Characterizing Discrete Representations for Reinforcement Learning

by

Edan Meyer

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

In reinforcement learning (RL), agents learn to maximize a reward signal using nothing but observations from the environment as input to their decision making processes. Whether the agent is simple, consisting of only a policy that maps observations to actions, or complex, containing auxiliary components like value functions and world models, the agent's sole dependence on observations remains invariant. With this fact in mind, the importance of representation becomes clear: changes to the representation of observations affect every part of an agent. Good representations can help an agent learn better policies faster, and bad representations can have the opposite effect.

In this work, we advocate for the use of a form a discrete representations in RL. Through a series of three distinct problem settings in pixel-based Minigrid environments, we incrementally build up to the continual RL setting, where an agent must continually adapt in order to change to best maximize reward. We compare models learned over discrete representation spaces to those learned over continuous representation spaces in each setting, identifying different benefits of discrete representations in each. When learning a model of the world, discrete representations enable more accurate modeling of the world. In episodic RL, policies learned over discrete representations learn faster. And in continual RL, agents learning from discrete representations are quicker to adapt to changes in the environment. In summary, we find that discrete representations enable learning faster and learning better solutions.

Acknowledgements

This thesis was written with tremendous support from Marlos C. Machado and Adam White. Beyond the plentiful amount of feedback and suggestions they offered, their teachings greatly shaped the direction and structure of this research for the better. Their mentorship has been an invaluable asset, and their influence will undoubtedly continue to shape my future research. I am also grateful to Levi Lelis for his participation as a member of my defense committee and for his well-thought-out suggestions on how to improve the work.

Being part of the RLAI community at the University of Alberta as a whole has been a rewarding (pun intended) experience beyond what words can express. I credit this community mainly because the passion and the vision—especially the vision—of people in this community never fails to inspire me. I also credit the community as a whole in part because I received so many comments and so much feedback from so many people that I do not remember all of the conversations. Subhojeet Pramanik, Khurram Javed, and Alex Lewandowski all provided great feedback on multiple occasions, but this is far from an exhaustive list of those that provided feedback. I owe Rich Sutton, in particular, gratitude for showing me RL through a new lens, shifting my view of research, and teaching me how to think big by first thinking small.

My time at as a Master's student at the University of Alberta, and as part of the RLAI community has been, and always will be, unforgettable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work is motivated by the long-standing quest to design an autonomous agent that can learn to achieve goals in its environment without any prior knowledge or human intervention. With an absence of external assistance, the agent's only option is to learn from the world it resides in. This idea is formalized in the field of reinforcement learning (RL) as an agent that receives an observation from the environment, chooses an action based on that observation, and then receives a scalar reward from the environment – the purpose of the agent to maximize the reward [48]. Given that the observation is the agent's sole input when choosing an action (unless one counts the history of reward-influenced policy updates), the representation of the observation plays an indisputably important role in RL. A good representation can facilitate the learning process, while a bad one can break it [11].

The idea of agents making decisions based solely on observations is common in multiple models of autonomous agents. Sutton [47] identifies a common model of intelligent agents (the Common Model) that is shared, or markedly similar, in the fields of psychology, artificial-intelligence, economics, control theory, and neuroscience. LeCun [33] proposes the JEPA model of an autonomous agent that includes many of the same components. In all of these models, observations are the root input to an agent's decision making process, and to nearly every other component that comprises such models. The policy uses observations to choose actions, the value function estimates how "good" a state is based on the current observation, and the world model predicts future observations from the current observation. We point this out not as a novel observation, but to stress the importance of the representation of observations. Changes to observations are the most wide-reaching in the sense that they affect every part of the agent. Perhaps for this

reason, both the Common Model and JEPA share a "perception" module that transforms observations before they reach other components of the agent.

In this thesis, we examine the understudied yet highly effective technique of representing observations as vectors of discrete values (discrete representations) – a method that stands in stark contrast to the conventional deep learning paradigm that operates on learning continuous representations. Despite the numerous uses of learned, discrete representations, the mechanisms by which they improve performance are not well understood, and the only clear demonstration of their usefulness in RL comes from a single result from Hafner et al. [20]. In this work, we dive deeper into the subject and investigate the effects of discrete representations in RL.

The successes of discrete representations in RL date back to at least as early as tile coding methods, which map observations to multiple one-hot vectors via a hand-engineered representation function [48, p. 217-222]. Prior to the proliferation of deep neural networks (DNNs), tile coding was appealing because it provided a way for models to generalize over states with similar properties. Even within the current deep learning paradigm, tile coding seems to improve the learning process by reducing interference between the hidden units of a neural network [17]. Continuous alternatives exist – notably, radial basis functions could be viewed as a generalization of tile coding that produce values in the interval $[0, 1]$ instead of only values of 0 and 1. However, despite the ability of RBFs to represent more total values, tile coding has been shown to generally perform better when the representations contain more than two dimensions [3, 32], which is almost always the case for any complex environment. Discrete representational methods outperforming their continuous counterparts, or at least performing comparably with simpler models, is a trend that can be observed throughout a number of works.

A similar comparison can be seen between the work of Mnih et al. [40] and Liang et al. [35]. Mnih et al. train a DNN to play Atari games, relying on the neural network to learn its own useful representation, or features, from pixels. In contrast, Liang et al. construct a function for producing binary feature vectors that represent the presence of various patterns of pixels, invariant to position and translation. From this representation, a linear function approximator is able to perform nearly as well as a DNN trained from pixels.

While hand-coded functions for producing discrete representations often benefit performance, they either require environment-specific engineering, like in the case of tile coding, or produce many features that may not be useful, like in the representation func-

tion constructed by Liang et al. (which produces millions of features). To circumvent this trade-off, discrete representation-based approaches have recently moved towards learning representation functions, and are finding success in complex, pixel-based domains. van den Oord et al. [54], for example, propose the vector quantized variational autoencoder (VQ-VAE) as a self-supervised method for learning discrete representations. Following the previous trend, VQ-VAEs perform comparably to their continuous counterparts, variational autoencoders [29], and do so while representing observations at a fraction of the size. When applied to DeepMind Lab [8], VQ-VAEs are able to learn representations that capture the core features of observations, like the placement and structure of walls, with as little as 27 bits.

Similar representation learning techniques have also been successfully applied in the domain of RL. Hafner et al. [20] train an agent on Atari, testing both learned, discrete and continuous representations. They find that agents learning from discrete representations achieve a higher average reward, and carry on the technique to a follow-up work [21] where they find success in a wider variety of domains, including the Proprio Control Suite [53], DeepMind Lab, Crafter [19], and Minecraft [18]. Works like those from Robine et al. [44] and Micheli et al. [39] further build on these successes, using discrete representations to learn world models and policies in the RL setting. Work from Wang et al. [57] find that successful representations are often sparse and orthogonal, suggesting that these properties may underpin the success of discrete representation.

To understand whether these discrete representations are beneficial, we must first answer the question: what purpose should these representations fulfill? The answer to this question itself further depends on the problem setting. We already partially define the problem setting as that of RL, but we further specify that the agent is in a world that is vastly larger than itself [30, 51]. In this scenario, the agent cannot hope to perfectly model the world, or even a sizeable portion of it. From the agent's perspective, the world is forever changing, and the agent is forced to continually adapt if it is to best maximize its reward [49]. An agent that adapts quickly, requiring less interactions with the environment, will better achieve this goal. Accordingly, **we aim to learn representations that allow for quick adaptation, and themselves adapt quickly.** These are not the only trait of a good representation, but it is the criterion we focus on in this work.

1.1 Thesis Statement

In the setting where an agent learns in a world far larger than itself, it does not have the modeling capacity to perfectly represent the world. An ideal agent will still learn to represent as much of the world as possible, but that alone is not enough to succeed. The agent must also learn to quickly adapt when it encounters new challenges. Throughout our experiments in pixel-based Minigrid [13] environments, we find that models learned from discrete representations fit these desiderata. Specifically, we claim that **when limited in training data, model capacity, or computation, models learned over discrete representations often find better solutions, and do so faster, than their continuous counterparts.**

Depending on the setting and constraints, the exact benefits of discrete representations manifest in different ways. If our goal is to perfectly model the world, discrete representations allow us to do so with fewer parameters (in the world model). Or conversely, if we wish to learn a world model with capacity insufficient to model the full world, we find that discrete representations allow for more accurately modeling a larger portion of the world.

Despite the increase in modeling capacity, attempts to perfectly model the full world in complex environments (like the real world) remain futile. An ideal agent will quickly adapt to cover its deficiencies with minimal environment interaction and computation. When learning to maximize reward in a changing environment, we find that agents trained with discrete representations more quickly adapt to unpredictable changes and achieve higher reward with less interactions and less policy updates than fully continuous agents. Stated concisely, agents with discrete representations obtain better results with limited training data and computation.

1.2 Approach

We aim to understand the benefits of learned, discrete representations in model learning and RL through a set of experiments conducted in three settings:

1. World model learning.
2. Episodic reinforcement learning.

3. Continual reinforcement learning.

In each setting, we compare function approximators learned on top of discrete representations to those learned on top of continuous representations. The representation functions themselves are also learned, using vanilla autoencoders [6] to learn continuous representation functions and vector-quantized variational autoencoders (VQ-VAEs) [54] to learn discrete ones.

We first examine the effects of learning a world model over a discrete representation space in Section 3, opting to keep the problem simple by using a static dataset in this first set of experiments. We use this setting to confirm that discrete representations increase the accuracy of world models, and to understand the mechanisms that contribute to that superior performance. This includes examining the importance of one-hot encoding as a method of representing discrete information, and understanding the relationship between the size of the function being learned and the usefulness of discrete representations. Performing these supervised learning experiments in the context of dynamics learning also hints that increased world model accuracy may be a contributing factor in the success of Hafner et al. [20].

We transition to reinforcement learning in Section 4, examining whether the previously observed advantages of discrete representations persist amidst shifting input and target distributions inherent to the new problem setting. Upon finding success in the standard RL task, we extend our investigation to continual RL, where the environment changes over time. We again find that discrete representations prevail over their continuous counterparts. Though the benefits manifest in a new form, we rely on our previous supervised learning experiments to draw a unified understanding of the results in each of the settings we explore.

Chapter 2

Background

In this chapter, we introduce the reinforcement learning problem in Section 2.1, policy gradient methods as a solution method in Section 2.2, and the concept of world models along with their role in reinforcement learning in Section 2.6. We discuss the importance of representation learning in this problem setting in Section 2.3, and present autoencoders and FTA as potential solutions in Sections 2.4 and 2.5. Along with a formalization of the autoencoder framework, we introduce a specific type of autoencoder that produces the discrete representations we use in the majority of our work.

Throughout this thesis, scalars and functions are indicated by lowercase letters (e.g., s, f), random variables by uppercase letters (e.g., S, A), vectors by bold lowercase letters (e.g., $\mathbf{e}$, $\boldsymbol{\theta}$), matrices by bold uppercase letters (e.g., $\mathbf{E}$, $\mathbf{Z}$), and sets by uppercase letters with a calligraphic font (e.g., $\mathcal{D}$, $\mathcal{S}$).

2.1 Reinforcement Learning

Reinforcement learning (RL) is a problem setting that formalizes the notion of an agent attempting to achieve some goal in a sequential decision making process. The sequential decision making process is further formalized as a Markov decision process (MDP), which is described by a 5-tuple, $(\mathcal{S}, \mathcal{A}, \mu, p, r)$. The state space of an MDP, $\mathcal{S}$, is the set of all possible states in the environment, the initial state distribution, μ, is the distribution over starting states, and the action space, $\mathcal{A}$, is the set of all possible actions that can be taken from any state. In each episode, the agent starts at state $S_0 \sim \mu$, takes an action $A_t \in \mathcal{A}$, transitions to a new state $S_{t+1} \in \mathcal{S}$, and receives reward $R_{t+1} \in \mathbb{R}$ for each time step

$t = 0, 1, ..., T$. The probability of transitioning between states is given by the transition function $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and the reward for each transition by $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. The agent's goal is to maximize the expectation of the cumulative discounted reward, also called the return:

$$G_t \doteq \sum_{k=0}^{T} \gamma^k R_{t+k+1} \ . \tag{2.1}$$

$\gamma \in [0, 1]$ is a discount factor that determines the importance of future rewards. The agent maximizes the return by learning the parameters for a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ that determines the probability of taking each action.

This work is motivated specifically by the continual RL setting, where an agent must learn continually in order to maximize reward. This could be because the environment is changing, or the environment may be static, but changing from the perspective of the agent who is limited in capacity and cannot learn a model of the full world. In complex, interesting environments like the real world, this is often the case. The agent must adapt quickly to be successful in the continual RL setting, and cannot settle for a single solution.

As we continue to cover other background related to RL, we often discuss agents making decisions based on observations as opposed to states. States are full specifications of the environment at a given time step, whereas observations are the information the agent receives about the state at a given time step. In the partially observable setting, these two items are distinct because the agent may receive observations that only partially describe the state of the environment. When discussing our own work, we often use them interchangeably, as we only deal will environments where the observation contains all the necessary information to best maximize reward.

2.2 Policy Gradient

We use neural networks to learn policies, and train them with policy gradient methods. This is a category of methods that directly parameterize the policy and backpropagate the loss through the policy to learn. One of the most common policy gradient objectives is given by the maximization of

$$\mathbb{E}_\pi \Big[\log \pi_\theta(A_t|S_t)(G_t - v^\pi(S_t)) \Big], \tag{2.2}$$

where θ are the parameters of the policy, and $v^\pi(S_t)$ is the value of state S_t [48, p. 229] . The value of a state is the expected return from that state when following some policy π, $v^\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$. The objective function works by increasing the likelihood of taking action A_t when the return is higher than expected (as indicated by the value function), and decreasing the likelihood when the return is lower than expected. Because the action itself is sampled from the policy, this objective has the effect of encouraging actions that lead to higher returns, and discouraging actions that lead to lower returns.

