Generalization to New Actions in Reinforcement Learning - Appendix

A. Environment Details

A.1. Grid World

The Grid World environment, based on Chevalier-Boisvert et al. (2018), consists of an agent and a randomly placed lava wall with an opening, as shown in Figure 1. The lava wall can either be horizontal or vertical. The agent spawns in the top left corner, and its objective is to reach the goal in the bottom-right corner of the grid while avoiding any path through lava. The agent can move using 5-step skills composed of steps in one of the four directions (Up, Down, Left, and Right). An episode is terminated when the agent uses a maximum of 10 actions (50 moves), or the agent reaches the goal (success) or lava wall (failure).

State: The state space is a flattened version of the 9x9 grid. Each element of the 81-dimensional state contains an integer ID based on whether the cell is empty, wall, agent, goal, lava, or subgoal.

Actions: An action or skill of the agent is a sequence of 5 consecutive moves in 4 directions. Hence, $4^5 = 1,024$ total actions are possible. Once the agent selects an action, it executes 5 sequential moves step-by-step. During a skill execution, if the agent hits the boundary wall, it will stay in the current cell, making a null interaction. If the agent steps on lava during any action, the game will be terminated.

Reward: Grid world provides a sparse subgoal reward on passing the subgoal for the first time and a sparse goal reward when the agent reaches the goal. The goal reward is discounted based on the number of actions taken to encourage a shorter path to the goal. More concretely,

$$R(s) = \lambda_{Subgoal} \cdot \mathbf{1}_{Subgoal} \; + \; (1 - \lambda_{Goal} \frac{N_{total}}{N_{max}}) \cdot \mathbf{1}_{Goal} \tag{1}$$

where $\lambda_{Subgoal} = 0.1,\ \lambda_{Goal} = 0.9,\ N_{max} = 50,$
$N_{total} =$ number of moves to reach the goal.

Action Set Split: The whole action set is randomly divided into a 2:1:1 split of train, validation, and test action sets.

Action Observations: The observations about each action demonstrate an agent performing the 5-step skill in an 80x80 grid with no obstacles. Each observation is a trajectory of states resulting from the skill being applied, starting from a random initial state on the grid. A set of 1024 such trajectories characterizes a single skill. By observing the effects caused on the environment through a skill, the action representation module can extract the underlying skill behavior, which is further used in the actual navigation task. Different types of action representations are described and visualized in Section B.

A.2. Recommender System

We adapt the Recommender System environment from Rohde et al. (2018) that simulates users responding to product recommendations (the schematic shown in Figure 2). Every episode, the agent makes a series of recommendations for a new user to maximize their cumulative click-through rate. Within an episode, there are two types of states a user can transition between: organic session and bandit session. In the bandit session, the agent recommends one of the available products to the user, which the user may select. After this, the user can transition to an organic session, where the user independently browses products. The agent takes action (product recommendation) whenever the user transitions to the bandit session. Every user interaction with organic or bandit sessions varies their preferences slightly, resulting in a change to the user's vector. As a result, the agent cannot repetitively recommend the same products in an episode, since the user is unlikely to click it again. The environment provides engineered action representations, which are also used by the environment to determine the likelihood of a user clicking on the recommendation. The episode terminates after 100 recommendations or stochastically in between the session transitions.

State: The state is a 16-dimensional vector representing the user, $\mathbf{v}_{user}$. Every episode, a new user is created with a vector $\mathbf{v}_{user} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. After each step in the episode, the user transitions between organic and bandit sessions, where the user vector is perturbed by resampling $\mathbf{v}_{user} \sim \mathcal{N}(\mathbf{v}_{user},\, \sigma_1\sigma_2\boldsymbol{I})$, where $\sigma_1 = 0.1$ and $\sigma_2 \sim \mathcal{N}(0, 1)$.

Actions: There are a total of 10,000 actions (products) to recommend to users. Each action is associated with a 16 dimension representation, $c \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. The selected product's representation and the current user vector determine the probability of a click. The agent's objective is to recommend articles that maximize the user's click-through rate. The probability of clicking a recommended product i with action representation c_i is given by:

$$\begin{aligned} p_{click}(\mathbf{v}_{user}, \mathbf{c}_i) &= f(\mathbf{c}_i \cdot \mathbf{v}_{user} + \mu_i), \text{where} \\ f(x) &= \sigma(a * \sigma(b * \sigma(c * x) - d) - e), \end{aligned} \tag{2}$$

where $a = 14, b = 2, c = 0.3, d = 2, e = 6, \sigma$ is the

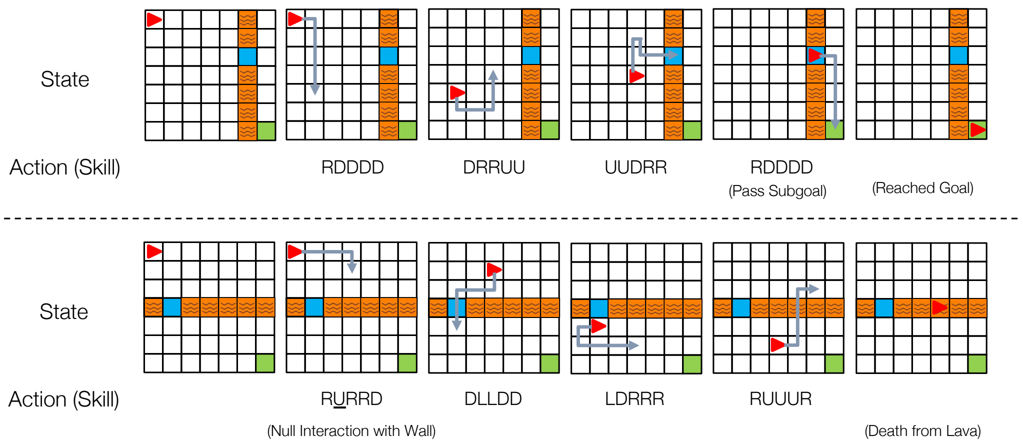


Figure 1. Grid World Environment: 9x9 grid navigation task. The agent is the red triangle, and the goal is the green cell. The environment contains one row or column of lava wall with a single opening acting as a subgoal (blue). Each action consists of a sequence of 5 consecutive moves in one of the four directions: U(p), D(own), R(ight), L(eft).

