

Decision Making in Complex Action Spaces

by

Ayush Jain

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2024

Acknowledgements

First and foremost, I want to thank my family for their incredible support and sacrifices throughout my academic life and PhD journey. My father, Jitendra Kumar Jain, for being a constant source of inspiration for hard work, ambition, confidence, and doing something big. My mother, Sannu Jain, for her selflessness, unconditional love, and being the person whom I strive to make proud. My sister, Sanjana Jain, for unwaveringly believing in me. And to my partner and best friend, Yashee Mathur, for inspiring me to become the best version of myself.

I am immensely grateful to my Ph.D. advisors, Joseph Lim and Erdem Bıyık for their mentorship. I want to thank Joseph for taking a chance on me, building me up, and always motivating me to aim higher. His empathy during my toughest times and remarkable ability to understand more about me than myself have been invaluable in my personal and academic growth over the past six years. I want to thank Erdem for welcoming me into his new lab, for his constant availability, and for the kindness he has shown throughout.

I wish to express my gratitude to my dissertation committee members, Gaurav Sukhatme, Stefanos Nikolaidis, and Feifei Qian, for providing guidance and insights on multiple occasions. I thank Jesse Thomason and Yan Liu for serving on my qualification exam committee. The support I received from the USC CS department through funding and teaching assistantships has been instrumental, and I am especially grateful to Lizsl, Andy, Asiroh, and Ellecia.

I have been incredibly fortunate to work alongside friends and colleagues at CLVR and LiraLab who made research an engaging and fulfilling experience. I learned the importance of first-principles thinking, articulate communication, and structured research from Youngwoon Lee, Shao-Hua Sun, and Karl Pertsch. Special thanks to my first-ever labmates, Telin Wu, Andrew Szot, and Edward

Hu, who helped shape my first experiences in research. Post-Covid, when I returned to the US after a break, I was lucky to have found myself surrounded by energetic and encouraging peers, Jesse Zhang, Graze Zhang, Anthony Liang, Jiahui Zhang, Xihu Li, Ziyi Liu, and many others at Lira and Glamor labs.

I have had amazing, diverse collaborators with whom I learned various aspects of research throughout my Ph.D. In particular, I would like to thank Andrew Szot, Norio Kosaka, Graze Zhang, Xihu Li, and Shao-Hua Sun for their major contributions. I also cherish the discussions and collaborations with Edward Hu, Doohyun Lee, Minho Heo, Sungjae Park, Lucy Shi, Shubham Sharma, Haeone Lee, Junyeob Baek, Ryan Lindeborg, Yigit Korkmaz, Urvi Bhuvania, and Zhaojing Yang. I am grateful for the opportunities to work outside of academia as well, having interned at Meta Reality Labs with Nitin Kamra, Microsoft Research Canada with Marc-Alexandre Côté and Eric Yuan, and Naver AI Labs with Kyung-Min Kim. These enriched my research experience in diverse environments and helped broaden my perspectives on research.

Finally, a heartfelt thanks to my friends, who have been my constant support system. Umang, Shobhit, Mohit, Kruttika, Abhinav, Arya, Gautam, Shihan, Sumedh, and many others — your friendship has been a source of joy and balance in my life. A special mention goes to Nitin, who not only introduced me to my Ph.D. advisor, Joseph, but also has been a steadfast friend and supporter throughout my journey.

TABLE OF CONTENTS

Acknowledgements	ii
List of Tables	x
List of Figures	xi
Abstract	xix
Chapter 1: Introduction	1
1.1 Generalization to Unseen and Varying Action Spaces	3
1.2 Near-optimal Action in Non-convex Q-functions	4
I Generalization to Unseen and Varying Action Spaces	5
Chapter 2: Generalization to New Actions via Action Representations	6
2.1 Introduction	6
2.2 Related Work	8
2.3 Problem Formulation	9
2.3.1 Reinforcement Learning	10
2.3.2 Generalization to New Actions	10
2.4 Approach	11
2.4.1 Unsupervised Learning of Action Representations	12
2.4.2 Adaptable Policy Architecture	13
2.4.3 Generalization Objective and Training Procedure	14
2.5 Experimental Setup	16
2.5.1 Environments	16
2.5.2 Experiment Procedure	18
2.5.3 Baselines	18
2.5.4 Ablations	19
2.6 Results and Analysis	19
2.6.1 Visualization of Inferred Action Representations	20
2.6.2 Results and Comparisons	20
2.6.3 Analyzing the Limits of Generalization	22