Typically, the above objective is optimized by collecting batches of agent-environment interactions, performing a gradient update, and then repeating with newly collected data. The same data cannot be used for multiple updates because the objective is an expectation over the behavior of the current policy, and performing an update changes the policy, invalidating the past data. We use proximal policy optimization (PPO) in our experiments, which changes the objective in a way that allows for multiple gradient updates on the same batch of data, increasing sample efficiency and empirical performance [46].

Through several steps, we can transform the previous policy gradient objective into the PPO objective. We first recognize that the right-hand term of the given policy gradient objective is the advantage function

$$a^\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t - v^\pi(S_t)|S_t = s, A_t = a\right], \tag{2.3}$$

the difference between the value of state s, and the expected return if the agent takes action a from state s and then behaves according to π. Rewriting the policy gradient objective with a shorthand for the advantage (not to be confused with an action), we get:

$$\mathbb{E}_\pi \Big[\log \pi_\theta(A_t|S_t) a_t^\pi\Big]. \tag{2.4}$$

To ameliorate the issue of policy updates invalidating previous training data, PPO disincentivizes large changes to the policy. The change in policy is estimated with a ratio between previous and current action probabilities, $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_t|s_t)}$. $\pi_{\boldsymbol{\theta}_{\text{old}}}$ is the policy before any updates with the current batch of data, and $\pi_{\boldsymbol{\theta}}$ is the most recent policy. Large changes are disincentivized by clipping the loss for ratios outside of the range $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter of the method. The full objective is given by the maximization of

$$\mathbb{E}_\pi \left[\min(r_t(\boldsymbol{\theta}) a_t^\pi, \text{clip}(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon) a_t^\pi)\right]. \tag{2.5}$$

The minimum over the clipped ratio removes the incentive for actions to become significantly more or less probable.

2.3 Representation Learning

As discussed in the introduction, the representation of observations fed to an agent is of significant importance. A good representation can expedite the learning process by facilitating generalization and backpropagation, whereas a bad representation can slow learning to a halt. While designers of RL systems may directly modify the environment's observations, environment-specific modifications do not scale. A more general solution is for the agent itself to learn a representation function that transforms observations $\mathcal{S}$ into latent states $\mathcal{Z}$: $f : \mathcal{S} \to \mathcal{Z}$. Rather than learning a policy (or other component, like a value function or world model) over the observation space, the agent then learns a policy over the latent space: $\pi : \mathcal{Z} \times \mathcal{A} \to [0, 1]$.

Fundamentally, deep neural networks learn representations; each layer transforms its preceding layer's output into a new representation of the observation. Although each layer within a policy's neural network forms its own representation, there can be advantages in defining a separate representation function f external to the policy network [7, 10, 14, 31]. By decoupling the policy from the representation function, we introduce the option of training f with an objective distinct from that of the policy (i.e. reward maximization). Such additional objectives aimed at facilitating the learning of useful representations are known as auxiliary objectives [16, 36, 57].

Auxiliary objectives have proven to be notably effective in approaches that align closely with our work. For instance, Hafner et al. [20] propose a successful model-based RL (MBRL) algorithm for Atari [9, 37] games that incorporates an observation reconstruction objective. Without this auxiliary objective, the agent fails to learn entirely. Similarly, Ye et al. [58] enhance an existing MBRL algorithm [45] by introducing an observation self-consistency objective, thereby significantly improving the method's sample efficiency on Atari games.

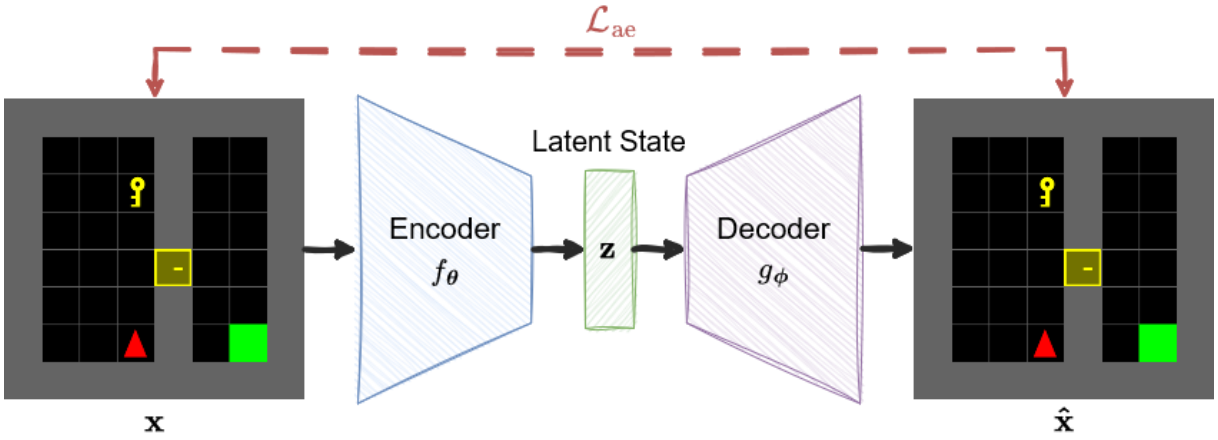


Figure 2.1: Depiction of a vanilla autoencoder with a continuous latent space. The input $\mathbf{x}$ is encoded with $f_{\boldsymbol{\theta}}$ to produce a latent state $\mathbf{z}$, which is decoded by $g_{\boldsymbol{\phi}}$ to produce the reconstruction $\hat{\mathbf{x}}$. The model is trained to minimize the distance between the input and reconstruction with the reconstruction loss $\mathcal{L}_{\text{ae}}$.

2.4 Autoencoders

In this work, we opt to learn representations with autoencoders, neural networks with the objective of reconstructing their own inputs. Autoencoders can be decomposed into an encoder, f_θ, that projects the input into a latent space, and a decoder, $g_{\boldsymbol{\phi}}$, that attempts to reverse the transformation. Where $\mathbf{x} \in \mathbb{R}^n$ is an observation input to the encoder, the corresponding latent state is given by $\mathbf{z} = f_{\boldsymbol{\theta}}(\mathbf{x}) \in \mathbb{R}^k$, and the goal is to learn parameters $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ such that $g_{\boldsymbol{\phi}}(f_{\boldsymbol{\theta}}(\mathbf{x})) = \mathbf{x}$. We achieve this by minimizing the squared error between the input and the reconstruction over observations sampled from some dataset, $\mathcal{D}$:

$$\mathcal{L}_{\text{ae}} = \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[||\mathbf{x} - g_{\boldsymbol{\phi}}(f_{\boldsymbol{\theta}}(\mathbf{x}))||_2^2\Big]. \tag{2.6}$$

Because the latent space of an autoencoder is constrained (generally by size, and sometimes by regularization), the model is encouraged to learn properties of the input distribution that are the most useful for reconstruction. We refer to this type of autoencoder, where the latent states are represented by vectors of real-valued numbers, as a vanilla autoencoder. An overview of the model is depicted in Figure 2.1.

To learn discrete representations, we use an autoencoder variant called a vector quantized variational autoencoder (VQ-VAE) [54]. VQ-VAEs also use an encoder, a decoder, and have the same objective of reconstructing the input, but include an additional *quan-*

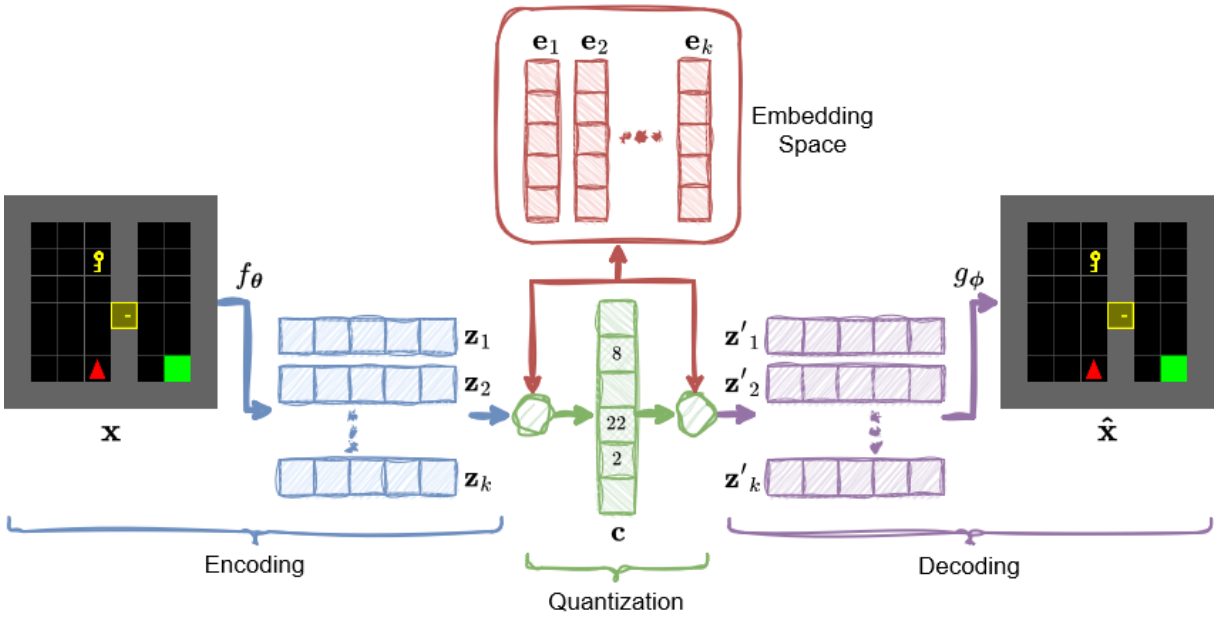


Figure 2.2: Depiction of the VQ-VAE architecture. The input $\mathbf{x}$ is encoded with encoder $f_{\boldsymbol{\theta}}$ to produce latent vectors $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_k\} \in \mathbb{R}^d$. In the first green circle, each latent vector is compared to every embedding vector to produce codebook $\mathbf{c}$, a vector of indices indicating the most similar embedding vectors (example values are depicted). In the second green circle, the indices are transformed into their corresponding embedding vectors to produce quantized vectors $\{\mathbf{z}'_1, \mathbf{z}'_2, \ldots, \mathbf{z}'_k\} \in \mathbb{R}^d$. The quantized vectors are then decoded by $g_{\boldsymbol{\phi}}$ to produce the reconstruction $\hat{\mathbf{x}}$. Our work uses one-hot encodings of the codebook $\mathbf{c}$ as discrete representations.

tization step that is applied to the latent state between the encoder and decoder layers. After passing the input through the encoder, the resultant latent state $\mathbf{z}$ is split into k latent vectors of dimension d: $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_k\} \in \mathbb{R}^d$. Each latent vector is quantized, or "snapped", to one of l possible values specified by a set of embedding vectors. The quantization function uses l embedding vectors of dimension d, $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_l\} \in \mathbb{R}^d$, which are learned parameters of the VQ-VAE.

The quantization happens in two phases. First, each latent vector is compared to every embedding vector using the L2 norm, and indices of the most similar embedding vectors are returned:

$$c_i = \arg\min_j \|\mathbf{z}_i - \mathbf{e}_j\|_2, \text{ for all } i = 1, 2, ..., k. \tag{2.7}$$

The resultant vector of integers $\mathbf{c}$ is called the *codebook*, and indicates which embedding vectors are the most similar to each latent vector. In the second phase, the indices in the

codebook are used to retrieve their corresponding embeddings, producing the quantized latent vectors:

$$\mathbf{z'}_i = \mathbf{e}_{c_i}, \text{ for all } i = 1, 2, ..., k. \tag{2.8}$$

The quantized vectors $\{\mathbf{z'}_1, \mathbf{z'}_2, \ldots, \mathbf{z'}_k\} \in \mathbb{R}^d$ are the final output of the quantization function, and are concatenated before being passed to the decoder. The full architecture is depicted in Figure 2.2.

Because the quantization process is not differentiable, a *commitmen loss* is added to pulls pairs of latent states and their matching embeddings towards each other. If latent vectors are always near an existing embedding, then there will be minimal difference between all $\mathbf{z}_i$ and $\mathbf{z'}_i$, and we can use the straight-through gradients trick [12] to pass gradients directly back from $\mathbf{z'}$ to $\mathbf{z}$ with no changes. Combining the reconstruction and commitment losses, the full objective is given by the minimization of

$$\mathcal{L}_{\text{vqvae}} = \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\left[||\mathbf{x} - g_{\boldsymbol{\phi}}(q_{\mathbf{e}}(f_{\boldsymbol{\theta}}(\mathbf{x})))||_2^2 + \beta \sum_{i=1}^{k} \|\mathbf{z}_i - \mathbf{e}_{\mathbf{z}_i}\|_2^2\right], \tag{2.9}$$

where $q_{\mathbf{e}}$ is the quantization function, β is a hyperparameter that weights the commitment loss, and $\mathbf{e}_{\mathbf{z}_i}$ is the closest embedding vector to $\mathbf{z}_i$. In practice, the speed at which the encoder weights and embedding vectors change are modified separately by weighting the gradients of both modules individually.