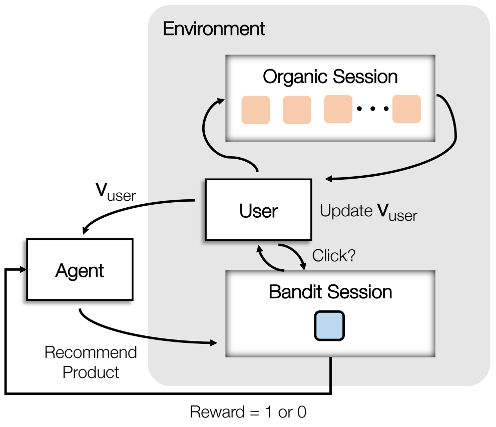


Figure 2. Recommender System schematic: The user transitions stochastically between two sessions: organic and bandit. Each transition updates the user vector. Organic sessions simulate the user independently browsing other products. Bandit sessions simulate the agent recommending products to the current user. A reward is given if the user clicks on the recommended product.

sigmoid function, $\cdot$ denotes a vector dot product. Here, μ_i is an action-specific constant kept hidden from the agent to simulate partial observability, as would be the case in real-world recommender systems. Constants used in the function f make the click-through rate, p_{click}, to be a reasonable number, adapted and modified from Rohde et al. (2018).

In Section C.5, we also provide results on the fully observable recommender system environment, where the agent has access to μ_i as well. Concretely, μ_i is concatenated to c_i to form the action representation which the learning agent utilizes to generalize.

Reward: There is a dense reward of 1 on every recommendation that receives a user click, which is determined by p_{click} computed in Eq 2.

Action Set Split: The 10,000 products are randomly divided into a 2:1:1 split of train, validation, and test action sets.

A.3. Chain REAction Tool Environment (CREATE)

Inspired by the popular video game, *The Incredible Machine*, Chain REAction Tool Environment (CREATE) is a physics-based puzzle where the objective is to get a target ball (red) to a goal position (green), as depicted in Figure 3. Some objects start suspended in the air, resulting in a falling movement when the game starts. The agent is required to select and place tools to redirect the target ball towards the goal, often using other objects in the puzzle (like the blue ball in Figure 3). The agent acts every 40 physics simulation steps to make the task reasonably challenging and uncluttered. An episode is terminated when the agent accomplishes the goal, or after 30 actions, or when there are no moving objects in the scene, ending the game. CREATE was created with the Pymunk 2D physics library (Blomqvist) and Pygame physics engine (Shinners).

CREATE environment features 12 tasks, as shown in Figure 8. Results for 3 tasks are shown in the main paper and 9 others in Figure 8. Concurrently developed related environments (Allen et al., 2019; Bakhtin et al., 2019) focus on single-step physical reasoning with a few simple polygon tools. In contrast, CREATE supports multi-step RL, features many diverse tools, and requires continuous tool placement.

State: At each time step, the agent receives an 84x84x3 pixel-based observation of the game screen. Here, each originally colored observation is turned into gray-scale and the

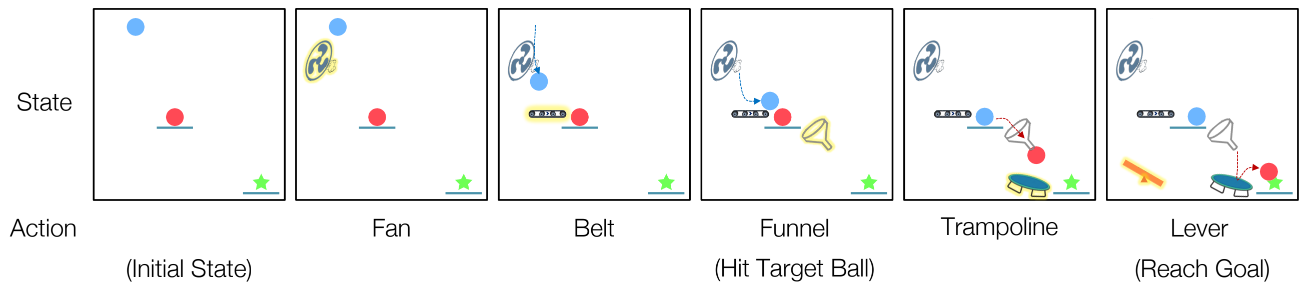


Figure 3. CREATE Push Environment: The blue ball falls into the scene and is directed towards the target ball (red), which is pushed towards the goal location (green star). This is achieved with the use of various physical tools that manipulate the path of moving objects in peculiar ways. At every step, the agent decides which tool to place and the (x, y) position of the tool on the screen.

past 3 frames are stacked channel-wise to preserve velocity and acceleration information in the state.

Actions: In total, CREATE consists of 2,111 distinct tools (actions) belonging to the classes of: ramp, trampoline, lever, see-saw, ball, conveyor belt, funnel, 3-, 4-, 5-, and 6-sided polygon, cannon, fan, and bucket. 2,111 tools are obtained by generating tools of each class with appropriate variations in parameters such as angle, size, friction, or elasticity. The parameters of variation are carefully chosen to ensure that any resulting tool is significantly different from other tools. For instance, no two tools are within 15° difference of each other. There is also a *No-Operation* action, resulting in no tool placement.

The agent outputs in a hybrid action space consisting of (1) the discrete tool selection from the available tools, and (2) (x, y) coordinates for placing the tool on the game screen.

Reward: CREATE is a sparse reward environment where rewards are given for reaching the goal, reaching any subgoal once, and making the target ball move in certain tasks. Furthermore, a small reward is given to continue the episode. There is a penalty for trying to overlap a new tool over existing objects in the scene and an invalid penalty for placing outside the scene. The agent receives the following reward:

$$R(s, a) = \lambda_{alive} + \lambda_{Goal} \cdot \mathbf{1}_{Goal} \cdot \lambda_{Subgoal} \cdot \mathbf{1}_{Subgoal} \cdot \lambda_{target\,hit} \cdot \mathbf{1}_{target\,hit} + \lambda_{invalid} \cdot \mathbf{1}_{invalid} + \lambda_{overlap} \cdot \mathbf{1}_{overlap} \tag{3}$$

where $\lambda_{alive} = 0.01$, $\lambda_{Goal} = 10.0$, $\lambda_{Subgoal} = 2.0$, $\lambda_{target\,hit} = 1$, and $\lambda_{invalid} = \lambda_{overlap} = -0.01$.

Action Set Split: The tools are divided into a 2:1:1 split of train, validation, and test action sets. In *Default Split* presented in the main experiments, the tools are split such that the primary parameter (angle for most) is randomly split between training and testing. This ensures that the test tools are considerably different from the training tools in the same class. The validation set is obtained by randomly splitting the testing set into half. In *Full Split*, 1,739 of the total tools are divided into a 2:1:1 split by tool class, as described in Table 1.