2.6.4	The Inefficiency of Finetuning on New Actions	25
2.7	Conclusion	26
Chapter 3:	Know Your Action Set in Varying Action Spaces via Action Relations	27
3.1	Introduction	27
3.2	Related Work	29
3.3	Problem Formulation	30
3.3.1	Reinforcement Learning with Varying Action Space	31
3.3.2	Challenges of Varying Action Space	31
3.4	Approach	32
3.4.1	AGILE: Action Graph for Interdependence Learning	32
3.4.2	Training AGILE framework with Reinforcement Learning	34
3.5	Environments	35
3.5.1	Dig Lava Grid Navigation	35
3.5.2	Chain REAction Tool Environment: CREATE	35
3.5.3	Recommender Systems	36
3.6	Experiments	38
3.6.1	Effectiveness of AGILE in Varying Action Spaces	38
3.6.2	Does the Attention in AGILE Learn Meaningful Action Relations?	41
3.6.3	Additional Analyses	42
3.7	Conclusion	43
II	Near-optimal Action in Non-convex Q-functions	44
Chapter 4:	Optimizing Action in Non-Convex Q-functions via Successive Actors	45
4.1	Introduction	45
4.2	Related Work	47
4.3	Problem Formulation	49
4.3.1	Deterministic Policy Gradients (DPG)	50
4.3.2	The Challenge of an Actor Maximizing a Complex Q-landscape	50
4.4	Approach: Successive Actors for Value Optimization (SAVO)	52
4.4.1	Maximizer Actor over Action Proposals	53
4.4.2	Successive Surrogates to Reduce Local Optima	53
4.4.3	Successive Actors for Surrogate Optimization	55
4.4.4	Approximate Surrogate Functions	55
4.4.5	SAVO-TD3 Algorithm and Design Choices	56
4.5	Environments	58
4.6	Experiments	59
4.6.1	Effectiveness of SAVO in challenging Q-landscapes	59
4.6.2	Q-Landscape Analysis: Do successive surrogates reduce local optima?	61
4.6.3	Challenging Dexterous Manipulation (Adroit)	62
4.6.4	Quantitative Analysis: The Effect of Successive Actors and Surrogates	63
4.6.5	Does RL with Resets address the issue of Q-function Optimization?	63

4.6.6	Further experiments to validate SAVO	64
4.7	Limitations and Conclusion	65
Chapter 5: Sharing Actions in Multi-Task Reinforcement Learning via Q-switch Mixture of Policies 66		
5.1	Introduction	66
5.2	Related Work	68
5.3	Problem Formulation	70
5.4	Approach	71
5.4.1	Multi-Task Behavior Sharing via Off-Policy Data Collection	71
5.4.2	Q-switch Mixture of Policies (QMP)	72
5.5	Why QMP Works: Theory and Didactic Example	74
5.5.1	QMP: Convergence and Improvement Guarantees	74
5.5.2	Illustrative Example: 2D Point Reaching	75
5.6	Experiments	77
5.6.1	Environments	77
5.6.2	Baselines	78
5.7	Results	79
5.7.1	Is Behavior Sharing Complementary to other MTRL frameworks?	79
5.7.2	Baselines: Comparing Different Approaches to Share Behaviors	81
5.7.3	Can QMP effectively identify shareable behaviors?	82
5.7.4	Ablations	83
5.8	Conclusion	84
Chapter 6: Conclusion 85		
6.1	Open Challenges and Future Directions	86
References 89		
Appendices 106		
Appendix A 108		
Generalization to New Actions via Action Representations 108		
A	Environment Details	108
A.1	Grid World	108
A.2	Recommender System	110
A.3	Chain REAction Tool Environment (CREATE)	112
A.4	Shape Stacking	115
B	Visualizing Action Representations	116
C	Further Experimental Results	120
C.1	Additional CREATE Results	120
C.2	Additional Finetuning Results	121
C.3	CREATE: No Subgoal Reward	121
C.4	Auxiliary Policy Alternative Architecture	121
C.5	Fully Observable Recommender System	121