The discrete representations we use for downstream tasks RL tasks are different from the quantized vectors that are passed to the decoder. We instead use one-hot encodings of the values in the codebook:

$$o_{ij} = \begin{cases} 1 & \text{if } j = c_i, \\ 0 & \text{otherwise} \end{cases} \quad \text{for } j = 1, 2, \ldots, l. \tag{2.10}$$

The result is a series of one-hot vectors $\{\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_k\} \in \mathbb{R}^l$ that represent a single state, which we refer to as a discrete representation.

2.5 Fuzzy Tiling Activations

Fuzzy tiling activations (FTA) [42] are a form of activation function for neural networks that produce sparse outputs. When applied to a hidden layer of an a neural network,

the intermediate representations produced by FTA have been shown to help learn better solutions and encourage generalization in RL [38, 42, 57]. We use FTA representations as a baseline in part of our work because the outputs they produce are similar to the one-hot, discrete representations produced by VQ-VAEs, and because of their strong, empirical performance.

We explain how FTA works by first explaining *tiling activations*, and then progressing to FTA. Tiling activations are a simpler version of FTA that convert single scalars into one-hot vectors. Given lower bound $l \in \mathbb{R}$ and upper bound $u \in \mathbb{R}$ on inputs to the activation function, tiling activations construct a k-dimensional tiling vector that specifies k bins, each of size $\delta = (u - l)/k$:

$$\mathbf{c} \doteq (l, l + \delta, l + 2\delta, \ldots, u - 2\delta, u - \delta). \tag{2.11}$$

A tiling activation maps each input to a bin, producing a one-hot vector. The tiling activation function is given by

$$\boldsymbol{\phi}(z) \doteq 1 - I_+(\max(\mathbf{c} - \mathbf{1}z, \mathbf{0}) + \max(\mathbf{1}(z - \delta) - \mathbf{c}, \mathbf{0})), \tag{2.12}$$

where I_+ is an indicator function that returns 1 for positive inputs and 0 otherwise, and $\mathbf{1}$ and $\mathbf{0}$ are k-dimensional vectors of all ones and zeros. Both the indicator function and max are applied element-wise and return vectors. When used as the activation function for the layer of a neural network, the tiling activation is applied element-wise. Note that the function maps scalars to vectors, so a tiling activation applied to a j-dimensional hidden layer would produce a (jk)-dimensional output.

Tiling activations offer a way to obtain discrete representations, but come at the cost of having almost entirely zero-valued gradients. FTA extends the idea of tiling activations, but mitigates the zero-valued gradient problem by generating sparse, instead of fully one-hot, vectors. To obtain the new FTA definition, only the indicator in Equation 2.12 function needs to be changed:

$$I_{\eta,+}(x) \doteq I_+(\eta - x)x + I_+(x - \eta). \tag{2.13}$$

η is a new hyperparameter that controls the level of sparsity (or "fuzziness") of the produced representations. This new indicator function outputs x if $x < \eta$ and 1 otherwise. When $\eta = 0$, the two functions are equivalent and FTA produces one-hot vectors. As the value of η increases, the surrounding bins also activate. Higher values of η lead to more

bins activating, with bins closer to the center taking on higher values.

2.6 World Models

World models (or simply "models", when unambiguous) are models of an environment's dynamics that are often learned and used to facilitate efficiently learning a policy [21, 45, 48]. Depending on the use case, world models may attempt to learn the environment's transition dynamics p, the reward function r, the starting state distribution $\mathcal{S}_0$, or a combination of any of these. In model-based RL (MBRL), world models are used to update the agent's policy (or perhaps other auxiliary components) is a step called *planning*. The simplest example of a planning step with a policy gradient method is perhaps the application of a gradient update based on transitions sampled from the world model instead of from the real environment. Being able to sample from a learned model for updates generally reduces the amount of interaction required in the real environment, resulting in more sample efficient algorithms [5, 25, 26, 50].

Training world models of deterministic environments is straightforward, as each state-action pair leads to a single outcome. However, training world models of stochastic environments is more difficult because multiple outcomes are possible, meaning there is no single "correct" answer. Models of stochastic environments are broadly categorized into three classes depending on how they handle stochasticity: expectation models, sample models, and distribution models.

Expectation models are the simplest and easiest to learn, predicting the probability-weighted average over all possible outcomes. Under constraints, some types of planning are possible with expectation models [55], but they lose information about the probability of different transitions and can produce states that do not truly exist. Sample models produce outcomes with probability equal to that of outcomes in the environment, essentially making them proxies for the real environment. Distribution models capture the environment dynamics by learning a probability distribution over all possible outcomes. Both sample and distribution models are sufficient for most planning based methods, as both can capture the full dynamics of an environment.

2.6.1 Learning a Stochastic World Model

We use a variant of the method proposed by Antonoglou et al. [4] to learn sample models for stochastic environments. The method works similarly to a distribution model, first learning a distribution over possible outcomes during training, and then sampling from that distribution during evaluation. The problem faced by most distribution models is how to represent a distribution over a complex state space (or latent space in our case). Antonoglou et al. circumvent this problem by learning an encoder e that discretizes each state-action pair, mapping it to a single, k-dimensional one-hot vector we call the outcome vector. Each of the possible k values represents a different outcome of the transition.

The high-level idea is that while directly learning a distribution over full latent states is intractable, learning a categorical distribution over a limited, discrete set of outcomes (the outcome distribution) is possible. Whenever we wish to use the world model, we can sample from the outcome distribution and include the one-hot outcome vector as an additional input to the world model, indicating which of the k outcomes it should produce.

The training process for the sample model includes two objectives, which we explain for the example case of a world model that predicts state-action-state transitions. The first objective is to learn a function ψ that predicts the outcome distribution of state-action pairs, which is given by the minimization of the categorical cross-entropy between the predicted and ground-truth distributions:

$$\mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[-\sum_{i=1}^{k} e(s')_i \log \psi(s,a)_i\right]. \tag{2.14}$$

$\psi(s,a)$ is the predicted outcome distribution and $e(s')$ is the ground-truth next outcome vector. The second objective is then predicting the correct, next state with the world model, which is given by the minimization of

$$\mathbb{E}_{(s,a,s')\sim\mathcal{D}}\Big[l(s', w(s,a,e(s')))\Big], \tag{2.15}$$

where w is the world model, and l is a distance function comparing the predicted and ground-truth states. Both the world model and encoder are trained according to this latter objective.

We use this method in our work to learn sample models that predict state-action-state

transitions $w : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ (the encoder e is encapsulated by w in this notation), learning a model of the transition function p. We isolate model learning from the greater MBRL framework, and focus on learning accurate models. We do not make any claims about the efficacy of our methods for MBRL, as accuracy is only a proxy for downstream RL performance. As a trade off, however, we are able to more precisely examine how changes in state representation affect the process of learning a world model.

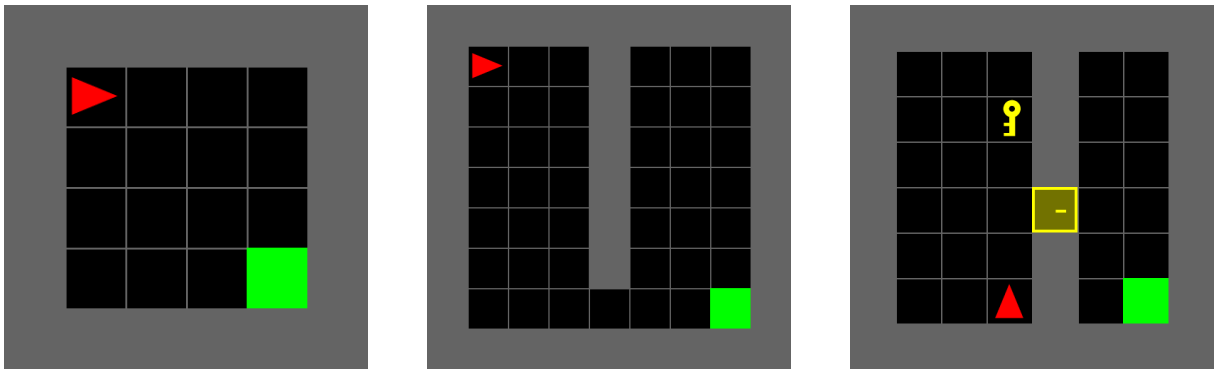
Chapter 3

World Model Learning with Discrete Representations

In this section, we examine the benefits of using discrete representations in the setting of learning a world model from a static dataset. We first learn a representation function that converts pixel-based observations to continuous or discrete representations, which we also refer to as the latent space. A world model is learned on top of the latent space, predicting transitions from one latent state to another. Starting on this supervised learning task that is void of non-stationarities allows us to better isolate the differences between discrete and continuous representations before introducing the difficulties presented by the full reinforcement learning problem.

3.1 Experimental Design

We begin by describing the setup for our experiments, including the environment specifications, world model training process, and evaluation methodology. Both the discrete and continuous world models are trained in a similar fashion, but the discrete world models require some extra steps to keep the representations discrete. We perform a hyperparameter sweep for both models, and evaluate the best models by analyzing the accuracy of their produced trajectories.



(a) Empty (b) Crossing (c) Door Key

Figure 3.1: Minigrid environments used in our experiments. The agent receives lower-resolution RGB arrays representing pixels as observations. We refer to these as the (a) *empty*, (b) *crossing*, and (c) *door key* environments.

3.1.1 Environments

Throughout this work, we use the *empty*, *crossing*, and *door key* Minigrid [13] environments, as displayed in Figure 3.1. In each environment, the agent controls a red arrow that points in the direction the agent is facing. The agent can freely move throughout the black, tiled area using the `left`, `right`, and `forward` actions. The agent in the *door key* environment also has access to `pickup` and `use` actions so that it may pickup the key and use it to unlock the door to traverse to the right side of the room.

The *empty* environment is deterministic, whereas the *crossing* and *door key* environments are stochastic. Taking an action in a stochastic environment has a 90% chance to enact the intended action, and a 10% chance to take a random, different action. The added stochasticity increases the difficulty of learning a world model by increasing the effective number of transitions that must be learned for each state-action pair. The increase in difficulty widens the performance between different methods, which makes the results easier to interpret.

The environment is episodic, with episodes terminating when the agent reaches the green square, or when the episode reaches a maximum length. The former yield a reward $[0.1, 1]$ depending on the length of the episode (shorter episodes yield higher rewards), and the latter yields no reward. The formula for the reward upon reaching the green square is given by $1 - 0.9\frac{t}{T}$, where t is the current step and T is the maximum episode length (dependent on the experiment). Observations are provided as arrays of pixels (RGB values) that are sized according to the dimensions of the environment. The environment

Environment Name	Image Dimensions	Actions	Stochastic	# of Unique States
Empty	$48 \times 48 \times 3$	left, right, forward	no	64
Crossing	$54 \times 54 \times 3$	left, right, forward	yes	172
Door Key	$64 \times 64 \times 3$	left, right, forward, pickup, use	yes	292

Table 3.1: Specifications for all Minigrid environments.

is technically partially observable because the agent does not have access to the time step, but in theory, this detail should not stop it from learning an optimal policy. The use of pixels obscures the simple, tabular nature of Minigrid environments, guaranteeing room for improving the representation. Further environment details are displayed in Table 3.1.

3.1.2 Training

We train autoencoders and world models on a static dataset of episodes collected with random walks. In each episode, the environment terminates when the agent reaches the green square or after 10,000 steps, until a total of one million transition tuples (s, a, r, s') are collected into dataset $\mathcal{D}$. In our experiments, observations are 3-dimensional RGB arrays, so we use convolutional and deconvolutional neural networks [34] for both the encoder and decoder architectures. The details of the architecture are given in Section 3.1.3.

The continuous model uses a vanilla autoencoder trained according to Equation 2.6, and the discrete model a VQ-VAE trained according to Equation 2.9. The VQ-VAE embedding gradient updates are weighted by a value of 0.25, as done in the original work by van den Oord et al. [54]. Both autoencoders are trained with states sampled from dataset $\mathcal{D}$ using the Adam optimizer [28] with hyperparameter values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a step size of 2×10^{-4}. Training continues until near-convergence, at which point the model weights are frozen and the world model training phase begins.