Train	Ramp, Trampoline, Ball, Bouncy Ball, See-saw, Cannon, Bucket
Validation and Test	Triangle, Bouncy Triangle, Lever, Fan, Conveyor Belt, Funnel

Table 1. Tool classes in the CREATE Full split.

Additionally, we used a total of 7566 tools generated at 3° angle differences for analysis experiments to study generalization properties. HVAE was trained as an oracle encoder over the entire action set, to get action representations suitable for all three analyses. The policy's performance was studied independently by training it on 762 distinct tools with at least 15° angle differences and evaluated based on analysis-specific action sampling from the rest of the tools (e.g. at least 5° apart).

Action Observations: Each tool's observations are obtained by testing its functionality through scripted interactions with a probe ball. The probe ball is launched at the tool from various angles, positions, and speeds. The tool interacts with the ball and changes its trajectory depending on its properties, e.g. a cannon will catch and re-launch the ball in a fixed direction. Thus, these deflections of the ball can be used to infer the characteristics of the tool. Examples of these action observations are shown at https://sites.google.com/view/action-generalization/create.

The collected action observations have 1024 ball trajectories of length 7 for each tool. The trajectory is composed of the environment states, which can take the form of either the 2D ball position (default) or 48x48 gray-scale images. The action representation module learns to reconstruct the corresponding data mode, either state trajectories or videos, for obtaining the corresponding action representations. Different types of action representations used are described and visualized in Section B.

A.4. Shape Stacking

In Shape Stacking, the agent must place shapes to build a tower as high as possible. The scene starts with two cylinders of random heights and colors, dropped at random locations on a line, which the agent can utilize to stack towers. For each action, the agent selects a shape to place and where to place it. The agent acts every 300 physics simulator steps to give time for placed objects to settle into a stable position. The episode terminates after 10 shape placements.

State: The observation at each time step is an 84x84 grayscale image of the shapes lying on the ground. We stack past 4 frames to preserve previous observations in the state.

Actions: The action consists of a discrete selection of the shape to place, the x position on the horizontal axis to drop the shape, and a binary episode termination action. The height of the drop is automatically calculated over the topmost shape, enabling a soft drop. If a shape has already been placed, trying to place it again does nothing. There are a total of 810 shapes of classes: triangle, tetrahedron, rectangle, cone, cylinder, dome, arch, cube, sphere, and capsule. These shapes are generated by varying the scale and vertical orientation in each shape class. The parameter variance is carefully chosen to ensure all the shapes are sufficiently different from each other.

In Figure 13, we compare various hybrid action spaces with shape selection. We study different ways of placing a shape: dropping at a fixed location, or deciding x-position, or deciding (x, y)-positions.

Reward: To encourage stable and tall towers, there is a sparse reward at episode end, for the final height of the topmost shape in the scene, added to the average heights of all N shapes in the scene:

$$R(s) = (\lambda_{top} \max(h_i) + \lambda_{avg} \frac{1}{N} \sum_i h_i) \cdot \mathbf{1}_{Done}, \quad (4)$$

where h_i is the height of shape i and $\lambda_{top} = \lambda_{avg} = 0.5$.

Action Set Split: The shapes are divided into a 2:1:1 split of train, validation, and test action sets. In Default Split presented in the main experiments, the shapes are split such that the primary parameter of scale is randomly split between training and testing. This ensures that test tools are considerably different in scale from the train tools in the same class. The validation set is obtained by randomly splitting the test set into half. In Full Split, the split is determined by shape class, as shown in Table 2.

Train	Domes, Rectangles, Capsules, Triangles, Arches, Spheres
Validation and Test	Cylinders, Tetrahedrons, Cubes, Cones, Angled-Rectangles, Angled-Triangles

Table 2. Shape classes in the Shape Stacking Full split.

Action Observations: In Shape Stacking the functionality of each action is characterized by the physical appearance of the shape. Thus, the action observations consist of images of the shape from various camera angles and heights. Each shape has 1,024 observed images of resolution 84x84. Examples of these action observations are shown at `https://sites.google.com/view/action-generalization/shape-stacking`.

B. Visualizing Action Representations

In this work, we train and evaluate a wide variety of action representations based on environments, data-modality, presence or absence of hierarchy in action encoder, and different action splits. We describe these in detail and provide t-SNE visualizations of the inferred action representations of previously unseen actions. These visualizations show how our model can extract information about properties of the actions, by clustering similar actions together in the latent space. Unless mentioned otherwise, the HVAE model is used to produce these representations.

Grid World: Figure 4 shows the inferred action or skill representations in Grid World. The actions are colored by the relative change in the location of the agent after applying the skill. For example, the skill ”Up, Up, Up, Right, Down” would translate the agent to the upper right quadrant from the origin, hence visualized in red color. All learned action representations are 16-dimensional. We plot the following action representations:

- State Trajectories (default): HVAE encodes action observations consisting of trajectories of 2D (x, y) coordinates of the agent on the 80x80 grid.
- Non-Hierarchical VAE (baseline): A standard VAE encodes all the state-based action observations individually, and then computes the action representation by taking their mean.
- One-hot (alternate): State is represented by two 80-dimensional one-hot vectors of the agent’s x and y coordinates on the 80x80 grid. Reconstruction is based on a softmax cross-entropy loss over the one-hot observations in the trajectory.
- Engineered (alternate): These are 5-dimensional representations containing the ground-truth knowledge of the five moves (up, down, left, right) that constitute a skill. The clustering of our learned representations looks com-

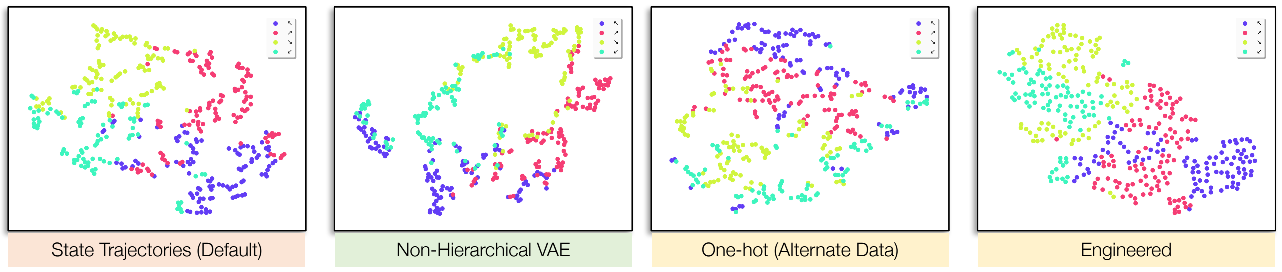


Figure 4. t-SNE Visualization of learned skill representation space for Grid World environment. Colored by the quadrant that the skill translates the agent to.