C.6	Additional Shape Stacking Results	122
C.7	Learning Curves	123
D	Experiment Details	124
D.1	Implementation	125
D.2	Hyperparameters	126
D.2.1	Hyperparameter Search	126
D.3	Network Architectures	127
D.3.1	Hierarchical VAE	127
D.3.2	Policy Network	128
Appendix B	131
Know Your Action Set in Varying Action Spaces via Action Relations	131
A	Environment Details	131
A.1	Dig Lava Grid Navigation	131
A.2	Chain REAction Tool Environment (CREATE)	132
A.3	RecSim	135
A.4	Real-data recommender system	137
B	Further Experimental Results	140
B.1	Effect of using domain knowledge in action graph edges	140
B.2	Validating design choices for value-based AGILE	141
B.2.1	Hyper-parameter Search in AGILE	142
B.2.2	Design Choices of AGILE	143
B.3	Effect of direct v/s indirect reward in RecSim	144
B.4	Recsim-Pairing Environment	144
C	Approach and Baseline Details	145
C.1	Details of Baselines and Ablations	145
C.2	Details on Listwise AGILE	148
C.3	Network Architectures	150
C.3.1	Action Graph	150
C.3.2	Summarizers: Bi-LSTM and Deep Set	150
C.3.3	Utility Network	151
C.3.4	Action Representation Network	151
C.3.5	Reward Inference Network (User Model in Real World RecSys)	152
D	Experiment Details	152
D.1	Implementation Details	152
D.2	Hyperparameters	153
D.3	Hyperparameter Tuning	154
Appendix C	157
Optimizing Action in Non-Convex Q-functions via Successive Actors	157
A	Proof of Convergence of Maximizer Actor in Tabular Settings	157
B	Proof of Reducing Number of Local Optima in Successive Surrogates	160
C	Environment Details	163
C.1	MiningEnv	163
C.2	RecSim	165

C.3	Continuous Control	166
C.3.1	Restricted Locomotion in Mujoco	167
D	Additional Results	168
D.1	Experiments on Continuous Control (Unrestricted Mujoco)	168
D.2	Per-Environment Ablation Results	169
D.3	SAC is Orthogonal to SAVO	170
D.4	Increasing Size of Discrete Action Space in RecSim	171
E	Validating SAVO Design Choices	172
E.1	Design Choices: Action summarizers	172
E.2	Conditioning on Previous Actions: FiLM vs. MLP	173
E.3	Exploration Noise comparison: OUNoise vs Gaussian	173
F	Network Architectures	174
F.1	Successive Actors	174
F.2	Successive Surrogates	175
F.3	List Summarizers	175
F.4	Feature-wise Linear Modulation (FiLM)	176
G	Experiment and Evaluation Setup	176
G.1	Aggregated Results: Performance Profiles	176
G.2	Implementation Details	177
G.3	Common Hyperparameter Tuning	177
G.4	Hyperparameters	178
H	Q-Value Landscape Visualizations	178
H.1	1-Dimensional Action Space Environments	178
H.2	High-Dimensional Action Space Environments	179
Appendix D		183
Sharing Actions in Multi-Task Reinforcement Learning via Q-switch Mixture of Policies		183
A	Qualitative Results	183
B	QMP Derivation	183
C	QMP Convergence Guarantees	184
D	Environment Details	189
D.1	Multistage Reacher	189
D.2	Maze Navigation	191
D.3	Meta-World Manipulation	191
D.4	Walker2D	192
D.5	Kitchen	193
E	Additional Results	193
E.1	Multistage Reacher Per Task Results	193
E.2	Data Sharing Results	194
E.3	PCGrad Results	195
E.4	QMP Scales with Task Set Size in Maze Navigation	196
E.5	Additional Comparisons	196
E.6	Temporally-Extended Behavior Sharing	198
F	QMP Behavior Sharing Analysis	199
F.1	Qualitative Visualization of Behavior-Sharing	200

