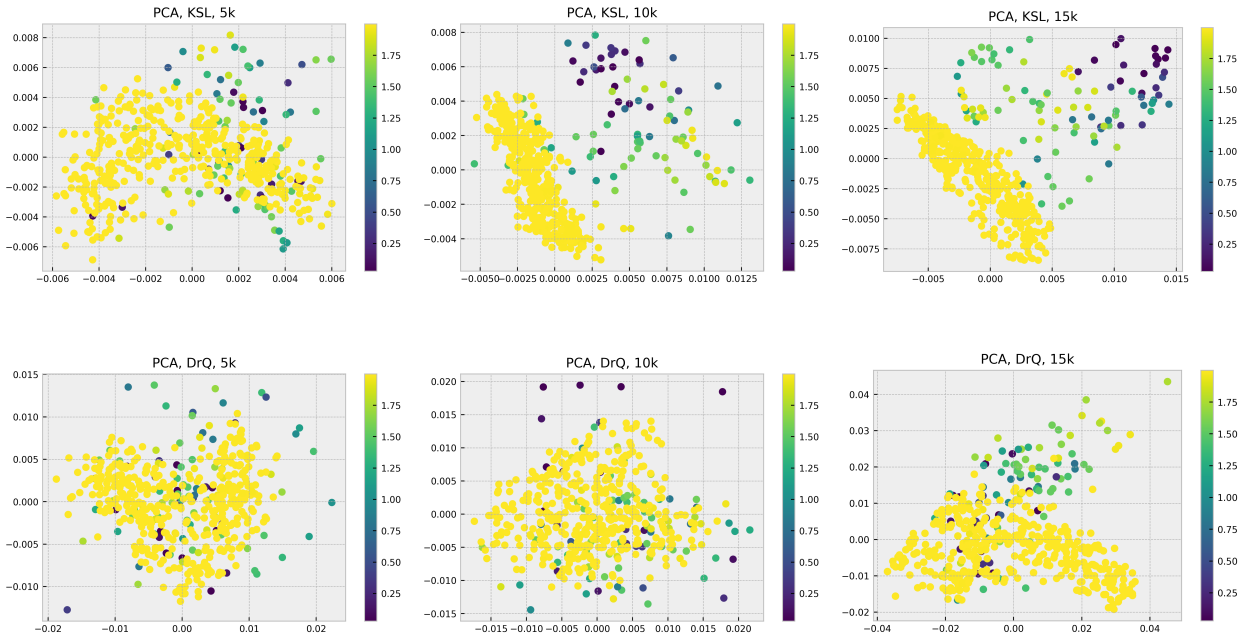


Figure 5.1: Depiction of PCA projections of latent representations, colored by reward, for 5k, 10k, and 15k steps of learning (left to right). Top row is produced by KSL and bottom row by DrQ. We posit that downstream RL tasks are made easier when incoming representations more clearly separate low-reward and high-reward states.

We highlight that KSL's encoders (top row) quickly learn to separate out high-valued states relative to DrQ's encoders (bottom row). By 10k steps (middle column), KSL's

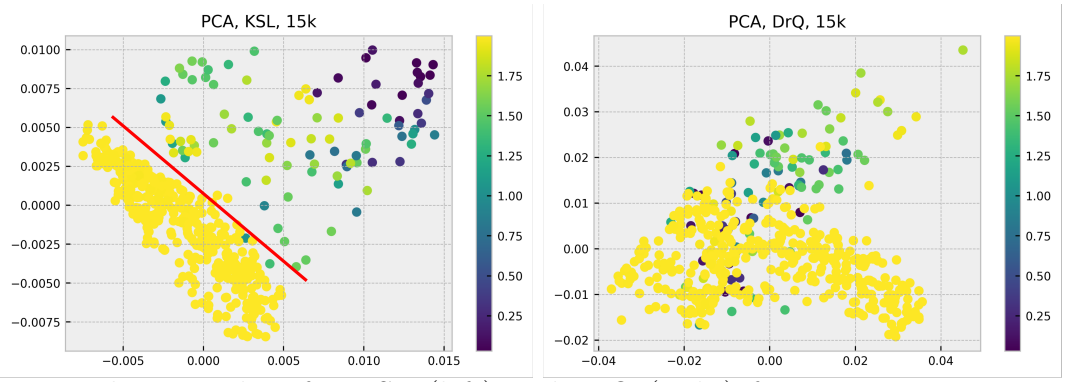


Figure 5.2: 15k steps plots for KSL (left) and DrQ (right) from Figure 5.1. We draw a red line on KSL's plot that separates nearly all high-reward states (yellow) from nearly all low-reward states (non-yellow). In contrast, no such line is possile on DrQ's plot.

encoder has created a visible divide while DrQ's encoder's understanding of the environment is still unclear. Additionally, Figure 5.2 shows the 15k step for either methods. We draw a red line on KSL's plot that demonstrates how KSL's latent representations separate out low- and high-reward states. Nearly all of the highest-reward states (>1.75, shown with yellow points) are to the bottom-left of the line while nearly all other states are to the top-right. In contrast, we could not draw a line on DrQ's plots that would produce a similar division. This can be related to the idea of natural clustering from Bengio, Courville, and Vincent (2013). They theorize that a useful representation should retain meaningful distance between task characteristics of interest. While they only explicitly explore the idea of categorical characteristics, we can plainly see that KSL learns an internal organization around continuous-valued reward.