World models learned over continuous representations, which we refer to as *continuous world models*, take a latent state $\mathbf{z}$, and an action a as input to predict the next latent state $\hat{\mathbf{z}}' = w_\psi(\mathbf{z}, a)$ with an MLP $w_{\boldsymbol{\psi}}$. The world model consists of three layers of 64 hidden units (32 in the *crossing* environment), and rectified linear units (ReLUs) [2] for activations. In deterministic environments, the loss is given by the squared error between

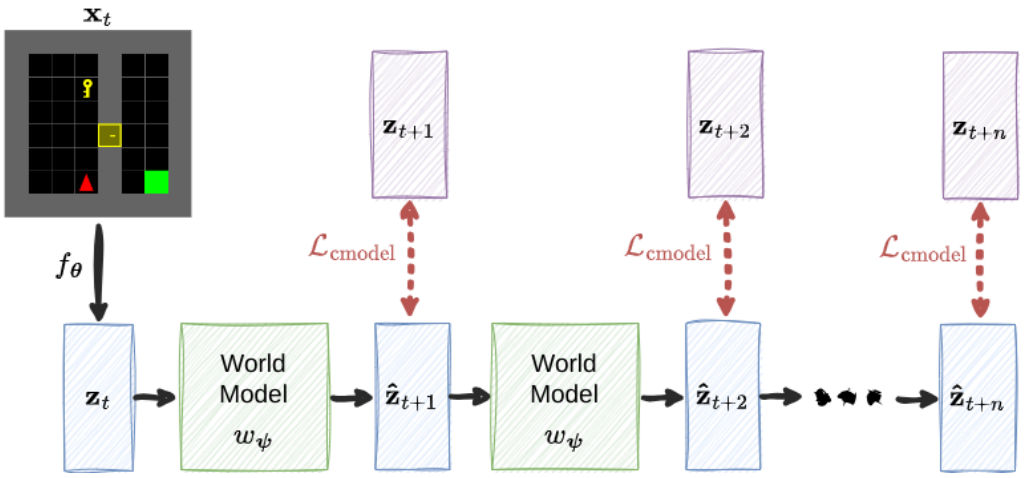


Figure 3.2: Depiction of a continuous world model training with n steps of hallucinated replay. After encoding the initial observation $\mathbf{x}_t$ into latent state $\mathbf{z}_t$, the world model rolls out a trajectory of predicted latent states, $\hat{\mathbf{z}}_{t+1}, \hat{\mathbf{z}}_{t+2}, \ldots, \hat{\mathbf{z}}_{t+n}$. Actions from a real trajectory are used in the training process, but are excluded in the depiction to avoid clutter. The loss at each time step is calculated by comparing the hallucinated latent state $\hat{\mathbf{z}}_{t+i}$ to the ground-truth, $\mathbf{z}_{t+i}$. This method is called hallucinated replay because the entire trajectory after the first latent state is hallucinated by the world model.

the predicted next latent state and the ground-truth next latent state:

$$\mathcal{L}_{\text{cmodel}} = \mathbb{E}_{s,a,s' \sim \mathcal{D}} \Big[||\hat{\mathbf{z}}' - \mathbf{z}'||_2^2 \,\Big|\, \mathbf{z}' = f_{\boldsymbol{\theta}}(s'), \hat{\mathbf{z}}' = w_{\boldsymbol{\psi}}(f_{\boldsymbol{\theta}}(s), a) \Big]. \tag{3.1}$$

Discrete world models use the same architecture, but concatenate the multiple one-hot vectors produced by VQ-VAEs $\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_K$ into a single latent state vector $\mathbf{z}$ before applying the world model. The output of the model $\hat{\mathbf{z}}'$ is split back into multiple latent vectors $\hat{\mathbf{z}}'_1, \hat{\mathbf{z}}'_2, ..., \hat{\mathbf{z}}'_K$, each corresponding to the logits of a one-hot vector. In deterministic environments, we use a categorical cross-entropy loss instead of the squared error because the discrete world model output represents a categorical distribution.[1] The loss is given by the mean of the cross-entropy between each of the predicted next latent vectors and the ground-truth next latent vectors:

$$\mathcal{L}_{\text{dmodel}} = \mathbb{E}_{s,a,s' \sim \mathcal{D}} \left[-\frac{1}{K} \sum_{i=1}^{K} \sum_{j=1}^{L} z'_{ij} \log \hat{z}'_{ij} \,\middle|\, \mathbf{z}' = f_{\boldsymbol{\theta}}(s'), \hat{\mathbf{z}}' = w_{\boldsymbol{\psi}}(f_{\boldsymbol{\theta}}(s), a) \right]. \tag{3.2}$$

In the above equation, $\mathbf{z}_i$ refers to the i-th latent vector of the latent state, and $\mathbf{z}_{ij}$ to the

[1] We also experimented with a squared error loss for the discrete world model and found it made little difference in the final world model accuracy.

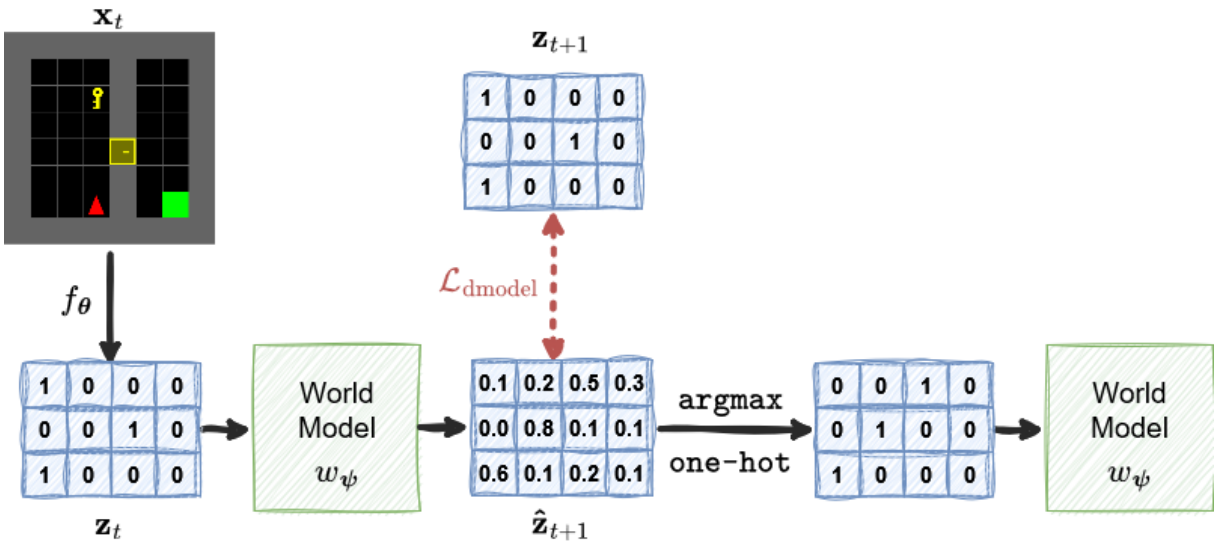


Figure 3.3: Depiction of a single step of discrete world model training and the subsequent discretization of the latent state. The observation $\mathbf{x}_t$ is encoded to produce latent state $\mathbf{z}_t$, which is passed to the world model to sample the logits $\hat{\mathbf{z}}_{t+1}$ for a following state. The predicted next state logits $\hat{\mathbf{z}}_{t+1}$ are compared to the ground truth state $\mathbf{z}_{t+1}$, which is constructed from the corresponding ground-truth observation: $\mathbf{z}_{t+1} = f_{\boldsymbol{\theta}}(\mathbf{x}_{t+1})$. Before the world model can be reapplied, the latent state logits must be discretized with an `argmax` operator and converted to the one-hot format.

j-th element of that vector.

All world models are trained with 4 steps of hallucinated replay as described by Talvitie [52] and depicted in Figure 3.2. Training continuous world models with hallucinated replay is as simple as feeding outputs of the model back in as new inputs. Training with the discrete world model, however, requires an extra step. Because the discrete model takes one-hot vectors as inputs but outputs vectors of logits, the logits must be converted back into one-hot vectors before being used. Each vector of logits is converted based on the index of the maximum value:

$$o_i = \begin{cases} 1 & \text{if } i = \text{argmax(logits)} \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

This conversion is applied prior to all recurrent applications of a discrete world model in both the training and later evaluation protocol. Figure 3.3 depicts a single step of training a discrete world model and the subsequent latent one-hot conversion.

As discussed in Section 2.6.1, training world models in stochastic environments is more difficult because there is no single "correct" answer for next state predictions. Recognizing this challenge, we opt to learn sample models in stochastic environments using the method proposed by Antonoglou et al. [4]. We learn a distribution over possible outcomes according to Equation 2.14, and both the continuous and discrete world model objective functions are augmented according to Equation 2.15. Table A.1 provides the relevant hyperparameters.

3.1.3 Autoencoder Architecture

The vanilla autoencoder and the VQ-VAE use the same encoder and decoder architecture, only differing in the layer that produces the latent state. The decoder is a mirror of the encoder, reversing each of the shape transformation, so we describe only the encoder architecture. The encoder starts with three convolutional layers with square filters of sizes $\{8, 6, 4\}$, channel of sizes $\{64, 128, 64\}$, strides of $\{2, 2, 2\}$ (or $\{2, 1, 2\}$ for the crossing environment), and uniform padding of $\{1, 0, 0\}$. Each convolutional layer is followed by a ReLU activation. The downscaling convolutions are followed by an adaptive pooling layer that transforms features into a shape of $(K \times K \times 64)$, and finally a residual block [22] consisting of a convolutional layer, batch norm [24], ReLU, convolutional layer, and another batch norm. These general layers are followed by layers specific to the type of autoencoder.

The vanilla autoencoder flattens the convolutional output and projects it to a latent space of size D with a linear layer. We use a value of $K = 8$ and sweep over values of $D = \{16, 64, 256, 1024\}$ for each environment. We use $D = 64$ for the *empty* environment, $D = 256$ for *crossing*, and $D = 1024$ for *door key*, though we note that we do not observe a statistically significant difference in performance for values of $D \geq 64$.

The VQ-VAE directly quantizes the output of the general layers, so the only other parameters added are the embedding vectors. The number of vectors that make up a latent state is given by K^2, and we let L be the number of embedding vectors, resulting in discrete representations of shape (K^2, L). We sweep over values of $K = \{3, 6, 9\}$ and $L = \{16, 64, 256, 1024\}$ for each environment. We use $K = 6$ and $L = 1024$ (for a total size of 6,144) for all environments except for *crossing*, which uses a value of $K = 9$ (for a total size of 9,216).

When designing the experiments, we considered how to construct a fair comparison

between the continuous and discrete methods despite the fact that each have different ideal sizes of the latent state, which makes one model bigger than the other. This is a particularly difficult question because it is unclear if we should focus on the size of a representation in bits, or the size of the representation in the number of values used to represent it in a deep learning system. A discrete representation is orders of magnitude smaller than a continuous representation if represented in bits ($9 \times \log_2 1024 = 90$ bits in the *crossing* environment), but takes an order of magnitude more values to represent as one-hot vectors being passed to a neural network ($9 \times 1024 = 9216$ values in the *crossing* environment). Ultimately, we found that answering this question was unnecessary, as the performance of both methods was limited no matter how large we made the size of the representations. In the *crossing* environment, for example, the performance of the continuous model would not increase even if we increased the size of the latent state from 256 to 9,216 values to match that of the discrete latent state.

3.1.4 Evaluation

The goal of the evaluation is to measure how the representation of the latent space affects the ability to learn an accurate world model. Unfortunately, this is not as simple as comparing a predicted latent state to the ground-truth latent state. In stochastic environments, there may be many correct transitions, and an accurate sample model will produce each of the possible outcomes with the correct frequency. To account for this, we look at the state distributions induced over many episodes to measure the accuracy of world models.

To evaluate representations in a specific environment, we start by choosing a target policy. One of the major benefits of using a world model instead of replaying past data is that a world model has the potential to simulate trajectories outside of those normally experienced by an agent. To reflect this use case, we choose some target policies that differ from the data collection policy. For the *empty* environment, we use a random target policy, for *crossing*, a target policy that moves to explore the right side of the environment, and for *door key*, a target policy that navigates directly to the goal. In each environment, we simulate the target policy for 10,000 episodes, cutting off each episode early, or freezing the environment at the terminal state to reach exactly 30 interaction steps.

We compare the ground-truth distributions at each step with those induced by re-

cursively applying a learned world model. An accurate model will induce a distribution similar to the ground-truth over the state space at each step, while an inaccurate model will induce a different state distribution. We measure these differences qualitatively by visualizing the distributions, and quantitatively by measuring a numerical difference between the ground-truth and predicted distributions. Figure 3.5 contains a visualization that helps build an intuition of how state distributions may differ, and is discussed further in the experiments section.

Because we learn world models over the latent space, we cannot directly visualize their outputs. At each step, we convert the latent state into a real environment state by passing it through the decoder and matching the output to the closest real environment state: $s = \arg\min_{s' \in \mathcal{S}} \|g_{\boldsymbol{\phi}}(\mathbf{z}) - s'\|$, where $g_{\boldsymbol{\phi}}(\mathbf{z})$ is the decoder applied to latent state $\mathbf{z}$. Averaging over the states from 10,000 separate runs at each step yields an image depicting how the agent is spread throughout the environment.

We measure the quantitative difference between two state distributions with the KL divergence. Our Minigrid environments all have less than 300 unique states, so we use the formula for the KL divergence between two categorical distributions:

$$\mathrm{KL}(p \parallel q) \doteq \sum_{x \in \mathcal{X}} p(x) \log \left(\frac{p(x)}{q(x)} \right), \tag{3.4}$$

where p is the ground-truth state distribution, and q is the state distribution induced by the world model. Because the KL divergence is a measure of the difference between two distributions, methods with a lower KL divergence are better.

We also include two additional, simple baselines in our comparisons: the uniform baseline and the delayed state baseline. The uniform baseline predicts a uniform distribution over all states. It is strong when the agent's target policy leads it to spread out, like in a random walk. The delayed state baseline predicts the ground-truth state distribution delayed by one step, which is information the other methods do not have access to. It is a strong baseline because of that, and especially so when agent does not move much, like when the agent has already reached the goal (because the agent is frozen until the 30-step termination mark).

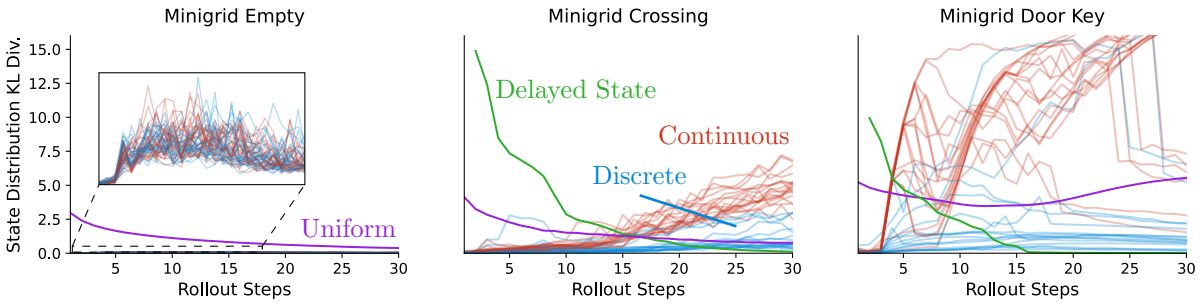


Figure 3.4: The KL divergence between the ground-truth state distribution and the world model induced state distribution. Lower values are better, indicating a closer imitation of the real environment dynamics. Each line depicts the evaluation of an individual run, which includes training an autoencoder and a world model. Between 16 and 30 runs are plotted per method.