G	Additional Ablations and Analysis	201
G.1	Probabilistic Mixture v/s Arg-Max	201
G.2	Approximation Expected Q-value Over Policy Action Distribution	202
G.3	QMP v/s Increasing Single Task Exploration	202
G.4	QMP Runtime	203
H	Implementation Details	204
H.1	Hyperparameters	204
H.2	No-Shared-Behaviors	204
H.3	Fully-Shared-Behaviors	204
H.4	DnC	205
H.5	QMP (Ours)	206
H.6	Online UDS	206

List of Tables

4.1	Compute v/s Performance Gain (Mujoco)	65
A.1	Tool classes in the CREATE Full split.	114
A.2	Shape classes in the Shape Stacking Full split.	116
A.3	Environment-specific hyperparameters for Grid World, Recommender, CREATE, and Shape Stacking tasks relevant to HVAE and policy.	124
A.4	General Hyperparameters for HVAE and policy shared across environments	126
B.1	Action representations for the skills used in Dig Lava Grid Navigation environment.	133
B.2	Action representations for the tools in CREATE environment.	135
B.3	AGILE, baseline and ablations: Hyperparameters for additional components	153
B.4	Environment/Policy-specific Hyperparameters for RL training in AGILE	156
C.1	Environment/Policy-specific Hyperparameters for SAVO	178
D.1	Subgoal Positions for Each Task in Multistage Reacher	189
D.2	Temporally Extended Behavior Sharing Results	199
D.3	Runtime Comparison of MTRL frameworks with and without QMP	203
D.4	Hyperparameters for QMP and baselines.	205

List of Figures

2.1	An illustration of zero-shot generalization to new actions in a sequential decision-making task, CREATE. (Left) Learning to select and place the right tools for reaching the goal. (Right) Generalizing the learned policy to a previously unseen set of tools.	6
2.2	Two-stage framework for generalization to new actions through action representations. (1) For each available action, a hierarchical VAE module encodes the action observations into action representations and is trained with a reconstruction objective. (2) The policy π_θ encodes the state with state encoder $f_\omega(s)$ and pairs it with each action representation using the utility function f_ν . The utility scores are computed for each action and output to a categorical distribution. The auxiliary network takes the encoded state and outputs environment-specific auxiliary actions such as tool placement in CREATE. The policy architecture is trained with policy gradients.	11
2.3	Benchmark environments for evaluating generalization to new actions. (Top) In CREATE, an agent selects and places various tools to move the red ball to the goal. Other moving objects can serve as help or obstacles. Some tasks also have subgoals to help with exploration (Appendix C.3 shows results with no subgoal rewards). The tool observations consist of trajectories of a test ball interacting with the tool. (Left) In Shape Stacking, an agent selects and places 3D shapes to stack a tower. The shape observations are images of the shape from different viewpoints. (Right) In Grid World, an agent reaches the goal by choosing from 5-step navigation skills. The skill observations are collected on an empty grid in the form of agent trajectories resulting from skill execution from random locations. . .	16
2.4	t-SNE visualization of action representations for held-out tools in CREATE inferred using a trained HVAE (left) and a VAE (right). The color indicates the tool class (e.g. cannons, buckets). The HVAE encoder learns to organize semantically similar tools together, in contrast to the flat VAE, which shows less structure.	20