Another way to quantify the relationship between latent representations and reward is to determine how linearly predictable one is from the other. Specifically, we estimate β, a weight vector that solves the closed-form equation for a linear regression $\beta = (X^\top X)^{-1} X^\top Y$. Here, Y is a vector of rewards and X is a matrix of latent states where Y_i is the reward that

directly corresponds to the ith row of X. Once estimated, we use β to predict a vector of rewards $\hat{Y}$ and measure the mean squared erred (MSE) $\mathbb{E}[(Y - \hat{Y})^2]$. We perform this operation for each encoder checkpoint from 0k steps to 100k steps. Figure 5.3, below, displays the MSE for all three types of agent. The mean of all five runs is shown in bold and $\pm$ one standard deviation is shown with the shaded area. The right plot is a zoomed-in version of the left plot that we use to highlight the difference between KSL and DrQ.

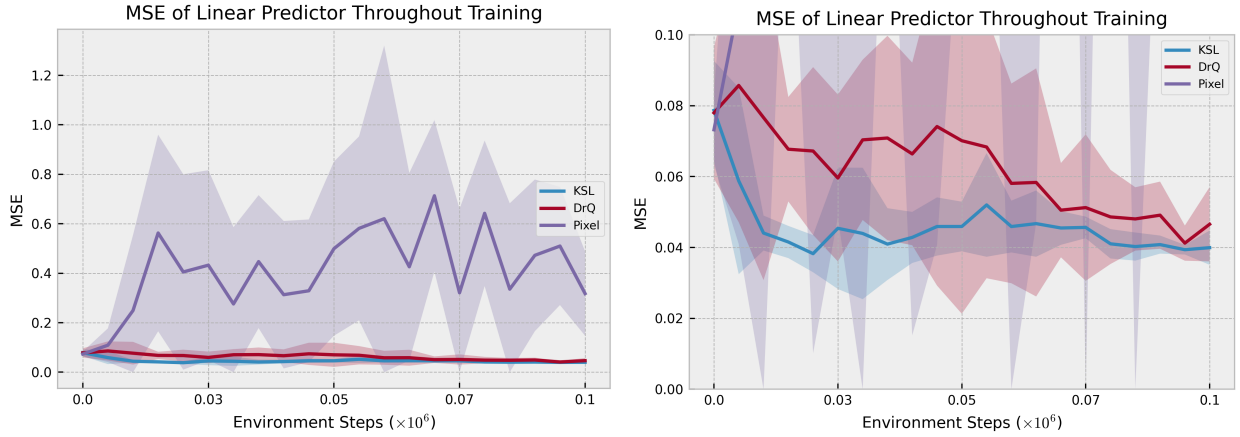


Figure 5.3: "Timeline" of MSE of a linear predictor of reward using latent representations.

We note that simply learning directly on pixels without any representation learning routine results in a diverging trend as well as relatively high variance from checkpoint to checkpoint. When comparing KSL to DrQ, KSL results in a much lower MSE throughout training as well as a significantly lower amount of variance between random seeds. This result, as well as the qualitative results from the PCA plots, suggest that latent representations that directly relate to reward are better for RL.

Sparsity

Bengio, Courville, and Vincent (2013) mention that sparsity is a desirable trait in representations. They define sparsity as elements in the latent representations being zero. We measure this phenomenon by computing the mean value of all z for all ten collected episodes. We collect this value for all training checkpoints from 0k to 100k steps for Pixel, DrQ, and KSL. Figure 5.4 display the mean (bold line) across all five seeds as well as $\pm$ one standard deviation (shaded area). These results show that learning directly on pixels

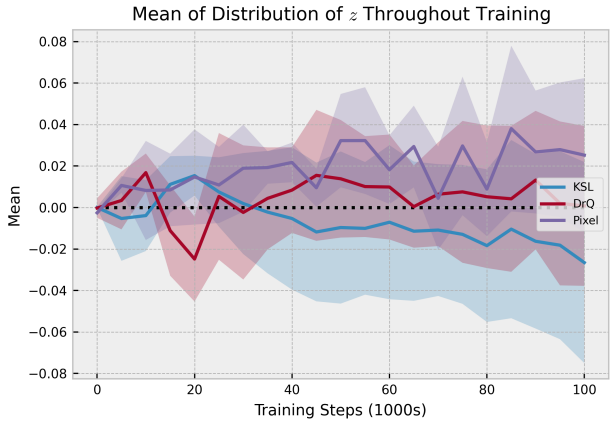


Figure 5.4: Mean of latent representations throughout agent training.

without representation learning routines is enough to achieve sparse representations. We note that there seem to be no material differences between all three methods in the context of sparsity.

We hypothesize the lack of difference between methods may be a result of encoder architecture. The tanh activation after the encoder's final layer forces all entries in the latent vectors to be $(-1, 1)$. This may distort the metric we are capturing in Figure 5.4.