3.2 Results

With our methodology defined, we run a number of experiments aimed at characterizing the effects of using discrete representations in this model learning setting. We begin with a straightforward comparison between the accuracy of the continuous and discrete methods, and then follow up with experiments to better understand the difference in performance. We change the size of the transition model to test how both methods behave under model capacity constraints, and run another experiment to confirm the discrete representation of the latent state is actually responsible for the observed performance increase. We conclude the section with a discussion of the findings before advancing to the episodic RL setting.

3.2.1 Model Rollouts

We roll out the trained world models for 30 steps and evaluate their accuracy, plotting the results in Figure 3.4. In the *empty* environment, there is virtually no difference between the performance of the continuous and discrete world models, as both predict transitions with near-perfect accuracy. As the complexity progressively increases in the *crossing* and then in the *door key* environment, the gap in accuracy widens, with the discrete world model performing better after just a few steps.

We examine visualizations of trajectories to better understand the patterns observed in Figure 3.4, showing two visualizations that most clearly represent these patterns in

Figures 3.5 and 3.6. The trajectories predicted by the continuous model in the *crossing* environment rarely make it across the gap in the wall, which manifests as a steady increase in the KL divergence starting around step 14. The performance of the continuous model in the *door key* environment suffers much earlier as the agent fails to swiftly pickup the key, and again as the agent struggles to pass through the door. Notably, these two actions occur infrequently in the training data because the training data is generated with random walks, and because they can only happen once per episode even when they do occur. Stated concisely, discrete representations are allowing the world model to more accurately predict transitions that occur less frequently in the training data.

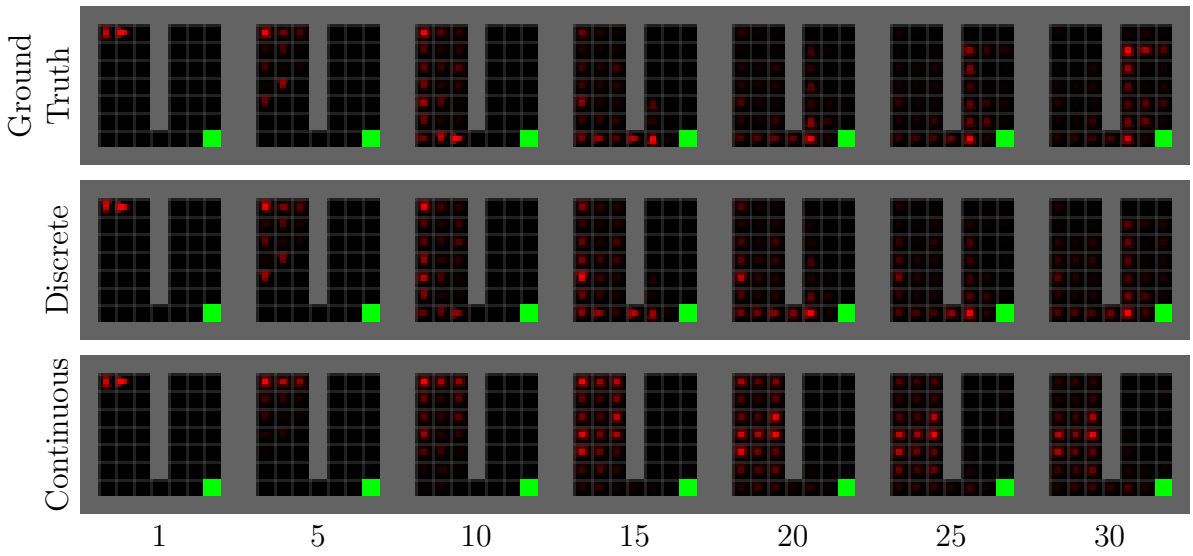


Figure 3.5: Comparison of rollouts predicted by different world models in the *crossing* environment. Each row visualizes the state distributions throughout rollouts predicted by different world models, with the x-axis giving the step in the rollout. The ground-truth row depicts the state distribution over rollouts as a policy that explores the right side of the environment is enacted in the true environment. Predicted observations are averaged over 10,000 rollouts. Being closer to the ground-truth indicates a higher accuracy.

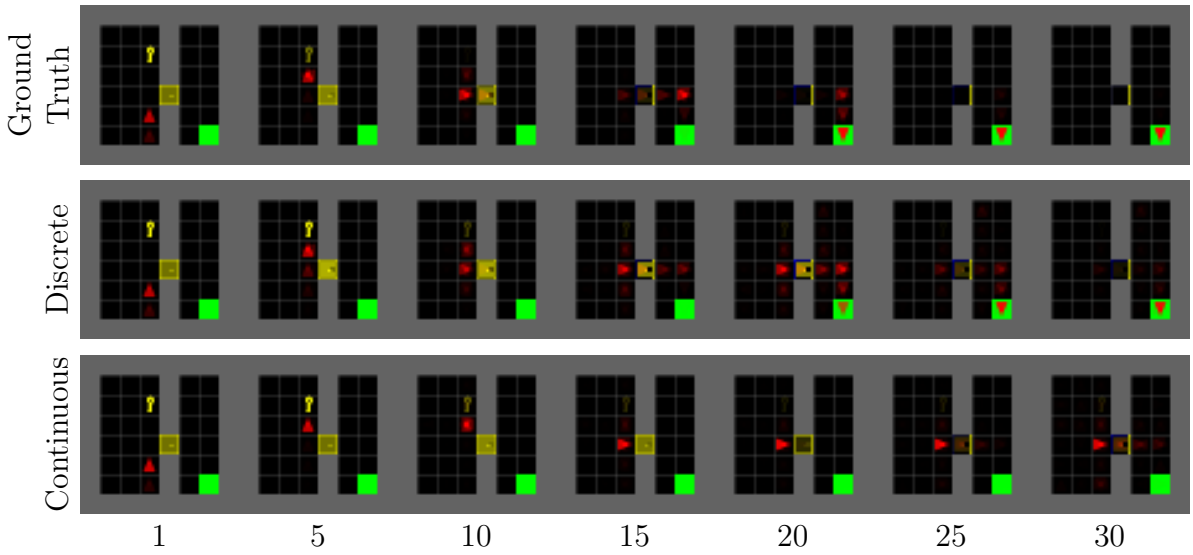


Figure 3.6: Comparison of rollouts predicted by different world models in the *door key* environment. Each row visualizes the state distributions throughout rollouts predicted by different world models, with the x-axis giving the step in the rollout. The ground-truth row depicts the state distribution over rollouts as a policy that navigates to the goal state is enacted in the true environment. Predicted observations are averaged over 10,000 rollouts. Being closer to the ground-truth indicates a higher accuracy.

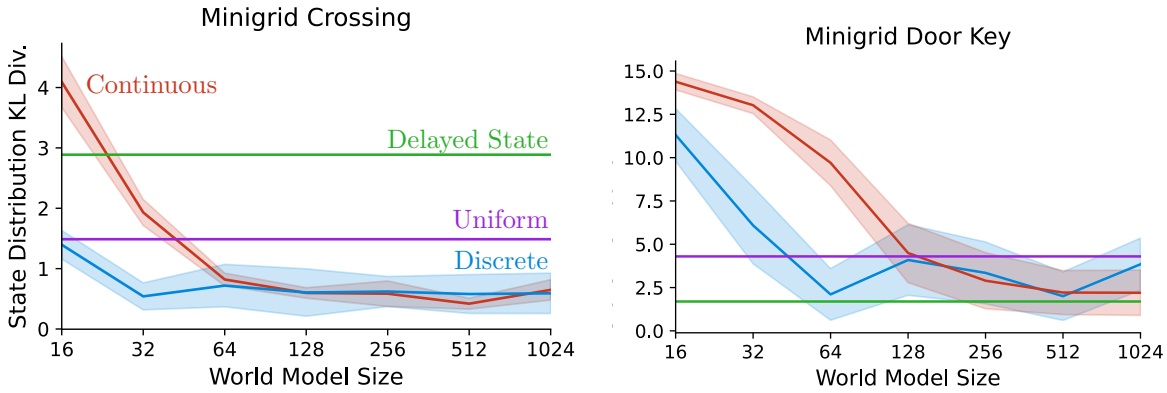


Figure 3.7: The KL divergence between the ground-truth state distribution and the world model induced state distribution, averaged over 30 steps. Lower values are better, indicating a closer imitation of the real environment dynamics. The x-axis gives the number of hidden units per layer for all three layers of the world model. Each point shows the mean KL divergence and 95% confidence interval of 14 to 20 runs.

3.2.2 Scaling the World Model

Despite sweeping over the size of the latent vector for the hyperparameter sweep, we were unable to find an encoder architecture that enabled the continuous world model to adequately learn transitions underrepresented in the training data. **Either the discrete representations allow learning something that is not learnable with the continuous representations, or the fixed size of the world model is limiting the continuous model's performance.** We vary the size of the world model, rather than the encoder, to test the latter hypothesis. For each size of the world model, we tune the size of the latent vectors for both the vanilla autoencoder and the VQ-VAE using the same sweep process described in Section 3.1.3. We plot the performance of each transition model in Figure 3.7, averaging the KL divergence over 30-step trajectories. Note that the naive baselines in the new setup seem much stronger for two reasons: the continuous and discrete methods suffer from significant error accumulation towards the later half of the rollouts, and they also contain outliers that significantly affect their average performances. In contrast, the delayed state baseline is near perfect at the end of trajectories due to privileged information that only it receives, and neither baseline has outliers given how they are defined.

In the plot, an interesting pattern emerges: the performance of the continuous and discrete methods are indistinguishable beyond a certain size for the world model. Only when the environment dynamics cannot be modeled near-perfectly, due to the limited

capacity of the world model, are the discrete representations beneficial. As the size of the world model shrinks, the performance of the continuous model degrades at a faster rate, showing that the discrete world model is able to model more of the world with less capacity. This gap is notable because we are interested in the setting where the world is much larger than the agent. In this setting, discrete representations are better because they allow an agent to learn more despite its limited capacity.

3.2.3 Representation Matters

Our experiments have demonstrated a clear advantage of using discrete representations when the size of the transition model is limited, but is the improvement actually attributable to the one-hot representation of the data? It is possible that the discrete world model performs better because the VQ-VAE learns more information, or information that better facilitates world modeling, and not because of the one-hot structure of the latent states. In our next experiment, that is the question we ask: **do the benefits of the discrete world models originate from the representation of the latent states, or from the informational content of the latent states?** We test this by constructing informationally equivalent continuous and discrete representations and testing them in the world model learning setting.

The discrete representations we use are unchanged – we use the same VQ-VAE one-hot encoding formula. We also use VQ-VAEs to produce continuous representations in this experiment, but with latent states that are comprised of multiple quantized vectors instead of one-hot encoded vectors. Referring back to Section 2.4, recall that there are two different ways of representing the same latent state. Given a single latent vector, its corresponding one-hot vector gives the index of the closest embedding, and the quantized vector is that embedding itself; both vectors contain the same semantic content, but have different representations. If we set the dimensionality of the embedding vectors equal to the number of embedding vectors, then both one-hot and quantized representations also take the same shape.

We exploit this representational dichotomy to probe at the importance of the representation in the world modeling setting. If the representation of the latent space does matter, then we would expect these two semantically equivalent representations to perform differently. If the representation does not matter, then we would expect them to perform similarly. To prepare this experiment, we set the number of embedding vectors

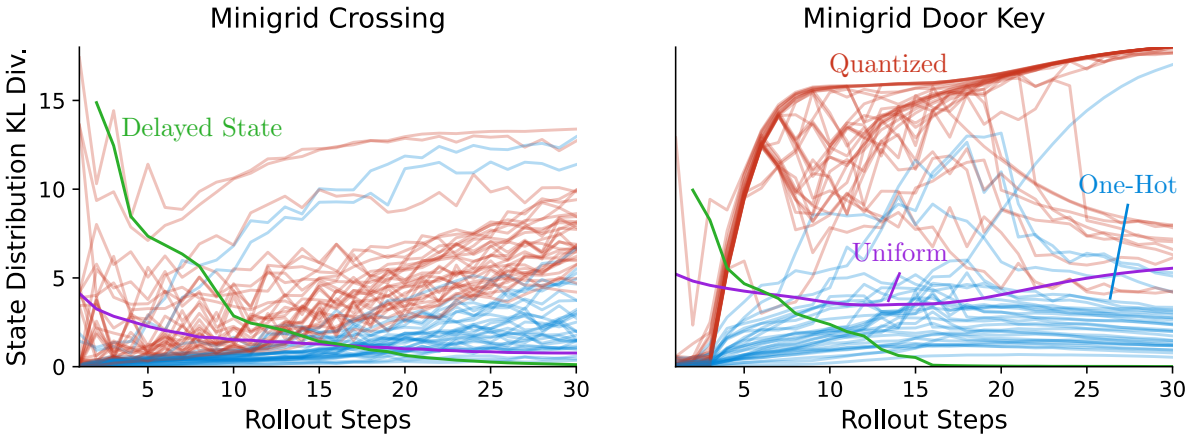


Figure 3.8: The KL divergence between the ground truth state distribution and the world model induced state distribution. Lower values are better, indicating a closer imitation of the real environment dynamics. Both methods use the same VQ-VAE architecture, but represent the information in different ways. The one-hot method is the same as the previously used discrete method, and the quantized representations are continuous vectors with the same semantic meaning. Each line depicts the evaluation of an individual run, with 30 runs per method.

and the dimensionality of the embeddings to 64 so that both types of representations take the same shape for a fair comparison. The world model learned with quantized representations functions similarly to the discrete world model described in Section 3.1.2, except the cross-entropy loss is replaced with an MSE loss. All outputs of the world model are re-quantized using the VQ-VAE's embedding space. This functionality is equivalent to how the outputs of discrete world models are "snapped" to the nearest one-hot vectors with an `argmax` function.