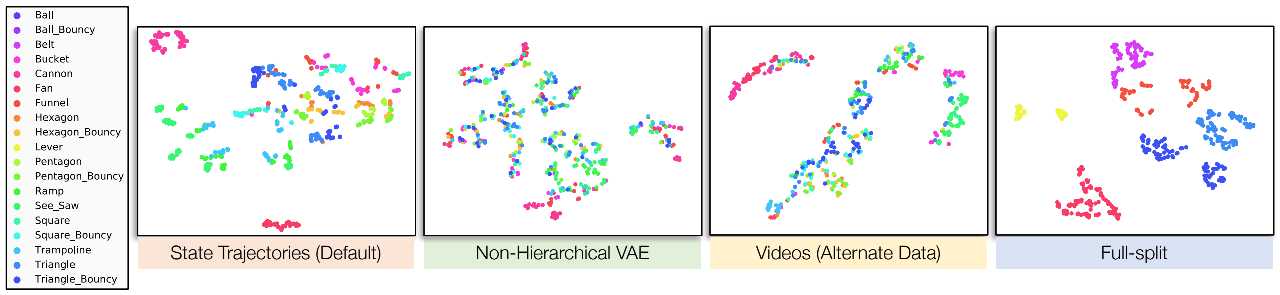


Figure 5. t-SNE Visualization of learned tool representation space for CREATE environment. Colored by the tool class.

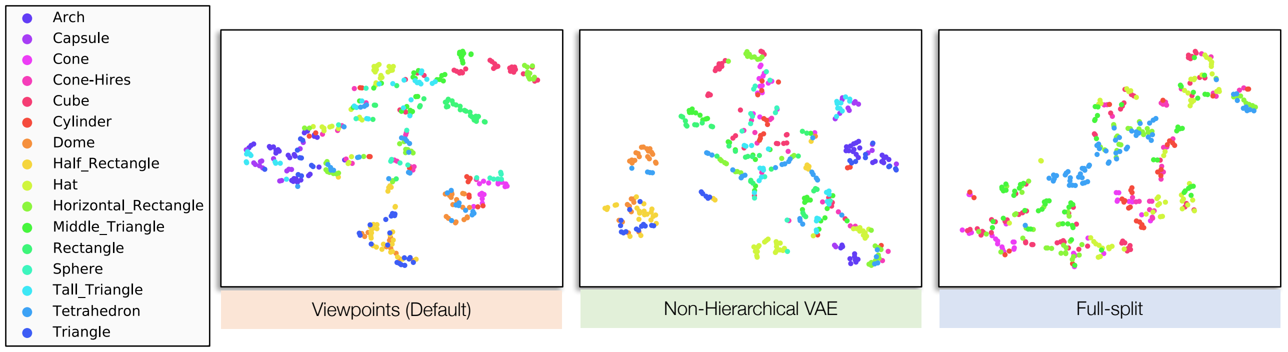


Figure 6. t-SNE Visualization of learned action representation space for the Shape-stacking environment. Colored by the shape class.

parable to these oracle representations.

CREATE: Figure 5 shows the inferred action or tool representations in CREATE. The actions are colored by tool class. All action representations are 128-dimensional.

- State Trajectories (default): HVAE encodes action observation data composed of (x, y) coordinate states of the probe ball's trajectory.
- Non-Hierarchical VAE (baseline): A standard VAE encodes all the state-based action observations individually, and then computes the action representation by taking their mean.
- Video (alternate): HVAE encodes action observation data composed of 84x84 grayscale image-based trajectories (videos) of the probe ball interacting with the tool. The data is collected identically as the state case, only the modality changes from state to image frames.
- Full Split: HVAE encodes state-based action observations, however, the training and testing tools are from the *Full Split* experiment. The visualizations show that even though training tools are vastly different from evaluation tools, HVAE generalizes and clusters well on unseen tools.

Shape Stacking: Figure 6 shows the inferred action representations in Shape Stacking. The shapes are colored according to shape class. All action representations are

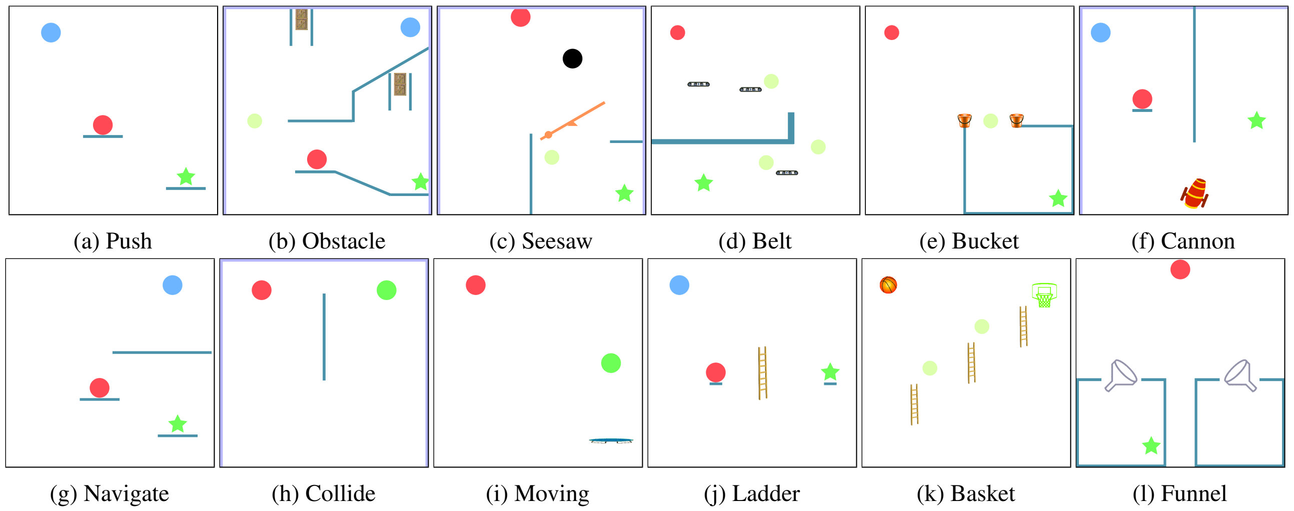


Figure 7. 12 CREATE tasks. Results on (a) - (c) are in the main paper, while (d) - (l) with our method are in Figure 8.

128-dimensional.