2.5	Comparison against baseline action representation and policy architectures on 6 environments, 3 of which are CREATE tasks. The solid bar denotes the test performance and the transparent bar the training performance, to observe the generalization gap. The results are averaged over 5000 episodes across 5 random seeds, and the error bars indicate the standard deviation (8 seeds for Grid World). All learning curves are present in Figure A.14. Results on 9 additional CREATE tasks can be found in Appendix C.1.	21
2.6	Analyzing the importance of the proposed action space subsampling and entropy regularization in our method. Training and evaluation follow Figure 2.5.	22
2.7	Additional analyses. (a) Our method achieves decent performance on out-of-distribution tools in 3 CREATE tasks, but the generalization gap is more pronounced. (b) Various action representations can be successfully used with our policy architecture.	23
2.8	Varying the test action space. An increasing x-axis corresponds to more difficult generalization conditions. Each value plotted is the average test performance over 5 random seeds with the error bar corresponding to the standard deviation.	24
2.9	Evaluation results showing the trajectories of objects in CREATE and the final tower in Shape Stacking. Our framework is generally able to infer the dynamic properties of tools and geometry of shapes and subsequently use them to make the right decisions.	25
2.10	Finetuning or training policies from scratch on the new action space. The horizontal line is the zero-shot performance of our method. Each line is the average test performance over 5 random seeds, while the shaded region is the standard deviation.	26
3.1	Picture hanging task with varying sets of tool-actions. The strategy with each action set depends on all the pairwise action relations. (Left) The agent infers that a nail and a hammer are strongly related (bold line). Thus, it takes nail-action in Step-1 because hammer-action is available for Step-2. (Right) With a different action set, the nail-action is no longer useful due to the absence of a hammer. So, the agent must use an adhesive-tape in Step-1 since its related action of the hook is available for later use. We show that a policy with GAT over actions can learn such action relations.	27
3.2	Given an action set, AGILE builds a complete graph where each node is composed of an action representation and the state encoding. A graph attention network (GAT) learns action relations by attending to other relevant actions for the current state. For example, the attention weight between cannon and fire is high because fire can activate the cannon. The GAT outputs more informed relational action representations than the original inputs. Finally, a utility network computes each action’s value or selection probability in parallel using its relational action representation, the state, and a mean-pooled summary vector of all available actions’ relational features.	33

3.3	Environment Setup. (Left) In Grid World, the red agent must avoid orange and pink lava to reach the green goal. The <i>dig-lava</i> skills enable shortcut paths to the goal when available. (Middle) In CREATE, tools are selected and placed sequentially to push the red ball to the green goal. General tools (cannon, fan) require activator tools (fire, electric) to start functioning. The choice of general tools depends on what activators are available and vice-versa. (Right) In simulated and real-data recommender systems, the agent selects a list of items. Lists with coherent categories and complementary sub-categories improve user satisfaction (e.g., click likelihood).	36
3.4	We evaluate AGILE against baselines on training actions (top) and unseen testing actions (bottom). Generalization is enabled by continuous action representations, except Grid World (Appendix A). All architectures share the same RL algorithm (PPO or CDQN). The results are averaged over 5 seeds, with seed variance shown with shading. AGILE outperforms all baselines that assume a fixed action set (cannot generalize) or treat actions independently (suboptimal).	39
3.5	Test performance of AGILE against ablations with utility network using only action set summary, but not the relational action representations. The difference is most pronounced in CREATE, where there are several diverse action (tool) relations for each action decision.	40
3.6	Qualitative Analysis. (a,b) The attention maps from GAT show the reasoning behind an action decision. The nodes show available actions, and edge widths are proportional to attention weight (thresholded for clarity). (b) Utility Policy learns the same suboptimal solution for any given action set, while Summary-GAT (like AGILE) adapts to the best strategy by exploiting dig-skills. (c) AGILE can optimize CPR by identifying the most common item category available, unlike Utility Policy. We provide qualitative video results for Grid World and CREATE on the project page https://sites.google.com/view/varyingaction	41
3.7	Analyses. (i) GAT v/s GCN (ii) state-action relations v/s action-only relations.	42
4.1	An actor μ trained with gradient ascent on a challenging Q-landscape gets stuck in local optima. Our approach learns a sequence of surrogates Ψ_i of the Q-function that successively prune out the Q-landscape below the current best Q-values, resulting in fewer local optima. Thus, the actors ν_i trained to ascend on these surrogates produce actions with a more optimal Q-value.	45
4.2	Complex Q-landscapes. We plot Q-value versus action a for some state. In control of Inverted-Double-Pendulum-Restricted (left) and Hopper-Restricted (middle), certain action ranges are unsafe, resulting in various locally optimal action peaks. In a large discrete-action recommendation system (right), there are local peaks at actions representing real items (black dots).	46
4.3	Non-convex Q-landscape in Inverted-Pendulum-Restricted leads to a suboptimally converged actor.	51