Spatial Attention Maps

The beginning stages of the Walker, walk task involves the Walker falling to the ground in a random position. This often requires the Walker to get itself up off the ground before beginning to walk. Figure 5.5, below, depicts a three-image stack (which represents a single state) of this critical moment.

Figure 5.5: Three-image stack of the beginning stages of the Walker, walk task.

In this section, we examine spatial attention maps from the encoders for both KSL and DrQ throughout training. Figure 5.6, below, is made of three columns, one for each in 5k, 50k, and 100k steps of training. Each of these is made of two columns, one for KSL and one for DrQ. The rows are produced by the first, second, third, and fourth convolutional layers (top to bottom) of the encoders. The attention maps are computed as the mean of the squared output of the layers after they pass through a ReLU nonlinearity. This is the same process as presented by Zagoruyko and Komodakis (2017). These maps visually display the largest-valued activations that are output by the convolutional layers. Spatial attention maps are commonly used as a way to qualitatively determine the portions of images that networks are learning to pay attention to.

We note that the first two layers of KSL's encoders pay the most attention to the most recent of the three images in the state-image stack (furthest right image in Figure 5.5).

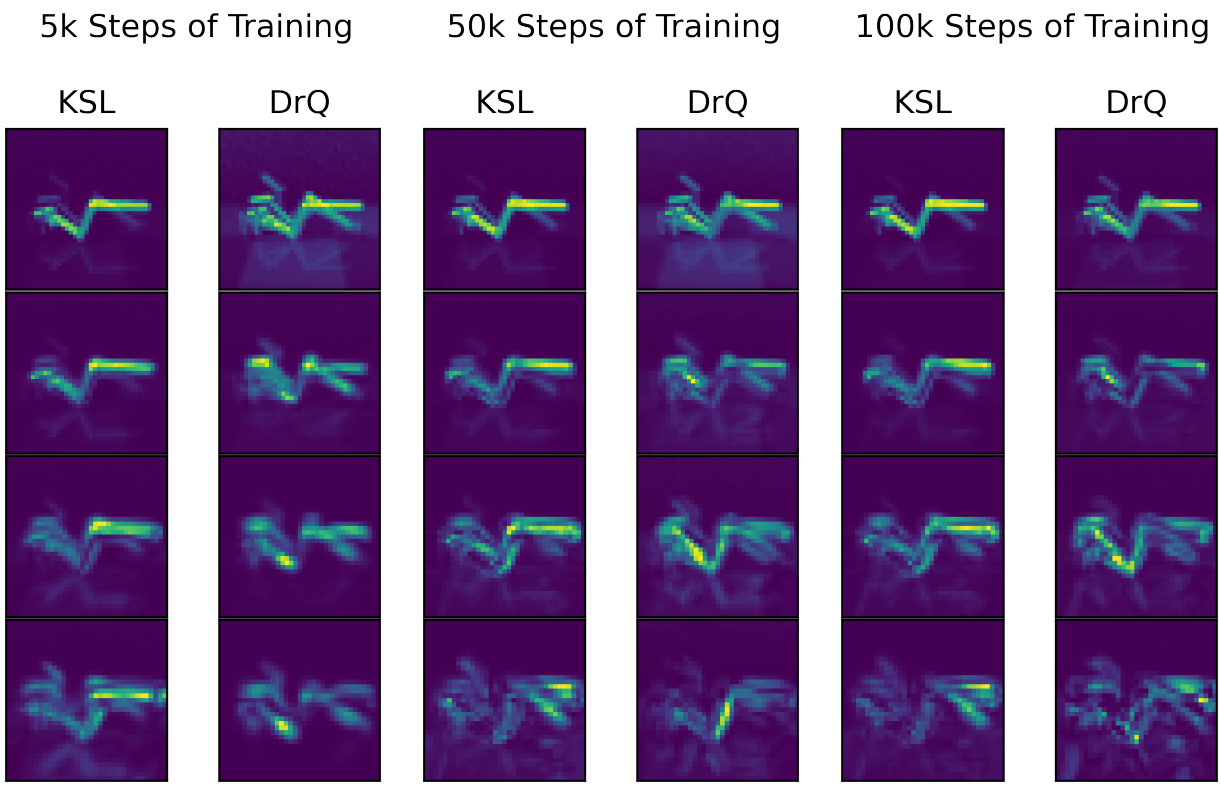


Figure 5.6: Spatial attention maps for KSL's and DrQ's encoders throughout training. The rows correspond to the spatial attention maps produced by convolutional layers with the encoders. For example, the first row is from the first convolutional layer, the second row is from the second convolutional layer, and so on. The first two columns are from 5k steps of training, the middle two columns are from 50k steps of training, and the final two columns are from 100k steps of training.

In contrast, these layers of DrQ's encoders have their attention more spread out amongst all images in the stack. The other main attention map of interest is the one produced by the final convolutional layer (bottom row). Throughout training, DrQ's encoder's attention becomes scattered, losing a clear depiction of the Walker's body. In contrast, KSL's encoder retains a clear and sensible boundary.