- Viewpoints (default): HVAE encodes action observations in the form of viewpoints of the shape from different camera angles and positions.
- Non-Hierarchical VAE (baseline): A standard VAE encodes all the image-based action observations individually, and then computes the action representation by taking their mean.
- Full-split: The training and testing tools are from the *Full Split* experiment. Previously unseen shape types are clustered well, showing the robustness of HVAE.

C. Further Experimental Results

C.1. Additional CREATE Results

Figure 7 visually describes all the CREATE tasks. The objective is to make the target ball (red) reach the goal (green), which may be fixed or mobile. Figure 8 demonstrates our method's results on the remaining nine CREATE tasks (the initial three tasks are in the main paper). Strong training and testing performance on a majority of these tasks shows the robustness of our method. The developed CREATE environment can be easily modified to generate more such tasks of varying difficulties. Due to the diverse set of tools and tasks, we propose CREATE and our results as a useful benchmark for evaluating action space generalization in reinforcement learning.

C.2. Additional Finetuning Results

We present additional results of finetuning and training from scratch to adapt to unseen actions across all CREATE Obstacle, CREATE Seesaw, Shape Stacking, Grid World, and Recommender. In the results presented in Figure 9, we observe the same trend holds where additional training takes many steps to achieve the performance our method obtains zero-shot.

C.3. CREATE: No Subgoal Reward

To verify our method's robustness, we also run experiments on a version of the CREATE environment without the subgoal rewards. The results in Figure 10 verify that even without reward engineering, our method exhibits strong generalization, albeit with higher variance in train and testing performance.

C.4. Auxiliary Policy Alternative Architecture

While in our framework, the auxiliary policy is computed from the state encoding alone, here we compare to also taking the selected discrete-action as input to the auxiliary policy. Comparison of this alternative auxiliary policy to the auxiliary policy from the main paper is shown in Figure 11. There are minimal differences in the average success rates of the two design choices.

C.5. Fully Observable Recommender System

Figure 12 demonstrates our method in a fully observable recommender environment where the product constant μ_i from Eq. 2 is also included in the engineered action representation. All methods achieve better training and generalization performance compared to the original partially observable Recommender System environment. However, full observability is infeasible in practical recommender systems. Therefore, we report the main results on the partially observed environment.

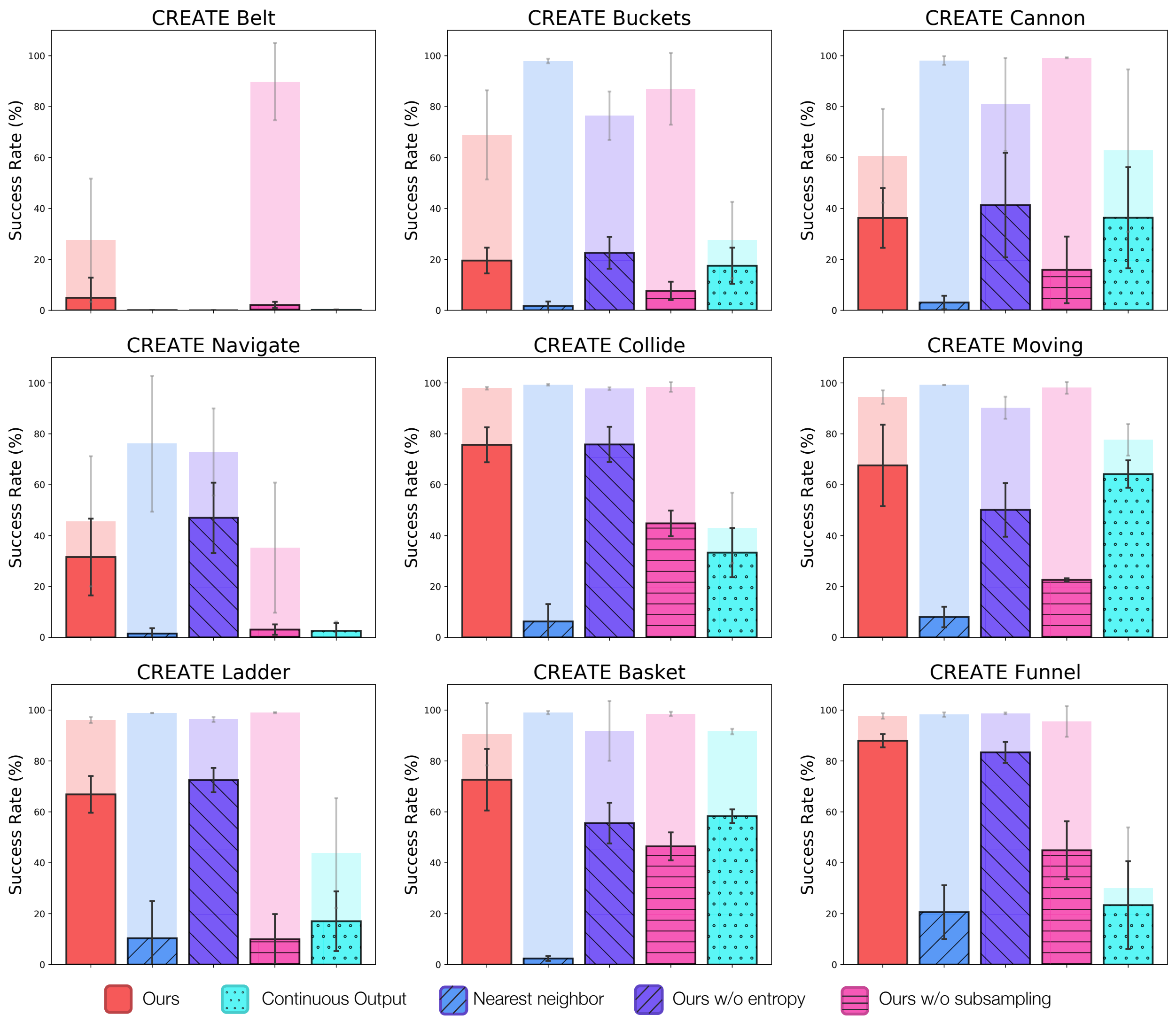


Figure 8. Results on the remaining 9 CREATE tasks with the same evaluation details as the main paper. We compare to both the baselines and ablations from the main paper.