In Figure 3.8 we compare the accuracy of world models learned on top of one-hot encoded vectors to those learned on top of quantized vectors. The quantized representations perform poorly in both environments, failing nearly entirely at picking up the key in the *door key* environment. Despite the semantic equivalence of the representations and nearly equivalent training procedure, the world model with one-hot encodings is more accurate, demonstrating the importance of the representation of the latent space.

3.3 Discussion

In every experiment in this section, the discrete world model performs at least as well as its continuous counterpart. Under some conditions, the discrete world model performs better by more accurately modeling transitions that occur less frequently in the training data. It performs better in the model rollout experiments as the environment becomes more complex, and it performs better in the scaling experiment when the size of the world model is limited. Synthesizing these observations, we see that the discrete world model is more accurate when the resources of the agent are insufficient to near perfectly model the environment. Hence, discrete representations enable us to accurately model more of the world with less resources.

Our motivation for investigating representation learning is to find a representation that allows an agent to succeed in a world much larger than itself. In this context, the ability to model more of the world is useful. Modeling parts of the world that were previously out of reach means that the agent can perform that much better. However, increasing the agent's capacity alone is not enough if we assume the world is *significantly* larger than the agent. The agent's model of the world will still be imperfect, so quick adaptation remains a necessity. We address this concern in the next section as we delve into the full reinforcement learning setting.

Chapter 4

Policy Learning with Discrete Representations

In the previous chapter, we demonstrated advantages of using discrete representations when dealing with a stationary, i.i.d. dataset. As we progress to the full reinforcement learning problem, we face new challenges, like that of learning from non-stationary distributions. Our first experiments of this section aim to understand the effects of using discrete representations in the standard, episodic RL setting. Identifying a clear benefit, we progress to the continual RL setting with continually changing environments [1] as a proxy for environments that are too big for the agent to perfectly model.

4.1 Experimental Design

We train all RL agents in this section with the clipping version of proximal policy optimization (PPO) [46], and bootstrap returns using the value function if the final state in a batch is not a terminal state. Instead of observations, the policy and value functions intake learned representations. Separate networks are used for the policy and value functions, but both share the same architecture – an MLP with two hidden layers of 256 units and ReLU activations. We perform a grid search over the most sensitive hyperparameters for the continuous model, sweeping over clipping values $\epsilon \in \{0.1, 0.2, 0.3\}$, and the number of training epochs per batch $n \in \{10, 20, 30, 40\}$. We use the same final hyperparameters for training the discrete model, which are provided in table A.2. We use dimensions for the latent spaces of both encoders that consistently achieve a low reconstruction loss. The

vanilla autoencoder uses a 256-dimensional latent space, and the VQ-VAE uses 36 latent vectors and 256 embedding vectors (which forms a latent space of shape 36×256). All other autoencoder hyperparameters and architectural details remain unchanged.

The training loop consists of three steps: collecting data, training the actor-critic model, and training the autoencoder. This setup differs from previous experiments primarily in the manner that data collection and training of all of the models happen in tandem instead of in separate phases. In each iteration of the loop, the policy is rolled out to collect 256 transition tuples, which are stored in a first-in, first-out buffer. The policy and value functions are trained on the most recent 256 transition tuples for 10 epochs using PPO. The PPO gradient updates only affect the policy and value function weights, with backpropagation stopping at the encoder outputs. The autoencoder is trained for eight epochs using the same loss functions as described in Section 3.1.2. For each epoch, a new, random set of 256 observations is sampled from the transition buffer.

Agents are trained in the *crossing* and *door key* environments shown in Figure 3.1, but with shortened episode lengths. The maximum episode length is set to 400 in the *crossing* environment and 1,000 in the *door key* environment.

4.2 Experiments

In this section, we compare RL agents trained over discrete representations to those trained over continuous representations. We begin with the episodic RL setting, and learn that discrete representations can help learn a good policy faster, but only if the representations are given enough time to learn prior to RL updates. We transition the continual RL setting, where the environment periodically changes. We find that the discrete representations help agents quickly adapt to changes in the environment, making them ideal for continual learning.

4.2.1 Episodic RL

We train RL agents with continuous and discrete representations in the *crossing* and *door key* environments, plotting the results in Figure 4.1. The continuous model is faster to achieve better performance in the *crossing* environment, but there is no statistically significant difference in the *door key* environment. Though discrete representations initially seem to convey no benefit in this vanilla RL setting, inspecting the autoencoder

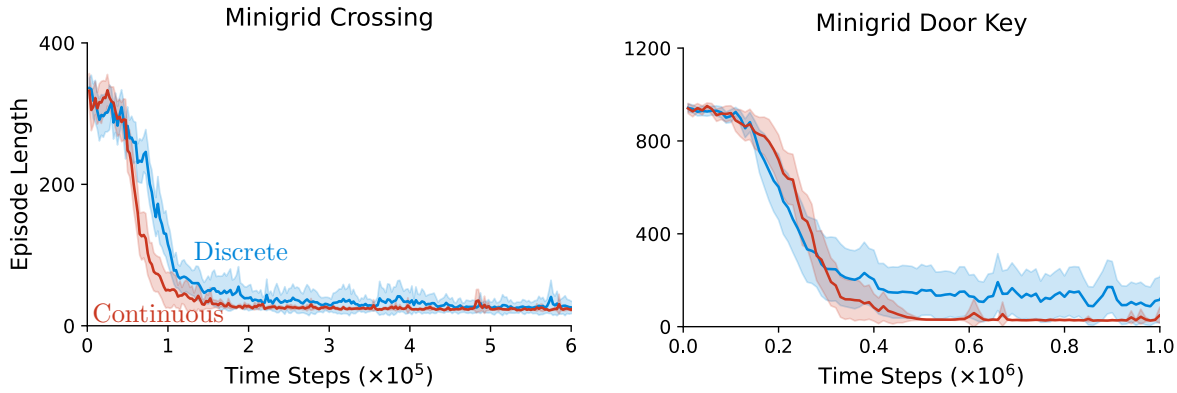


Figure 4.1: Episode length binned into 100 buckets and averaged over 30 runs with a 95% confidence interval. Both methods are trained with PPO; only the type of autoencoder differs [46]. Shorter episode lengths are better, as they indicate the agent is finding the goal faster.

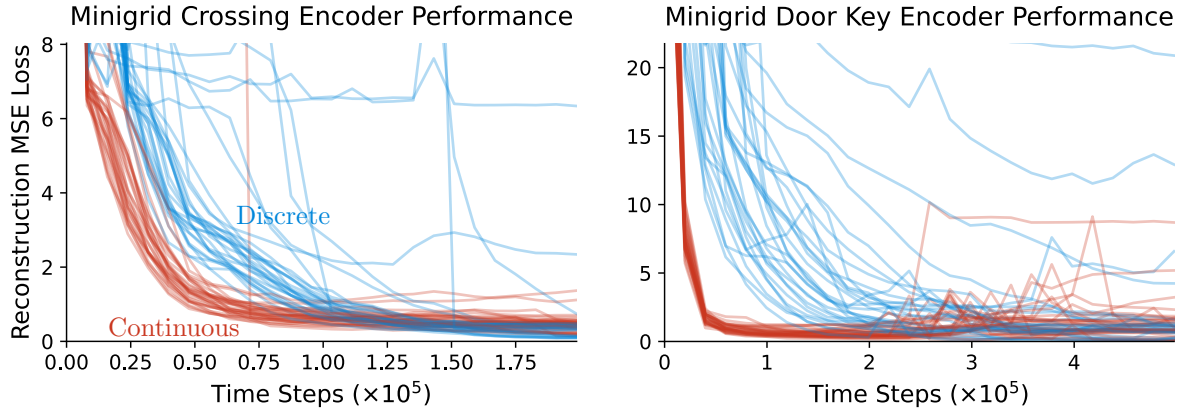


Figure 4.2: Reconstruction loss of the autoencoder binned into 100 buckets for each of the 30 runs per method. The autoencoder is trained on observations randomly sampled from a buffer that grows as the RL training progresses. Lower is better, indicating a better reconstruction of the input observation

learning curves in Figure 4.2 reveals an important detail: the vanilla autoencoder learns faster than the VQ-VAE. If the speed of the RL learning updates is our primary concern (whether it actually is will be discussed later), then the learning speed of the autoencoder is a confounding factor. We can remove this confounding factor by delaying PPO updates until both autoencoders are trained to around the same loss.

We rerun the same RL experiment, but delay the first PPO update by a fixed number of steps, until the autoencoders in most runs have converged. We plot the results in Figure 4.3. Instead of the previously ambiguous results, we now see the discrete model

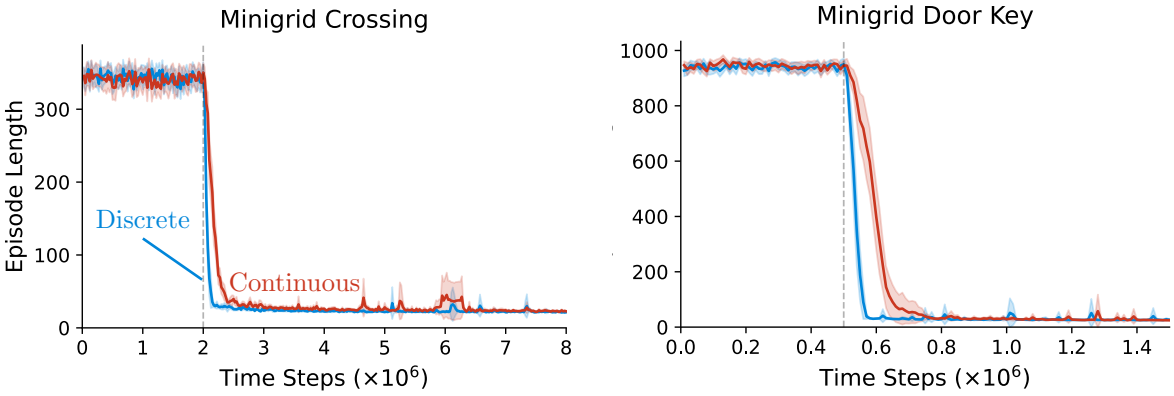


Figure 4.3: Episode length binned into 100 buckets and averaged over 30 runs with a 95% confidence interval. Both methods are trained with PPO; only the type of autoencoder differs [46]. Only the autoencoder is trained up to the dotted, black line, at which point PPO updates also begin. Shorter episode lengths are better, as they indicate the agent is finding the goal faster.

decisively outperforming the continuous model. Though the gap between both methods looks small, the discrete model achieves the same performance as the continuous model in a fraction of the time.

4.2.2 Continual RL

In the previous section, we showed that an agent was able to more quickly learn to navigate to the goal when using discrete representations, but this benefit only arose when we allowed the VQ-VAE extra time to learn, prior to RL training. In a Minigrid environment, this cost is bearable; the environment is static and simple, making representation learning a relatively quick and easy process. In the big world setting, where the agent cannot perfectly model the world, and the world is always changing from the agent's perspective, is this approach still beneficial?

To answer this question, we modify the *crossing* and *door key* environments from the previous experimental setup. Every fixed amount of steps, and on the very first step of each run, the starting state of the environment is randomized. All of the same items and walls remain, but their positions are randomized, only the positions of the goal and outer walls staying constant. The *crossing* environment changes every 40,000 steps, and *door key* environment every 100,000 steps. Example environment initializations are shown in Figure 4.4. By only changing the environment after a long delay, we create specific points

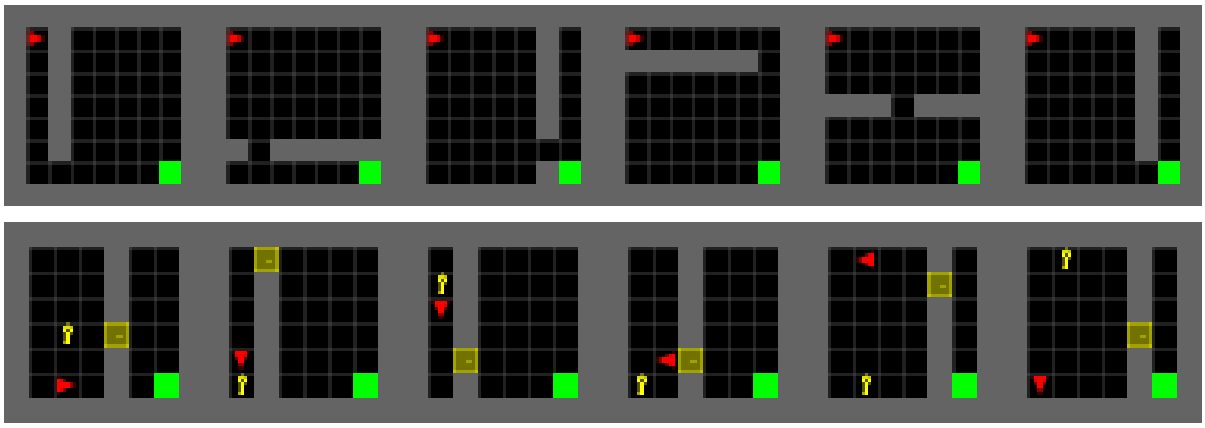


Figure 4.4: The top row depicts random initializations of the *crossing* environment, and the bottom that of the *door key* environment. Each time the environment changes, the positions of all internal walls and objects are randomized, with the exception of the agent position in the *crossing* environment and the goal in both environments.

in the learning process where we can clearly observe the difference between how discrete and continuous methods adapt to change.