Temporal Coherence and Translation Invariance

Bengio, Courville, and Vincent (2013) posit that temporal and spatial coherence is a key characteristic of valuable representations. Specifically, they rely on the idea that important features move slowly over time relative to unimportant features. Therefore, data that are close together in time should also be projected close together in space by a learned representation function. In addition, they mention that a valuable representation should be insensitive to small changes in the original data space. To quantify these two characteristics, we measure the temporal coherence and translation invariance of the learned representations of KSL, DrQ, and Pixel.

We measure translation invariance via the distance between the latent representations of multiple augmented versions of a given state. For each collected state s, we make ten random variations $\hat{s}$ via the translation augmentation described in Chapter 3 and then project these states into a latent representation $\hat{z}$. We then calculate, and average together, the distance between each of these representations: $\frac{1}{10^2-10}\sum_{i=1}^{10}\sum_{j=1}^{10}|\hat{z}_i - \hat{z}_j|_2 \; \forall \; i \neq j$. This process is repeated for all encoder checkpoints for each agent type for each random seed. Figure 5.7, below, shows this metric.

We also measure temporal coherence as the distance between every state and next-state (s, s') for each collected episode. These results are also averaged and collected for each encoder checkpoint for each agent type for each random seed. Figure 5.8, below, shows this metric.

Low values for both metrics can only result when the given vectors being compared are close together. We should desire this trait in our latent representations for several

reasons. For one, low values imply a learned smoothness to the encoder's function $\Phi(\cdot)$. That is, $x \approx y, \quad \Psi(x) \approx \Psi(y)$. If x and y are very similar, such as two consecutive states or augmented versions of the same state, our encoder should project them close together in the latent space. Second, a lower value implies that the learned encoder is robust to perturbations. A small perturbation in the original-data space can be achieved via an image translation or a small movement in the Walker's limbs. Ideally, these minute changes should barely affect an encoder's projection.

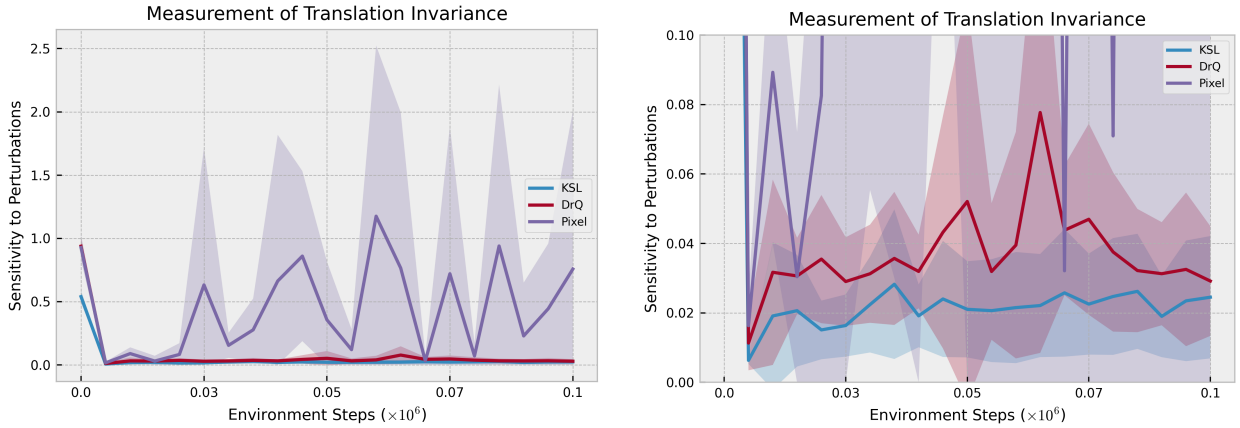


Figure 5.7: Translation invariance metric for KSL, DrQ, and Pixel throughout training. Lower is better.

For both metrics, we can clearly see that the lack of representation learning support causes Pixel to perform poorly and have a high level of variance across random seeds. The zoomed-in version of both plots shows that KSL achieves a lower value than DrQ for both metrics. These results suggest that temporal coherence and translation invariance are valuable latent representation traits for RL agents.

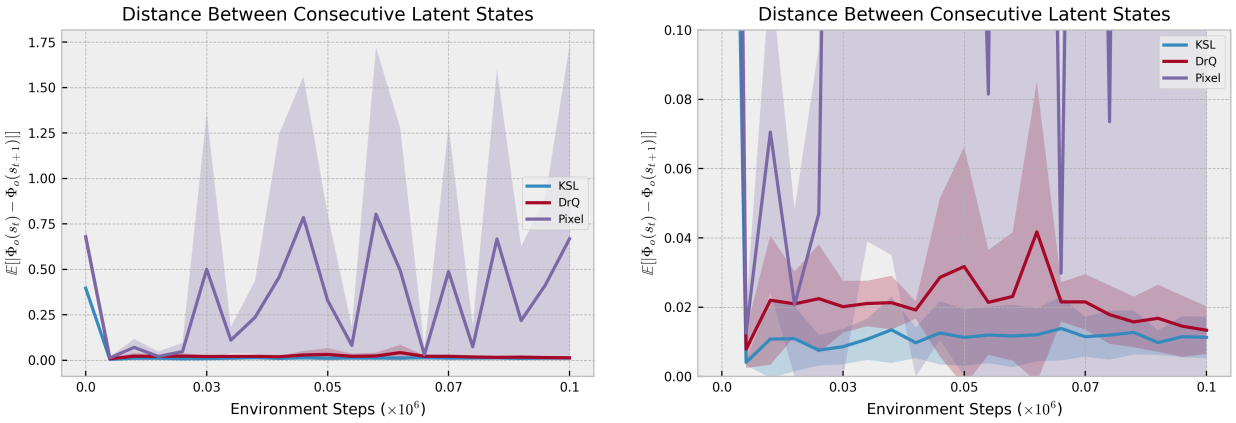


Figure 5.8: Temporal coherence metric for KSL, DrQ, and Pixel throughout training. Lower is better.