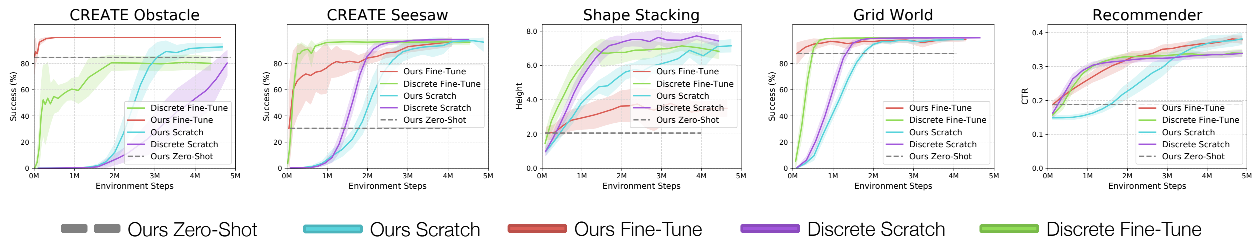


Figure 9. Finetuning or training the policy from scratch on the new action space across the remaining 5 tasks not present in the main paper. The evaluation settings are the same as in the main paper.

C.6. Additional Shape Stacking Results

Figure 13 demonstrates performance on different shape placement strategies in Shape Stacking using our framework. In *No Place*, the shapes are dropped at the center of the table, and the agent only selects which shape to drop from the available set. Since there are two randomly placed cylinders on the table, this setting of drop-

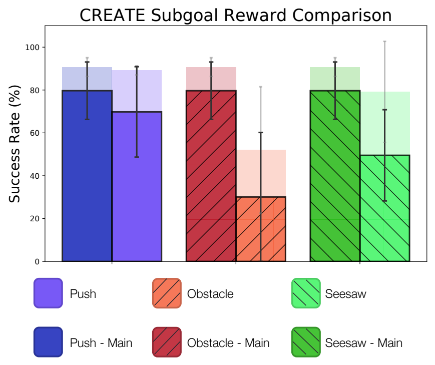


Figure 10. Comparison of a version of CREATE that does not use subgoal rewards. The “Main” methods are from the main paper using subgoal rewards.

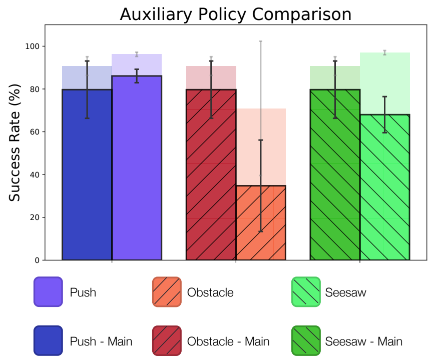


Figure 11. Comparison of an alternative auxiliary network architecture that is conditioned on the selected discrete action. The “Main” results are the default results that do not condition the auxiliary policy on the selected action.

ping in the center gives less control to the agent while stacking tall towers. Thus we report default results on *1D Place*, where the agent outputs in a hybrid action space consisting of shape selection and 1D placement through x-coordinate of the dropping location. The y-coordinate of the drop is fixed to the center. Finally, in *2D Place*, the agent decides both x and y coordinates to have more control but makes the task more challenging due to the larger search space. The evaluation videos of these new settings are available on https://sites.google.com/view/action-generalization/shape-stacking.

Figure 13 also shows the results of our method trained and evaluated on *Full Split* which was introduced in Table 2. Poor performance on this split could be explained by the policy not seeing enough shape classes during training to be able to generalize well to new shape classes during testing. This is also expected since this split severely breaks the i.i.d. assumption essential for generalization (Bousquet et al., 2003).

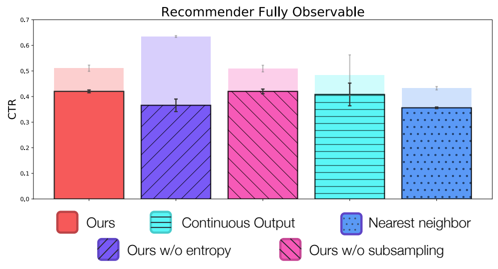


Figure 12. Training and testing results on the fully observable version of Recommender System with standard evaluation settings.

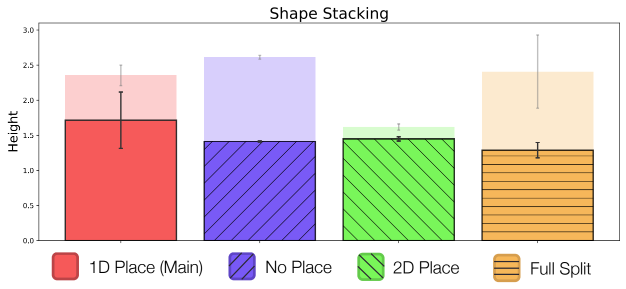


Figure 13. Comparing different placement strategies in shape stacking and showing performance on the *Full Split* action split. Results are using our main method with the standard evaluation details.

C.7. Learning Curves

Figure 14 show the training and validation performance curves for all methods and environments to contrast the training process of a policy against the objective of generalization to new actions. The plots clearly show how the generalization gap varies over the training of the policy. Ablation curves (last two columns) for some environments depict that an increase in training performance corresponds to a drop in validation performance. This is attributed to the policy overfitting to the training set of actions, which is often observed in supervised learning. Our proposed regularizing training procedure aims to avoid such overfitting.

D. Experiment Details

D.1. Implementation

We use PyTorch (Paszke et al., 2017) for our implementation, and the experiments were primarily conducted on workstations with 72-core Intel Xeon Gold 6154 CPU and 4 NVIDIA GeForce RTX 2080 Ti GPUs. Each experiment seed takes about 6 hours (Recommender) to 25 hours (CREATE) to converge. For logging and tracking experiments, we use the Weights & Biases tool (Biewald, 2020). All the environments were developed using the OpenAI Gym interface (Brockman et al., 2016). The HVAE implemen-

Hyperparameter	Grid world	Recommender	CREATE	Shape Stacking
HVAE				
action representation size	16	16	128	128
batch size	128	-	128	32
epochs	10000	-	10000	5000
Policy				
entropy coefficient	0.05	0.01	0.005	0.01
observation space	81	16	$84 \times 84 \times 3$	$84 \times 84 \times 4$
actions per episode	50	500	50	20
total environment steps	4×10^7	4×10^7	6×10^7	3×10^6
max. episode length	10	100	30	10
continuous entropy scaling	-	-	0.1	0.1
PPO batch size	4096	2048	3072	1024

Table 3. Environment-specific hyperparameters

tation is based on the PyTorch implementation of Neural Statistician (Edwards & Storkey, 2017), and we use RAdam optimizer (Liu et al., 2019). For training the policy network, we use PPO (Schulman et al., 2017; Kostrikov, 2018) with the Adam optimizer (Kingma & Ba, 2015). Further details can be found in the supplementary code.[1]

D.2. Hyperparameters

The default hyperparameters shared across all environments are shown in Table 4 and environment-specific hyperparameters are given in Table 3. We perform linear decay of the learning rate over policy training.