4.4	SAVO Architecture. (left) Q-network is unchanged. (center) Instead of a single actor, we learn a sequence of actors and surrogate networks connected via action predictions. (right) Conditioning on previous actions is done with the help of a deep-set summarizer and FiLM modulation.	54
4.5	In restricted inverted pendulum, given an anchor $Q(a_0)$ value, Ψ (left) has some zero-gradient surfaces which $\hat{\Psi}$ (right) <i>approximately</i> follows while allowing non-zero gradients towards high Q-values to flow into its actor ν	56
4.6	Restricted Hopper’s 3D visualization of Action Space.	58
4.7	Aggregate performance profiles using normalized scores over 7 tasks and 10 seeds each.	60
4.8	SAVO against baselines on discrete and continuous tasks. Results over 10 seeds.	61
4.9	SAVO helps a single actor escape the local optimum a_0 in the Restricted Inverted Pendulum Task. Each successive surrogate learns a Q-landscape with fewer local optima and thus is easier to optimize by its actor.	62
4.10	SAVO improves the sample-efficiency of TD3 on Adroit dexterous manipulation tasks.	62
4.11	(L) More successive actor-surrogates are better, (R) SAVO v/s single-actor on inference.	63
4.12	Reset (primacy bias) does not improve Q-optimization.	64
5.3	QMP generalized policy iteration	75
5.5	QMP improves performance using other policies, increasingly so when task-relevant.	76
A.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).	108
A.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.	111
A.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.	111

A.4	t-SNE Visualization of learned skill representation space for Grid World environment. Colored by the quadrant that the skill translates the agent to.	116
A.5	t-SNE Visualization of learned tool representation space for CREATE environment. Colored by the tool class.	117
A.6	t-SNE Visualization of learned action representation space for the Shape-stacking environment. Colored by the shape class.	117
A.7	12 CREATE tasks. Complete results on (a) - (c) are in Figure 2.5, 2.6, while (d) - (l) are in Figure A.8.	118
A.8	Results on the remaining 9 CREATE tasks with the same evaluation details as the main paper (Figure 2.5). We compare our method against all the baselines (Section 2.5.3) and ablations (Section 2.5.4).	119
A.9	Finetuning or training the policy from scratch on the new action space across the remaining 5 tasks (Figure 2.10 only shows results on CREATE Push). The evaluation settings are the same as described in Section 2.6.4.	120
A.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 2.5).	122
A.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 (Figure 2.5).	123
A.12	Training and testing results on the fully observable version of Recommender System with standard evaluation settings.	124
A.13	Comparing different placement strategies in shape stacking and showing performance on the <i>Full Split</i> action split. Results are using our method with the standard evaluation details.	125
A.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.	130
B.1	CREATE Activator Mapping: Original CREATE tools (right) are activated by the respective newly introduced activator tools (left). E.g., a <i>Cannon</i> tool (abbreviated as CNN in attention maps) placed on the environment will only be functional when a <i>Fire</i> tool is placed in contact with it. Other objects are not affected by non-functional tools - they simply pass through.	134
B.2	t-SNE visualization of synthetically generated items in RecSim.	137