We rerun the previous RL experiment (shown in Figure 4.3), using the same training procedure with the same delay before the first PPO update, but periodically change the environment. We plot the results in Figure 4.5, where we observe a spike in the episode length each time the environment changes, indicating that the agents' previous policies are no longer sufficient to solve the new environments. Both agents are able to adapt and improve their policies before the environment changes again, but on average, the discrete agent adapts faster and achieves better performance 10 out of 10 times in both the *changing crossing* and *changing door key* environments.

While the slower, initial learning speed of the VQ-VAE hinders its ability to maximize reward at the beginning of the training process, it does not seem to hinder its ability to adapt after an initial representation has already been learned. Inspecting the reconstruction loss of both autoencoders, plotted in Figure 4.6, we see that the VQ-VAE's reconstruction loss increases much less when the environment changes. The shorter spikes suggest that the VQ-VAE representations generalize better, allowing them to adapt faster when the environment changes.

With these results, we return to the initial question: can discrete representations be beneficial in RL even if the initial representation is learned slower? We argue in the affirmative. If we consider continually learning RL agents in the big world setting, where the goal of the agent is to maximize reward over its lifetime by quickly adapting

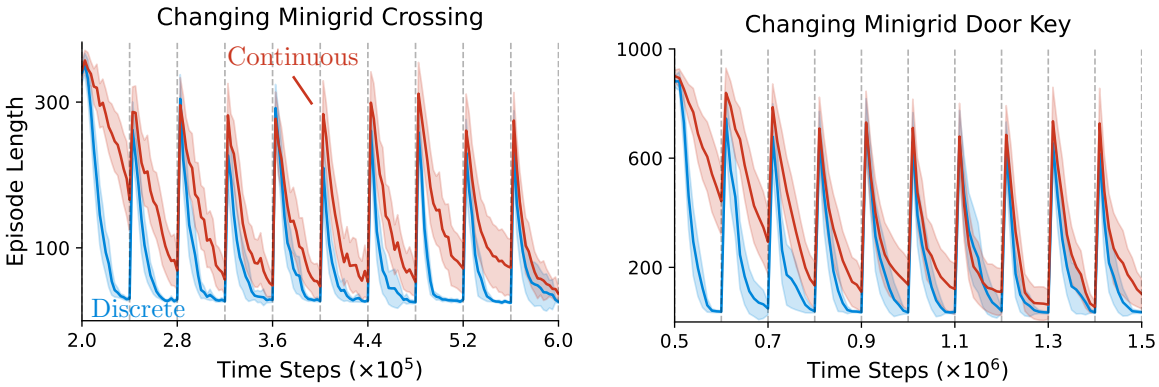


Figure 4.5: Episode length averaged over 30 runs with a 95% confidence interval. The average episode length is plotted every 2,500 steps in the *crossing* environment and every 10,000 steps in the *door key* environment. Black, dotted lines indicate a change to the environment. Both methods are trained with PPO after an initial delay, which allows the autoencoder to get a head start on learning representations. Only the type of autoencoder differs between methods. The plot only depicts the performance starting from the first PPO update. Refer to Figure A.1 for the full figure. A faster drop in episode length is better, indicating faster adaptation to the changed environment.

to unpredictable scenarios, then the cost of learning an initial representation is easily amortized by a lifetime of faster adaptation.

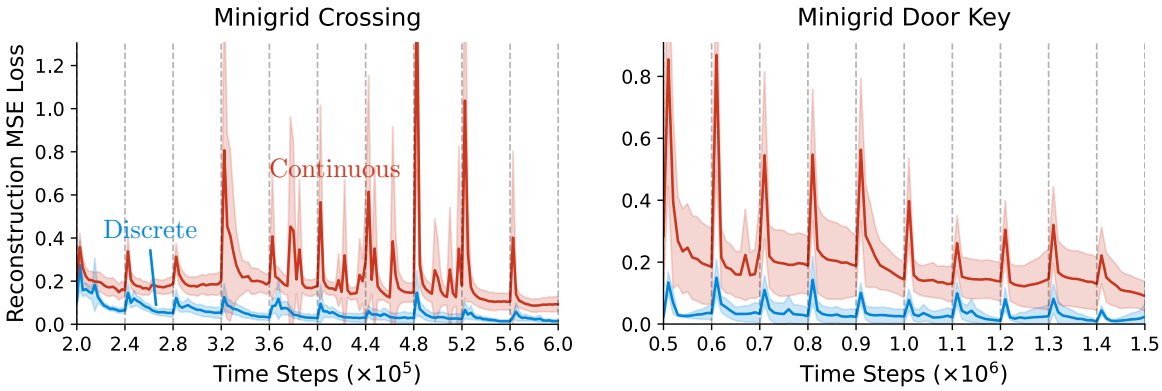


Figure 4.6: Reconstruction loss of both encoders averaged over 30 runs with a 95% confidence interval. The average reconstruction loss is plotted every 2,500 steps in the *crossing* environment and every 10,000 steps in the *door key* environment. An autoencoder and policy are trained in tandem for each run. Lower peaks mean the representation generalizes better, and a quicker decrease means the autoencoder is learning faster. Overall, a lower average reconstruction loss is better.

4.3 Baseline Comparison

The continual RL agents using representations produced by VQ-VAEs have worked better than their continuous counterparts in our experiments, but how do our results compare to those of other methods? In our final experiment, we introduce two new methods to better understand how our results fit into the bigger picture. We introduce a baseline that uses only the PPO objective with no autoencoder reconstruction loss, and another baseline that generates representations with FTA.

We refer to the former as the *RL only* baseline, as it has no auxiliary reconstruction objective. It shares the same architecture as the continuous model, with the exception that is does not have a decoder (because the decoder is only needed to calculate the reconstruction loss). The *RL only* baseline trains end-to-end, backpropagating the PPO gradients through the policy network, value network, and encoder. This is the simplest, and perhaps the most common way of applying RL.

The FTA baseline is nearly identical to the vanilla AE (previously denoted as the *continuous*) method, with the exception that FTA, as described in Section 2.5, is applied to the representation layer. This creates a sparse representation, but the architecture and training procedure are otherwise the same. FTA is known to be a strong baseline [38, 42,

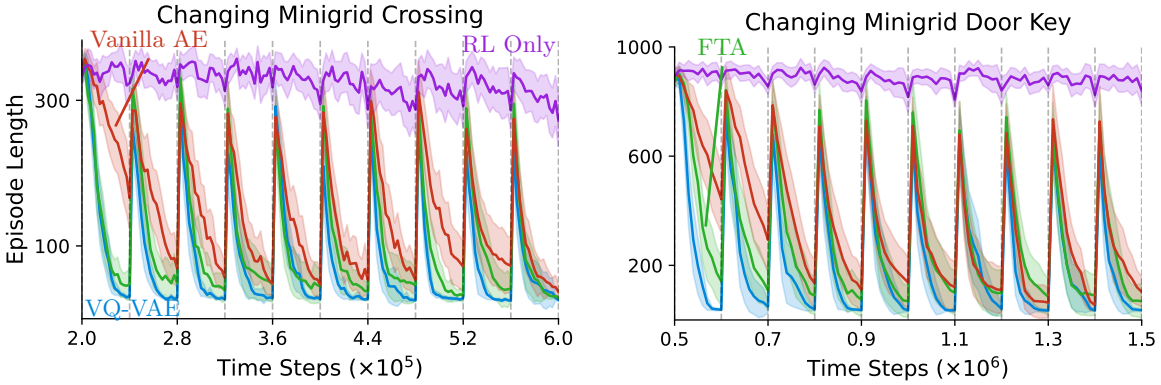


Figure 4.7: Episode length averaged over 30 runs with a 95% confidence interval. The average episode length is plotted every 2,500 steps in the *crossing* environment and every 10,000 steps in the *door key* environment. Black, dotted lines indicate a change to the environment. All methods are trained with PPO after an initial delay, which allows the autoencoder to get a head start on learning representations on methods other than *RL Only*. Only the type of autoencoder differs, except for *RL Only*, which has no reconstruction loss. The plot only depicts the performance starting from the first PPO update. Refer to Figure A.2 for the full figure. A faster drop in episode length is better, indicating faster adaptation to the changed environment.

57], and it is similar to the discrete baseline in the way that it generates "fuzzy" one-hot vectors. For this method, we set the input bounds to $[-2, 2]$, and in each environment, sweep over the latent dimension (prior to FTA) $D \in \{16, 64, 256, 1024\}$, the number of bins $k \in \{8, 16, 32, 64\}$, and the sparsity hyperparameter $\eta \in \{0.125, 0.25, 0.5\}$. The final hyperparameters in both environments are $D = 256$, $k = 8$, and $\eta = 0.5$, which sets the size of each bin to 0.25.

We plot the results in Figure 4.7, which includes both the old and new methods. The *RL only* method falls far behind all of the other methods, with a slight improvement noticeable only towards the end of the *crossing* environment plot. These results fall in line with what is expected given previous work [20, 27, 43, 58], that show learning from autoencoder representations is more efficient than learning from raw pixel observations.

The FTA method produces better results that fall in between the vanilla AE and VQ-VAE curves. If we measure performance as the cumulative reward (or average episode length) per randomized environment configuration, we see that the VQ-VAE still performs the best, having the best performance in nine out of ten configurations in the *crossing* environment, and ten out of ten of the configurations in the *door key* environment. FTA performs better than the vanilla AE in ten out of ten of the *crossing* configurations, and

seven out of ten of the *door key* configurations. Though the performance of FTA does not surpass that of the VQ-VAE, it partially bridges the gap between the vanilla AE and the VQ-VAE. Given that FTA representations are sparse but not discrete, these results raise the question of whether "discretness" is truly necessary. Perhaps the sparsity of the VQ-VAE representations is what makes them so effective.

4.4 Discussion

In this section, we moved to the full RL problem, where the agent must learn in a world that is no longer static. In this setting, agents learning from discrete representations were quicker to learn how to navigate to the goal state. When tasked with learning in a periodically changing environment, faster learning along with the better generalization exhibited by discrete representations translated to faster adapting agents. In the face of a large world that the agent cannot hope to fully model, sample efficient adaptation is key. These experiments demonstrate that discrete representations learned by a VQ-VAE can facilitate quick adaptation, making them a promising candidate in the search for alternative ways of representing the current state of an environment.

Chapter 5

Conclusion & Future Work

At the start of this work, we claimed that when limited in training data, model capacity, or computation, models learned over discrete representations often find better solutions, and do so faster, than their continuous counterparts. With experimental results that support this claim, we now turn to integrate our findings and discuss an overall interpretation of the results, the implications of our work, and future directions.

5.1 A Common Thread

In the model learning setting, discrete representations enabled learning more of the world with a limited capacity. In the continual RL setting, discrete representations enabled faster learning and adaptation. However, what remains ambiguous is why the application of discrete representations results in distinct benefits across different contexts. Is there a deeper understanding that would draw a connection between these distinct results? Consolidating our findings in the past sections, we hypothesize that all of the improvements observed from discrete representations can be explained via the same underlying mechanism.

The goal in most machine learning tasks, including the settings in this work, can be simplified into learning a model that maps a given input distribution to some other output distribution. That is, the model is trying to best approximate the true, underlying function that maps inputs to outputs, which we refer to as the target function. One way to judge the difficulty of a learning task is by the size or complexity of the target function; more complex target functions correspond to more difficult tasks and less complex

functions to easier tasks. Though our work lacks a precise tool for measuring this notion of "complexity", the idea is analogous to the VC-dimension in PAC learning. A more complex concept class (space of target functions) will require a hypothesis space (model) with a higher VC-dimension (capacity) to be sufficiently learned. We hypothesize that target functions mapping discrete representations to things we tend to care about (like the next state or the optimal action) are often less complex than those mapping from raw, continuous inputs, and hence, are easier to learn.

If learning from discrete representations makes the target function less complex when learning a model of a vast world, we would expect to more accurately model more of the world. This is what we observe in Figures 3.4 and 3.7. As the difficulty of the environment increases or the size of the world model shrinks, making perfect modeling impossible, the discrete world model gets better and better. If the target function is less complex when learning a policy, then we would expect to learn the policy faster because learning smaller, less complex functions is an easier task. This is what we observe in Figures 4.1 and 4.5. Learning a policy from discrete representations is faster, and when the environment changes, the agent is again faster to learn a new, strong policy.

5.2 Future Work

Our work offers several contributions, but there is still much to be understood about using discrete representations in RL. Much of potential future work can be categorized as further understanding why discrete representations are effective. We propose the hypothesis that target functions mapping from discrete representations are less complex, but is that truly the case? And if so, why are they less complex? Although we show that the discrete representation of information can be important in Figure 3.8, it is not clear that the discreteness of the representations is what matters. It is possible, for example, that the sparsity of the the VQ-VAE representations is what makes them effective [56]. There are likely also other ways to learn useful, discrete representations that could be worth exploring.