Hyperparameter	Value
HVAE	
learning rate	0.001
action observations	1024
MLP hidden layers	3
q_ϕ hidden layer size	128
default hidden layer size	64
Policy	
learning rate	0.001
discount factor	0.99
parallel processes	32
hidden layer size	64
value loss coefficient	0.5
PPO epochs	4
PPO clip parameter	0.1

Table 4. General Hyperparameters

[1]Code available at `https://github.com/clvrai/new-actions-rl`

D.2.1. Hyperparameter Search

Initial HVAE hyperparameters were inherited from the implementation of Edwards & Storkey (2017) and PPO hyperparameters from Kostrikov (2018). The hyperparameters were finetuned to optimize the performance on the held-out validation set of actions. Certain hyperparameters were sensitive to the environment or the method being trained and were searched for more carefully.

Specifically, entropy coefficient is a sensitive parameter to appropriately balance the ease of reward maximization during training versus the generalizability at evaluation. For each method and environment, we searched for entropy coefficients in subsets of $\{0.0001, 0.001, 0.005, 0.01, 0.05, 0.1\}$, and selected the best parameter based on the performance on the validation set. We found PPO batch size to be an important parameter affecting the speed of convergence, convergence value, and variance across seeds. Thus, we searched for the best value in $\{1024, 2048, 3072, 4096\}$ for each environment. Total environment steps are chosen so all the methods and baselines can run until convergence.

D.3. Network Architectures

D.3.1. Hierarchical VAE

Convolutional Encoder: When the action observation data is in image or video form, a convolution encoder is applied to encode it into a latent state or state-trajectory. Specifically, for CREATE video case, each action observation is a 48x48 grayscale video. Thus, each frame of the video is encoded through a 7-layer convolutional encoder with batch norm (Ioffe & Szegedy, 2015). Similarly, for Shape Stacking, the action observation is an 84x84 image, that is encoded through 9 convolutional layers with batch norm.

Bi-LSTM Encoder: When the data is in trajectory form (as in CREATE and Grid World), the sequence of states are encoded through a 2-layer Bi-LSTM encoder. For CREATE video case, the encoded image frames of the video are passed through this Bi-LSTM encoder in place of the raw state vector. After this step, each action observation is in the form of a 64-dimensional encoded vector.

Action Inference Network: The encoded action observations are passed through a 4-layer MLP with ReLU activation, and then aggregated with mean-pooling. This pooled vector is passed through a 3-layer MLP with ReLU activation, and then 1D batch-norm is applied. This outputs the mean and log-variance of a Gaussian distribution q_ϕ, which represents the entire action observation set, and thus the action. This is then used to sample an action latent to condition reconstruction of individual observations.

Observation Inference Network: The action latent and individual encoded observations are both passed through linear layers and then summed up, and followed by a ReLU nonlinearity. This combined vector is then passed through two 2-layer MLPs with ReLU followed by a linear layer, to output the mean and log-variance of a Gaussian distribution, representing the individual observation conditioned on the action latent. This is used to sample an observation latent, which is later decoded back while being conditioned on the action latent.

Observation Decoder: The sampled observation latent and its action latent are passed through linear layers, summed and then followed by a nonlinearity. For non-trajectory data (as in Shape Stacking), this vector is then passed through a 3-layer MLP with ReLU activation to output the decoded observation's mean and log-variance (i.e. a Gaussian distribution). For trajectory data (as in CREATE and Grid World), the initial ground truth state of the trajectory is first encoded with a 3-layer MLP with ReLU. Then an element-wise product is taken with the action-observation combined vector. The resulting vector is then passed through an LSTM network to produce the latents of future states of the trajectory. Each future state latent of the trajectory goes through a 3-layer MLP with ReLU, to result in the mean and log-variance of the decoded trajectory observation (i.e. a Gaussian distribution).

Convolutional Decoder: If the observation was originally an image or video, then the mean of the reconstructed observation is converted into pixels through a convolutional decoder consisting of 2D convolutional and transposed-convolutional layers. For the case of video input, the output of the convolutional decoder is also channel-wise augmented with with a 2D pixel mask. This mask is multiplied with the mean component of the image output (i.e. log-variance output stays the same), and then added to the initial frame of the video. This is the temporal skip connection technique (Ebert et al., 2017), which eases the learning process with high-dimensional video observation datasets.

Finally, the reconstruction loss is computed using the Gaussian log-likelihood of the input observation data with respect to the decoded distribution.

D.3.2. Policy Network

State Encoder f_ω: When the input state is in image-form (channel-wise stacked frames in CREATE and Stacking), f_ω is implemented with a 5-layer convolutional network, followed by a linear layer and ReLU activation function. When the input is not an image, we use 2-layer MLP with tanh activation to encode the state.

Critic Network V: For image-based states, the output of the state encoder f_ω is passed through a linear layer to result in the value function of the state. This is done to share the convolutional layers between the actor and critic. For non-image states, we use 2-layer MLP with tanh activation, followed by a linear layer to get the state's value.

Utility Function f_ν: Each available action's representation c is passed through a linear layer and then concatenated with the output of the state encoder. This vector is fed into a 2-layer MLP with ReLU activation to output a single logit for each action. The logits of all the available actions are then stacked and input to a Categorical distribution. This acts as the policy's output and is used to sample actions, compute log probabilities, and entropy values.

Auxiliary Policy f_χ: The output of the state encoder is also separately used to compute auxiliary action outputs. For CREATE and Shape Stacking, we have a 2D position action in $[-1, 1]$. For such constrained action space, we use a Beta distribution whose α and β are computed using linear layers over the state encoding. Concretely, $\alpha = 1 + \text{softplus}(\text{fc}_\alpha(f_\omega(s))$ and $\beta = 1 + \text{softplus}(\text{fc}_\beta(f_\omega(s))$, to ensure their values lie in [1, ∞]. This in turn ensures that the Beta distribution is unimodal with values constrained in [0,1] (as done in (Chou et al., 2017)), which we then convert to [-1,1]. The Shape Stacking environment also has a binary termination action for the agent. This is implemented by passing the state encoding through a linear layer which outputs two logits (for continuation/termination) of a Categorical distribution. The auxiliary action distributions are combined with the main discrete action Categorical distribution from f_ν. This overall distribution is used to sample hybrid actions, compute log probabilities, and entropy values. Note, the entropy value of the Beta distribution is multiplied by a scaling factor of 0.1, for better convergence.