B.3	Real-Data Recommender System: F1 Score for Training/test user models. (Left) The online user model is trained on online-training data and evaluated on both online-training (green) and online-held-out (orange) data. (Right) The offline user model is trained on offline-training data and evaluated on both offline-training (green) and offline-held-out data (orange). Thus, the disparity between online data training and evaluation curves (left) shows that it is hard to train the online user model. In contrast, the offline user model generalizes reasonably well (right). . . .	138
B.4	(a) t-SNE visualization of split of train and test item representations. This shows that the train and test items are within the same distribution (b) Visualizations of item representations clustered according to the main category. This shows that item representations contain information about the main category, which is necessary to maximize CPR. (c) Item distribution labeled according to sub category, which is another information necessary for CPR.	139
B.5	Effect of using domain knowledge to predefine the possible relations via edges in the action graph of AGILE. We know that in the Grid Navigation task, only the action relations with respect to variable actions are important. Thus, we remove all the other edges. This makes the learning slightly faster and more stable as shown by a reduction in seed variance. (5 seeds)	140
B.6	Results of value-based AGILE on RecSim CPR task (a) hyperparameter testing and (b) validating architecture design choices, on train actions (left) and test actions (right).	141
B.7	Comparison against the baselines on train (top) and test (bottom) actions on the Direct CPR (left) and Pairing (right) RecSim environments. Along with Figure 3.4, these results exhibit the same trend that AGILE consistently outperforms all the baselines on both train and test actions.	142
B.8	Comparison against ablations on the Direct CPR (left) and Pairing (right) RecSim environments. Along with Figure 3.5, these results exhibit the same trend that AGILE slightly outperforms the various summary ablations, which do not explicitly utilize relational action features and rely only on the summary vector to encode all the necessary action relations.	143
B.9	Analyses of (i) GAT v/s GCN and (ii) state-action graph v/s action-only graphs on (left) RecSim: Direct CPR environment and (right) RecSim: Pairing environment. .	145

B.10	Architectures of all methods used as baselines (Sec. 3.6.1), ablations (Sec. 3.6.1), and analyses (Sec. 3.6.3). (a) Following prior work in SAS-MDPs (Boutillier et al., 2018; Chandak et al., 2020a) and invalid action masking (Huang and Ontañón, 2020; Ye et al., 2020; Kanervisto et al., 2020), the mask-output baseline masks out the unavailable actions assuming a known action set. The mask-input-output baseline additionally augments the state with the action availability mask. (b) Utility-policy (Jain et al., 2020) can generalize over actions by using action representations. However, it computes each action utility independent of other available actions. (c) Summary Ablations augment the utility-policy with an extra action-summary input, which is a compressed version of the list of available action representations. This compression can be done by mean-pooling over a <i>Bi-LSTM</i> output layer, or a <i>deep set</i> , or a graph network (<i>GAT</i>) processed node features. (d) AGILE (Ours) uses a <i>GAT</i> 's node features both to compute an action set summary and as relational action features replacing the original action representations. While the action set summary is a compact representation of the available actions, it does not sufficiently scale when there are many actions or the task requires many action relations for each action decision. (e) AGILE-Only Action is the version of AGILE where the state is not used to compute action relations in <i>GAT</i> . This is a simpler architecture to learn but is not expected to work for certain tasks where action relations change depending on the state.	146
C.1	Benchmark Environments involve discrete action space tasks like Mine World and recommender systems (simulated and MovieLens-Data) and restricted locomotion tasks.	163
C.2	Mining Expedition. The red agent must reach the green goal by navigating the grid and using one or more pick-axes to clear each mine blocking the path.	163
C.3	Unrestricted Locomotion (Section D.1). SAVO and most baselines perform optimally in standard MuJoCo continuous control tasks, where the Q-landscape is easy to navigate.	169
C.4	Ablations of SAVO variations (Section D.2) shows the importance of (i) the approximation of surrogates, (ii) removing TD3's action smoothing, (iii) conditioning on preceding actions in the successive actor and surrogate networks, and (iv) individual actors that separate the action candidate prediction instead of a joint high-dimensional learning task.	169
C.5	SAC is orthogonal to the effect of SAVO (Section D.3). SAC is a different algorithm than TD3, whereas SAVO is a plug-in actor architecture for TD3. Thus, tasks where SAC outperforms TD3 differ from tasks where SAVO outperforms TD3. Also, TD3 outperforms SAC in Restricted Hopper and Inverted-Double-Pendulum. However, SAVO+TD3 guarantees improvement over TD3.	170
C.6	SAC is suboptimal in complex Q-landscape (Section D.3) of Restricted Inverted Double Pendulum, but SAVO helps.	171