Chapter 6

Conclusion

In this thesis, we took on the idea of using representation learning techniques to improve the sample efficiency of deep reinforcement learning (RL) agents in high-dimensional state spaces. We introduced k-Step Latent (KSL), a new representation learning method for RL. KSL learns a recurrent transition model in the latent space by using augmented versions of the environment's states. Using ablation studies, we were able to determine optimal values for k as well as the efficacy of knowledge sharing between the networks used for representation learning and those used for RL. Most importantly, KSL was able to produce state of the art results on both checkpoints for all six tasks in the PlaNet benchmark suite.

Also, we analyzed the learned representations of KSL and its closest competing method, DrQ, and showed that KSL produces representations that have a series of favorable characteristics. First, KSL more quickly learns representations that are cleanly organized around reward. Second, KSL's representation learning routine results in an encoder with more coherent spatial attention. Thirdly, KSL encourages the learning of representation that have a high level of temporal coherence. Finally, KSL encourages the learning of an encoder that is robust to perturbations in the original-data space.

Using the aforementioned analysis, we have arrived at a list of characteristics of a "good" representation. With this list, we can now have a principled way of designing deep RL algorithms with improved levels of sample efficiency. As sample efficiency is the main hurdle between simulation RL and real-world RL, KSL and these characteristics may help future research to leap this obstacle.

Chapter 7

Future Directions

Despite KSL's performance advances, it may present an undesirable increase in compute requirements over other baseline methods. To help save on compute, it may be possible to leverage KSL to pretrain the RL agent's encoder. For example, the agent could collect a small number of steps within the environment, perform a number of KSL updates, and then perform RL training as normal.

Parallel processing may be another option for speeding up KSL. For example, we could create a buffer that is filled with augmented states from which mini-batches are drawn for training. This buffer could be filled in parallel to the computation required for training. Doing so would remove the sequential dependency between replay memory sampling and network training.

Appendix A

Knowledge Sharing and Choice of k

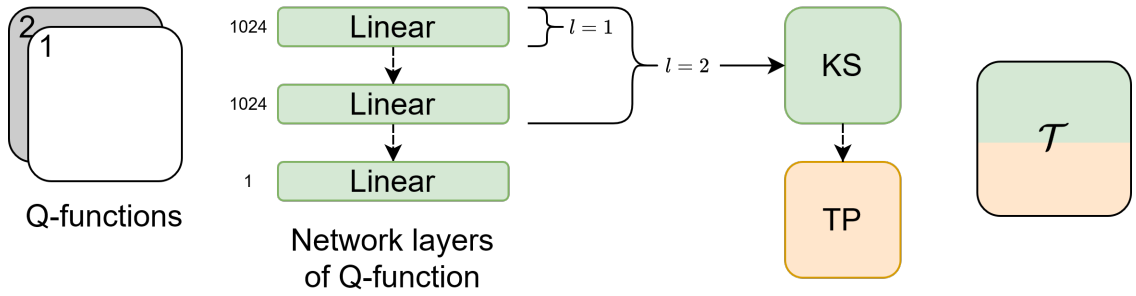


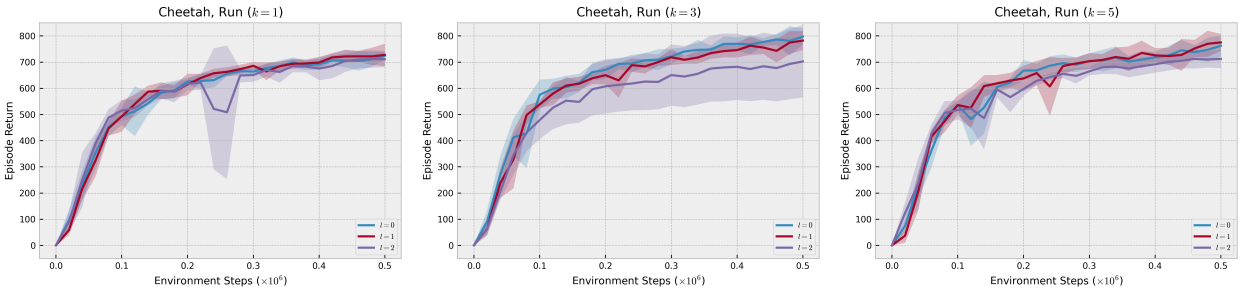
Figure A.1: Depiction of options for knowledge sharing. Critic's Q-functions shown on far left. KSL's transition module $\mathcal{T}$ is shown on far right with its two components: knowlege sharing (KS) and transition predictor (TP). Data-flow is shown with dotted lines.