References

Allen, K. R., Smith, K. A., and Tenenbaum, J. B. The tools challenge: Rapid trial-and-error learning in physical problem solving. *arXiv preprint arXiv:1907.09620*, 2019. 2

Bakhtin, A., van der Maaten, L., Johnson, J., Gustafson, L., and Girshick, R. Phyre: A new benchmark for physical reasoning. *arXiv:1908.05656*, 2019. 2

Biewald, L. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com. 8

Blomqvist, V. Pymunk. URL `http://www.pymunk.org/`. Accessed 2020-02-18. 2

Bousquet, O., Boucheron, S., and Lugosi, G. Introduction to statistical learning theory. In *Summer School on Machine Learning*, pp. 169–207. Springer, 2003. 8

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 8

Chevalier-Boisvert, M., Willems, L., and Pal, S. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018. 1

Chou, P.-W., Maturana, D., and Scherer, S. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *International Conference on Machine Learning*, pp. 834–843, 2017. 10

Ebert, F., Finn, C., Lee, A. X., and Levine, S. Self-supervised visual planning with temporal skip connections. In *Conference on Robot Learning*, pp. 344–356, 2017. 10

Edwards, H. and Storkey, A. Towards a neural statistician. In *International Conference on Learning Representations*, 2017. URL `https://openreview.net/forum?id=HJDBUF5le`. 9

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 9

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. 9

Kostrikov, I. Pytorch implementations of reinforcement learning algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018. 9

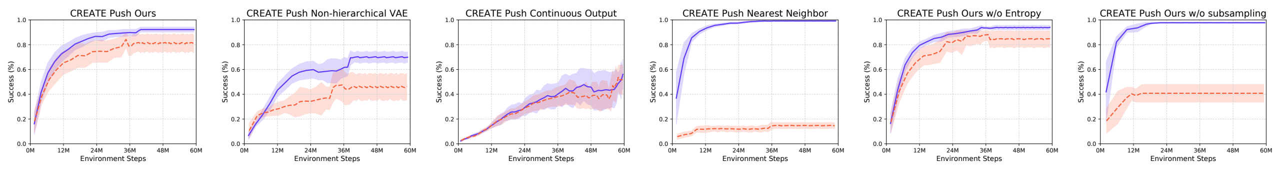
Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., and Han, J. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019. 9

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017. 8

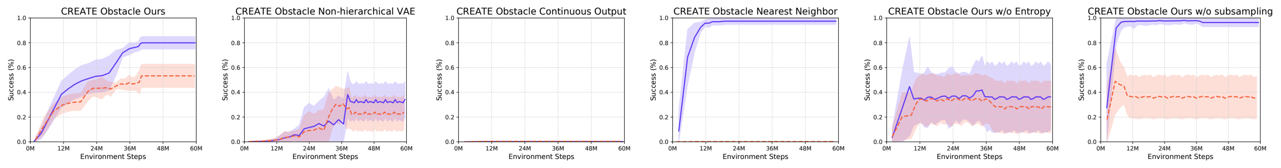
Rohde, D., Bonner, S., Dunlop, T., Vasile, F., and Karatzoglou, A. Recogym: A reinforcement learning environment for the problem of product recommendation in online advertising. *arXiv preprint arXiv:1808.00720*, 2018. 1, 2

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 9

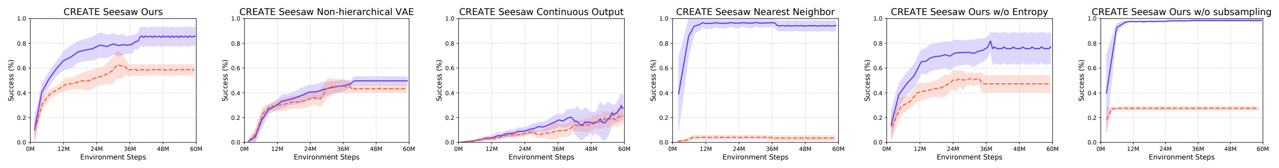
Shinners, P. Pygame. URL `http://pygame.org/`. Accessed 2020-02-18. 2



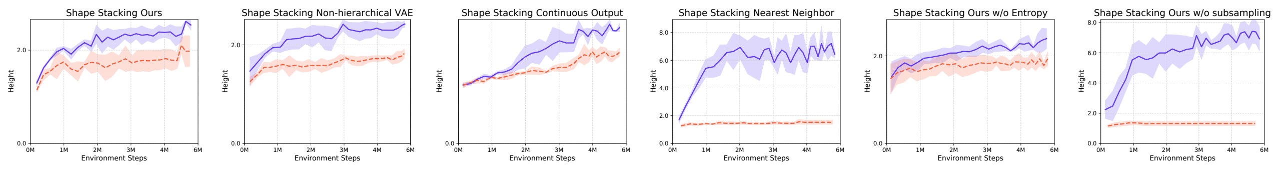
(a) CREATE Push



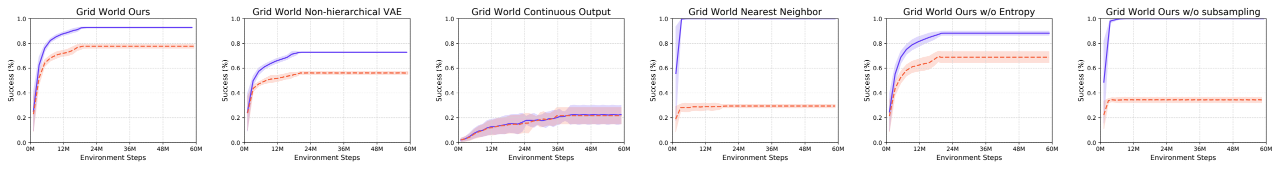
(b) CREATE Obstacle



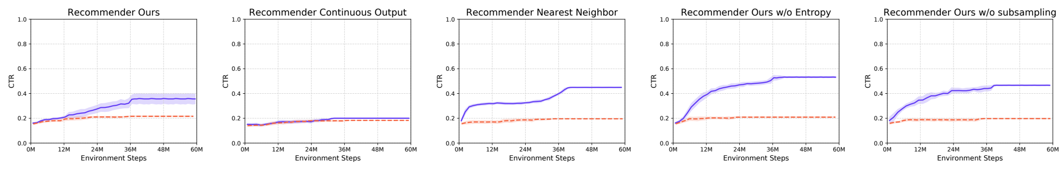
(c) CREATE Seesaw



(d) Shape Stacking



(e) Grid World



(f) Recommender System

🟪 Training 🟥 Validation

Figure 14. Learning curves for all environments and methods showing performance on both the training and validation sets. Each line shows the performance of 5 random seeds (8 for Grid World) as average value and the shaded region as the standard deviation.