Our work also does little to suggest how far these results extend. The use of discrete representations in models like DreamerV3 [21] and the success of VQ-VAEs in even the domain of computer vision [15, 23, 41, 54], where inputs are complex, suggest that that these results will scale up. Without a thorough study, however, we cannot be certain. Future work could study the effectiveness of discrete representations in environments that

are fully continuous, unlike Minigrid, which has a finite number of states and a state space that is inherently represented (in the code) by a set of discrete values. If the performance of discrete representation-based method do suffer in fully continuous environments, then the combination of both continuous and discrete values within a single representation could be another interesting direction to explore.

5.3 Conclusion

In this work, we explored the effects of learning from discrete representations in three modules that are commonly found in models of intelligent agents: a world model, a value function, and a policy. In both settings, the use of discrete representations benefited the learning process. Discrete world models were able to better model more of the world with less resources, and discrete agents learned to navigate to the goal and adapt to changes in the environment faster. Synthesizing these observations, we offered a hypothesis that explains these results as separate manifestations of a single phenomenon: functions we tend to care about, like world models and optimal policies, are less complex, and hence, easier to learn when learned from discrete representations.

Our contributions extend beyond understanding, and implicate discrete representations learned by VQ-VAEs as a promising candidate for the representation of observations in continual RL agents. If we care about agents working in worlds much larger than themselves, we must accept that they will be incapable of perfectly modeling the world. The agent will see the world as forever changing due to its limited capacity, which is the case in complex environments like the real world [30, 51]. If we wish to address this issue in the representation learning space, agents must learn representations that enable quick adaptation, and are themselves quick to adapt [49]. Discrete representations learned by VQ-VAEs do exactly that, and provide a path towards ever more efficient, continually learning RL agents.

References

[1] Zaheer Abbas, Rosie Zhao, Joseph Modayil, Adam White, and Marlos C. Machado. Loss of plasticity in continual deep reinforcement learning. In *Conference on Lifelong Learing Agents (CoLLAs)*, 2023.

[2] Abien Fred Agarap. Deep learning using rectified linear units (ReLU). *CoRR*, abs/1803.08375, 2018.

[3] P. C. Edgar An, W. Thomas Miller III, and P. C. Parks. Design improvements in associative memories for cerebellar model articulation controllers (CMAC). *Artificial Neural Networks*, 47:1207–1210, 1991.

[4] Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K. Hubert, and David Silver. Planning in stochastic environments with a learned model. In *International Conference on Learning Representations (ICLR)*, 2022.

[5] Christopher G. Atkeson and Juan Carlos Santamaría. A comparison of direct and model-based reinforcement learning. In *International Conference on Robotics and Automation (ICRA)*, 1997.

[6] Dana H. Ballard. Modular learning in neural networks. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 1987.

[7] André Barreto, Will Dabney, Rémi Munos, Jonathan J. Hunt, Tom Schaul, David Silver, and Hado van Hasselt. Successor features for transfer in reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[8] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton,

Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016.

[9] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research (JAIR)*, 47:253–279, 2013.

[10] Marc G. Bellemare, Will Dabney, Robert Dadashi, Adrien Ali Taïga, Pablo Samuel Castro, Nicolas Le Roux, Dale Schuurmans, Tor Lattimore, and Clare Lyle. A geometric perspective on optimal representations for reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[11] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *Institute of Electrical and Electronics Engineers (IEEE)*, 35(8):1798–1828, 2013.

[12] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432, 2013.

[13] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

[14] Will Dabney, André Barreto, Mark Rowland, Robert Dadashi, John Quan, Marc G. Bellemare, and David Silver. The value-improvement path: Towards better representations for reinforcement learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2021.

[15] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[16] Jesse Farebrother, Joshua Greaves, Rishabh Agarwal, Charline Le Lan, Ross Goroshin, Pablo Samuel Castro, and Marc G. Bellemare. Proto-value networks: Scaling representation learning with auxiliary tasks. In *International Conference on Learning Representations (ICLR)*, 2023.

[17] Sina Ghiassian, Banafsheh Rafiee, Yat Long Lo, and Adam White. Improving performance in reinforcement learning by breaking generalization in neural networks. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2020.

[18] William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[19] Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations (ICLR)*, 2022.

[20] Danijar Hafner, Timothy P. Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering Atari with discrete world models. In *International Conference on Learning Representations (ICLR)*, 2021.

[21] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy P. Lillicrap. Mastering diverse domains through world models. *CoRR*, abs/2301.04104, 2023.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Institute of Electrical and Electronics Engineers (IEEE)*, 2016.

[23] Yan Hong, Li Niu, Jianfu Zhang, and Liqing Zhang. Few-shot image generation using discrete content representation. In *International Conference on Multimedia*, 2022.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015.

[25] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[26] Chi Jin, Zeyuan Allen-Zhu, Sébastien Bubeck, and Michael I. Jordan. Is q-learning provably efficient? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[27] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *International Conference on Robotics and Automation (ICRA)*, 2019.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[29] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations (ICLR)*, 2014.

[30] Saurabh Kumar, Henrik Marklund, Ashish Rao, Yifan Zhu, Hong Jun Jeon, Yueyang Liu, and Benjamin Van Roy. Continual learning as computationally constrained reinforcement learning. *CoRR*, abs/2307.04345, 2023.

[31] Charline Le Lan, Stephen Tu, Adam Oberman, Rishabh Agarwal, and Marc G. Bellemare. On the generalization of representations in reinforcement learning. In Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera, editors, *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2022.

[32] S.H. Lane, D.A. Handelman, and J.J. Gelfand. Theory and development of higher-order cmac neural networks. *IEEE Control Systems*, 12(2):23–30, 1992.

[33] Yann LeCun. A path towards autonomous machine intelligence. *Open Review*, 2022.

[34] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551, 1989.

[35] Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael H. Bowling. State of the art control of Atari games using shallow reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2016.

[36] Clare Lyle, Mark Rowland, Georg Ostrovski, and Will Dabney. On the effect of auxiliary tasks on representation dynamics. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2021.

[37] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew J. Hausknecht, and Michael Bowling. Revisiting the Arcade Learning Environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research (JAIR)*, 61:523–562, 2018.

[38] Erfan Miahi. Feature generalization in deep reinforcement learning: An investigation into representation properties, 2022.

[39] Vincent Micheli, Eloi Alonso, and François Fleuret. Transformers are sample-efficient world models. In *International Conference on Learning Representations (ICLR)*, 2023.

[40] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015.

[41] Charlie Nash, Jacob Menick, Sander Dieleman, and Peter W. Battaglia. Generating images with sparse representations. In *International Conference on Machine Learning (ICML)*, 2021.

[42] Yangchen Pan, Kirby Banman, and Martha White. Fuzzy tiling activations: A simple approach to learning sparse representations online. In *International Conference on Learning Representations (ICLR)*, 2021.

[43] Bharat Prakash, Mark Horton, Nicholas R. Waytowich, William David Hairston, Tim Oates, and Tinoosh Mohsenin. On the use of deep autoencoders for efficient embedded reinforcement learning. In *Great Lakes Symposium on VLSI (GLSVLSI)*, 2019.

[44] Jan Robine, Tobias Uelwer, and Stefan Harmeling. Smaller world models for reinforcement learning, 2021.

[45] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering Atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[47] Richard S. Sutton. The quest for a common model of the intelligent decision maker. *CoRR*, abs/2202.13252, 2022.

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

[49] Richard S. Sutton, Anna Koop, and David Silver. On the role of tracking in stationary environments. In *International Conference on Machine Learning (ICML)*, 2007.

[50] Richard S. Sutton, Csaba Szepesvári, Alborz Geramifard, and Michael H. Bowling. Dyna-style planning with linear function approximation and prioritized sweeping. In *Uncertainty in Artificial Intelligence*, volume 24, pages 528–536, 2008.

[51] Richard S. Sutton, Michael H. Bowling, and Patrick M. Pilarski. The Alberta plan for AI research. *CoRR*, abs/2208.11173, 2022.

[52] Erik Talvitie. Self-correcting models for model-based reinforcement learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2017.

[53] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller. Deepmind control suite. *CoRR*, abs/1801.00690, 2018.

[54] Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[55] Yi Wan, Muhammad Zaheer, Adam White, Martha White, and Richard S. Sutton. Planning with expectation models. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 28, pages 3649–3655, 2019.

[56] Han Wang, Erfan Miahi, Martha White, Marlos C. Machado, Zaheer Abbas, Raksha Kumaraswamy, Vincent Liu, and Adam White. Investigating the properties of neural network representations in reinforcement learning. *CoRR*, abs/2203.15955, 2022.

[57] Han Wang, Erfan Miahi, Martha White, Marlos C. Machado, Zaheer Abbas, Raksha Kumaraswamy, Vincent Liu, and Adam White. Investigating the properties of neural network representations in reinforcement learning. *CoRR*, abs/2203.15955, 2023.

[58] Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering Atari games with limited data. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Appendix A

Extra Results & Hyperparameters

A.1 Experiment Details

Hyperparameter	**Value**
Bin count	32
Discretization projection	256, 256
Prediction projection	256, 256

Table A.1: Hyperparameters for all sample-based world models that account for stochasticity using the same method as Antonoglou et al. [4]. *Bin count* is the number of discrete classes into which outcomes can be discretrized. The projection hyperparameters give the sizes of the hidden layers used to discretize states and predict the next discrete state. ReLUs are used between all hidden layers.

Hyperparameter	**Value**
Horizon (T)	256
Adam step size	256
(PPO) Num. epochs	10
(PPO) Minibatch size	64
Discount (γ)	0.99
(Autoencoder) Num. epochs	8

Table A.2: Hyperparameters for all RL training procedures used in Chapter 4. Hyperparameters for both PPO and other training details are included. Hyperparameters for PPO are referred to by the same naming convention as in Schulman et al. [46].

A.2 Supplemental Continual RL Figures

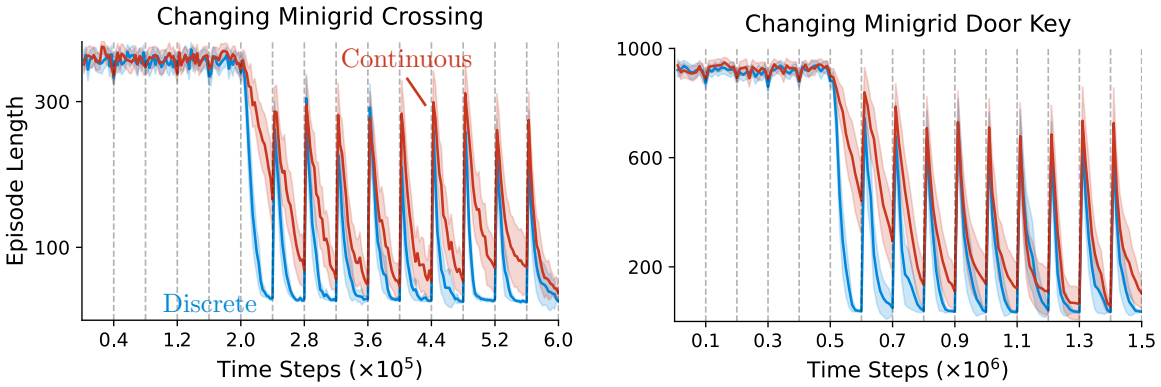


Figure A.1: Episode length averaged over 30 runs with a 95% confidence interval. The average episode length is plotted every 2,500 steps in the *crossing* environment and every 10,000 steps in the *door key* environment. Black, dotted lines indicate a change to the environment. Both methods are trained with PPO after an initial delay, which allows the autoencoder to get a head start on learning representations. The PPO updates start after 200K steps in the *crossing* environment, and after 500K steps in the *door key* environment. A faster drop in episode length is better, indicating faster adaptation to the changed environment.

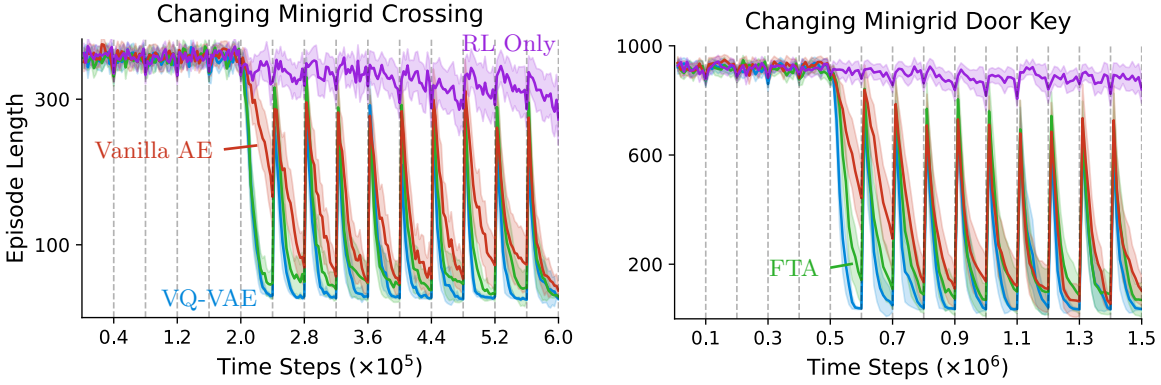


Figure A.2: Episode length averaged over 30 runs with a 95% confidence interval. The average episode length is plotted every 2,500 steps in the *crossing* environment and every 10,000 steps in the *door key* environment. Black, dotted lines indicate a change to the environment. All methods are trained with PPO after an initial delay, which allows the autoencoder to get a head start on learning representations on methods other than *RL Only*. The PPO updates start after 200K steps in the *crossing* environment, and after 500K steps in the *door key* environment. Only the type of autoencoder differs, except for *RL Only*, which has no reconstruction loss. A faster drop in episode length is better, indicating faster adaptation to the changed environment.