Figure A.1, above, shows the potential options for the knowledge-sharing (KS) mechanism within KSL. The two Q-functions of the critic are shown on the far left. Although the KS mechanism uses layers from both Q-functions, we only show one in the "Network Layers" section for simplicity. When $l = 1$, only the first layer of the Q-functions is used for KS, and when $l = 2$, the first two layers are used. When $l = 0$, the KS section of KSL's transition module $\mathcal{T}$ is made of a single linear layer that is not part of the Critic's Q-functions.

The choice of k creates a trade off in terms of algorithm performance and compute expense. As the value of k increases, KSL has access to deeper levels of temporal information. However, a larger k results in a larger amount of compute, both in terms of data sampling and the recurrent loop within KSL. With every incremental k, KSL needs to sample and perform the translation augmentation on *batch size* more states from the replay memory. Then, it needs to perform several additional passes through neural network modules as well

as compute a loss. This additional complexity leads to an exponential growth in compute requirements.

We perform a full grid-sweep across all combinations of $k \in \{1, 3, 5\}$ and $l \in \{0, 1, 2\}$ in the task of Cheetah, run. Figure A.2, below, shows the mean (bold line) and one standard deviation (shaded area) across five random seeds. Table A.2, below, summarizes these results.



(a) (b) (c)

Figure A.2: Grid search performance for KSL for $k \in \{1, 3, 5\}$ (left to right) and $l \in \{0, 1, 2\}$ (various lines within plots).

	$l = 0$	$l = 1$	$l = 2$
$k = 1$	493 ± 33	493 ± 59	512 ± 40
$k = 3$	$\mathbf{575 \pm 58}$	538 ± 13	479 ± 73
$k = 5$	535 ± 14	536 ± 37	517 ± 36

Table A.1: 100k steps

	$l = 0$	$l = 1$	$l = 2$
$k = 1$	725 ± 15	728 ± 43	711 ± 31
$k = 3$	$\mathbf{798 \pm 30}$	782 ± 36	703 ± 138
$k = 5$	763 ± 40	775 ± 32	712 ± 35

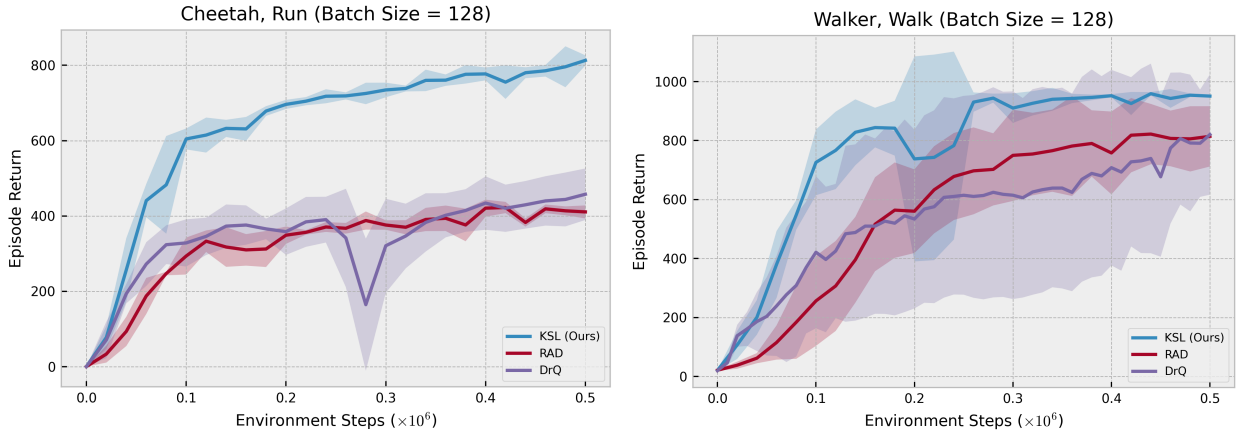
Table A.2: 500k steps

These results show that knowledge sharing provides no benefit and may actually harm the RL learning process. Also, the results show that there are diminishing marginal returns for increasing k. As such, the main results in the paper use $l = 0$ and $k = 3$.

Appendix B

Small Batch Sizes

Reported results in the paper for current SotA methods (DrQ and RAD) are obtained using a batch size of 512. In contrast, KSL is able to outperform DrQ and RAD using only a batch size of 128. Figure A.3, below, shows the mean (bold line) and one standard deviation (shaded area) across five random seeds for each agent type in the Cheetah, run (left) and Walker, walk (right) tasks. Table A.4 summarizes these results.



(a) (b)

Figure A.3: Performance of KSL, DrQ, and RAD at smaller batch sizes in the Cheetah, run (left) and Walker, Walk (right) tasks.

	512	128	Performance Drop
RAD	728	410	43.7%
DrQ	660	328	30.8%

Table A.3: Cheetah, run at 500k steps

	512	128	Performance Drop
RAD	918	814	11.3%
DrQ	921	820	11.0%

Table A.4: Walker, walk at 500k steps

The results above show significant drops in performance for DrQ and RAD as they go from a batch size of 512 to 128. Despite the improvements that DrQ and RAD see with a larger batch size, KSL is still able to outperform them with a batch size of 128.

Appendix C

Hyperparameter Settings

In the below table ϕ refers to all four networks in the critic and target critic. Φ refers to all modules within KSL. There is a large overlap in hyperparameter settings between KSL, DrQ, RAD, CURL, and SAC+AE. This is done on purpose to create as even a comparison as possible.

Hyperparameter	Value
Image padding	4 pixels
Initial steps	1000
Stacked frames	3
Evaluation episodes	10
Optimizer	Adam
$(\beta_1, \beta_2) \rightarrow (\theta, \phi, \Psi)$	(0.9, 0.999)
$(\beta_1, \beta_2) \rightarrow (\alpha)$	(0.9, 0.999)
Learning rate $(\theta, \phi, \Psi, \alpha)$	$2e-4$ Cheetah, run $1e-3$ otherwise
Batch size	124
Q function EMA ζ	0.01
Φ EMA ζ	0.05
Target critic update freq	2
Actor learning freq	2
Critic learning freq	1
KSL learning freq	1
$dim(z)$	50
γ	0.99
Initial α	0.1
Replay memory capacity	100000
Actor log stddev bounds	[-10,2]

Table A.5: Hyperparameters used to produce KSL results.

Bibliography

Ahmed, Zafarali, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. 2018. "Understanding the Impact of Entropy on Policy Optimization." In *Proceedings of the 36th International Conference on Machine Learning (ICML),* 97:151–160.

Becker, Suzanna, and Geoffrey Hinton. 1992. "Self-Organizing Neural Network That Discovers Surfaces in Random-Dot Stereograms." *Nature* 355:161–163.

Bellman, Richard. 1959. *Dynamic Programming.* Princeton, NJ: Princeton University Press.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent. 2013. "Representation Learning: A Review and New Perspectives." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8): 1798–1828.

Breyer, Michael, Fadri Furrer, Tonci Novkovic, Roland Siegwart, and Juan Nieto. 2019. "Comparing Task Simplifications to Learn Closed-Loop Object Picking Using Deep Reinforcement Learning." *IEEE Robotics and Automation Letters* 4 (2): 1549–1556.

Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. "A Simple Framework for Contrastive Learning of Visual Representations." In *Proceedings of the 37th International Conference on Machine Learning (ICML).*

Chen, Ting, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. 2020. "Big Self-Supervised Models are Strong Semi-Supervised Learners." In *34th Conference on Neural Information Processing Systems (NeurIPS).*

Chen, Xinlei, Haoqi Fan, Ross Girshick, and Kaiming He. 2020. "Improved Baselines with Momentum Contrastive Learning," arXiv: 2003.04297 [cs.LG].

Eysenback, Benjamin, and Sergey Levine. 2021. "Maximum Entropy RL (Provably) Solves Some Robust RL Problems," arXiv: 2103.06257 [cs.LG].

Fujimoto, Scott, Herke van Hoof, and David Meger. 2018. "Addressing Function Approximation Error in Actor-Critic Methods." In *Proceedings of the 35th International Conference on Machine Learning (ICML),* 80:1587–1596.

Geist, Matthieu, Bruno Scherrer, and Olivier Pietquin. 2019. "A Theory of Regularized Markov Decision Processes." In *Proceedings of the 36th International Conference on Machine Learning (ICML),* 97:2160–2169.

Ghosh, Partha, Mehdi S. M. Sajjadi, Antonio Vergari, Michael Black, and Bernhard Scholkopf. 2020. "From Variational to Deterministic Autoencoders." In *International Conference on Learning Representations (ICLR).*

Grill, Jean-Bastien, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, et al. 2020. "Bootstrap your Own Latent A New Approach to Self-Supervised Learning." In *34th Conference on Neural Information Processing Systems (NeurIPS).*

Grondman, Ivo, Lucian Buşoniu, Gabriel A.D. Lopes, and Robert Babuška. 2012. "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients." *IEEE Transactions on Systems, Man, and Cybernetics, Part C* 42 (6): 1291–1307.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." In *Proceedings of the 35th International Conference on Machine Learning (ICML),* 80:1861–1870.

Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, et al. 2018. "Soft Actor-Critic Algorithms and Applications," arXiv: 1812.05905 [cs.LG].

Hafner, Danijar, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. 2019. "Learning Latent Dynamics for Planning from Pixels." In *Proceedings of the 36th International Conference on Machine Learning (ICML),* 2555–2565. Long Beach, CA, USA.

Hasselt, Hado van, Arthur Guez, and David Silver. 2016. "Deep Reinforcement Learning with Double Q-Learning." In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence,* 2094–2100.

He, Kaiming, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. "Momentum Contrast for Unsupervised Visual Representation Learning." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR),* 9726–9735.

Kingma, Diederik P., and Max Welling. 2014. "Auto-Encoding Variational Bayes." In *Proceedings of the 2nd International Conference on Learning Representations (ICLR).*

Kober, Jens, J. Andrew Bagnell, and Jan Peters. 2013. "Reinforcement learning in Robotics: A Survey." *The International Journal of Robotics Research* 32 (11): 1238–1274.

Laskin, Michael, Aravind Srinivas, and Pieter Abbeel. 2020. "CURL: Contrastive Unsupervised Representations for Reinforcement Learning." In *Proceedings of the 37th International Conference on Machine Learning,* 119:5639–5650.

Laskin, Misha, Kimin Lee, Adam Stooke, Lerrel Pinto, Pieter Abbeel, and Aravind Srinivas. 2020. "Reinforcement Learning with Augmented Data." In *34th Conference on Neural Information Processing Systems (NeurIPS),* 33:19884–19895.

Lee, Alex X., Anusha Nagabandi, Pieter Abbeel, and Sergey Levine. 2020. "Stochastic Latent Actor-Critic: Deep Reinforcement Learning with a Latent Variable Model." In *Advances in Neural Information Processing Systems (NeurIPS),* 33:741–752.

Levine, Sergey, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. 2017. "Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection." *The International Journal of Robotics Research* 37:421–436.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. "Continuous Control with Deep Reinforcement Learning." In *International Conference on Learning Representations (ICLR).*

Maaløe, Lars, Marco Fraccaro, Valentin Liévin, and Ole Winther. 2019. "BIVA: A Very Deep Hierarchy of Latent Variables for Generative Modeling." In *33rd Conference on Neural Information Processing Systems (NeurIPS).*

Marr, David. 2010. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* Cambridge, MA: The MIT Press.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518:529–533.

Mobahi, Hossein, Ronan Collobert, and Jason Weston. 2009. "Deep Learning from Temporal Coherence in Video." In *Proceedings of the 26th International Conference on Machine Learning (ICML),* 737–744.

Oord, Aaron van den, Yazhe Li, and Oriol Vinyals. 2018. "Representation Learning with Contrastive Predictive Coding," arXiv: 1807.03748 [cs.LG].

Pearson, Karl. 1901. "On Lines and Planes of Closest Fit to Systems of Points in Space." *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science* 2:559–572.

Razavi, Ali, Aaron van den Oord, and Oriol Vinyals. 2019. "Generating Diverse High-Fidelity Images with VQ-VAE-2." In *33rd Conference on Neural Information Processing Systems (NeurIPS).*

Shelhamer, Evan, Parsa Mahmoudieh, Max Argus, and Trevor Darrell. 2017. "Loss is its own Reward: Self-Supervision for Reinforcement Learning." In *International Conference on Learning Representations (ICLR).*

Silver, David. 2009. "Reinforcement Learning and Simulation-Based Search in Computer Go." PhD diss., University of Alberta.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai1, Arthur Guez1, Marc Lanctot1, et al. 2018. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." *Science,* 1140–1144.

Sinver, David, Guy Lever, Nicolas Heess, Thomas Degris, and Daan Wiestra. 2014. "Deterministic Policy Gradients." In *Proceedings of the 31st International Conference on International Conference on Machine Learning (ICML),* 32:387–395.

Sutton, Richard S., and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press.

Szepesvári, Csaba. 2010. *Algorithms for Reinforcement Learning.* San Rafael, CA: Morgan & Claypool Publishers.

Tassa, Yuval, Yotam Doron, Alistair Muldal, Tom Erez andYazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel andAndrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. 2018. "DeepMind Control Suite," arXiv: 1801.00690 [cs.AI].

Tassa, Yuval, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. 2020. "dm_control: Software and Tasks for Continuous Control," arXiv: 2006.12983 [cs.RO].

Todorov, Emanuel, Tom Erez, and Yuval Tassa. 2012. "MuJoCo: A Physics Engine for Model-Based Control." In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems,* 5026–5033.

Viellard, Nino, Tadashi Kozuno, Bruno Scherrer, Olivier Pietquin, Rémi Munos, and Matthieu Geist. 2020. "Leverage the Average: an Analysis of KL Regularization in Reinforcement Learning." In *34th Conference on Neural Information Processing Systems (NeurIPS).*

Vincent, Pascal, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. 2010. "Stacked Denoising Autoencoders: Learning Useful Representationsina Deep Network with a Local Denoising Criterion." *Journal of Machine Learning Research* 11:3371–3408.

Watkins, Christopher. 1989. "Learning from Delayed Rewards." PhD diss., University of London.

Williams, Ronald. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8:229–256.

Yarats, Denis, Ilya Kostrikov, and Rob Fergus. 2021. "Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels." In *International Conference on Learning Representations (ICLR).*

Yarats, Denis, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, and Rob Fergus. 2020. "Improving Sample Efficiency in Model-Free Reinforcement Learning from Images," arXiv: 1910.01741 [cs.LG].

Zagoruyko, Sergey, and Nikos Komodakis. 2017. "Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer." In *International Conference on Learning Representations (ICLR).*

Zhang, Cheng, and Liang Ma. 2020. "A Sample Efficient Model-Based Deep Reinforcement Learning Algorithm with Experience Replay for Robot Manipulation." *International Journal of Intelligent Robotics and Applications* 4:217–228.

Ziebart, Brian D., Andrew Maas, J .Andrew Bagnell, and Anind K. Dey. 2008. "Maximum Entropy Inverse Reinforcement Learning." In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence.*