C.7	Increasing RecSim action set size (Section D.4). (Left) 100,000 items, (Right) 500,000 items (6 seeds) maintains the performance trends of SAVO and the best-performing baseline (TD3 + Sampling) and the best-performing ablation (SAVO with Joint-Action).	172
C.8	Action summarizer comparison (Section E.1). The effect is not significant. The results are averaged over 5 random seeds, and the seed variance is shown with shading.	172
C.9	FiLM to condition on preceding actions (Section E.2). FiLM ensures layerwise dependence on the preceding actions for acting in actors ν_i and for predicting value in surrogates $\hat{\Psi}_i$, which generally results in better performance across tasks.	173
C.10	OU versus Gaussian Noise (Section E.3). We do not see a significant difference due to this choice, and select OU noise due to better overall performance in experiments.	174
C.11	Successive surrogate landscapes and the Q-landscape of Inverted Pendulum-v4.	179
C.12	Successive surrogate landscapes and Q landscape for Restricted Inverted-Pendulum and Restricted Inverted-Double-Pendulum environments.	180
C.13	Hopper-v4: Q landscape visualization at different states show a path to optimum.	181
C.14	Hopper-restricted: Q landscape visualization at different states show several local optima.	182
D.12	Ten Tasks Defined for the Maze Navigation	191
D.8	Success Rates for Individual Tasks in Multistage Reacher	194
D.9	QMP + PCGrad Results	195
D.10	Task Scaling and Data Sharing Results	195
D.11	Comparison with MCAL and DECAF	196
D.12	Comparison of H-step to 1-step QMP	197
D.14	Mixture Probabilities Per Task	200
D.15	Average Mixture Composition in Multistage Reacher	201
D.16	Probabilistic Mixture and Behavior Length Ablations, SAC Exploration Comparison	202

Abstract

The action space of a reinforcement learning (RL) agent defines how it interacts with the world, whether selecting discrete items in a recommendation system or controlling continuous movements in robotics. Most RL algorithms, however, assume a fixed and well-defined action space, which contrasts with humans' flexibility in making optimal decisions across various dynamic action spaces. Therefore, the goal of my research is to enable decision-making in complex action spaces that are unseen, varying, non-convex, or shareable across multiple tasks.

In this thesis, we see these action space complexities through the lens of Q-functions — a fundamental tool that captures the relationship between each action and its long-term effect on a task. Specifically, we address decision-making in (1) unseen actions, where agents must generalize to new tools or skills, by leveraging action representations and a flexible policy architecture; (2) varying action spaces, such as dynamically changing inventories, by learning the interdependence between available action space; (3) non-convex Q-functions, where locally optimal actions hinder the search for global optima, by successively refining policies and Q-functions; and (4) shareable action spaces in simultaneous multi-task RL, by selectively sharing action proposals across tasks for improved sample efficiency.

The work presented here is a step towards identifying and addressing fundamental complexities within the action spaces of RL agents. We hope it will pave the way toward self-reliant agents capable of autonomously defining and maneuvering their action spaces to solve various decision-making tasks with human-like robustness.

Chapter 1

Introduction

Humans exhibit a remarkable ability to select optimal actions in a variety of decision-making tasks, ranging from simple binary choices like moving left or right to fine-grained control of limb movements. Consider the example of cooking in an unfamiliar kitchen: despite encountering a novel set of tools, we can quickly identify which tool is best suited for each task, and despite the vast range of motions possible with the tool, we can skillfully manipulate it. This robustness, both in selecting discrete actions and executing continuous ones, is enabled by our understanding of each action’s long-term effect on the task and our ability to reason about the most promising actions. How can we build intelligent agents that are similarly capable of acting optimally despite the complexities in their action spaces?

The introduction of deep neural networks in reinforcement learning (RL) has led to powerful agents to play Go ([Silver et al., 2017](#)), Dota 2 ([Berner et al., 2019](#)) and StarCraft II ([Vinyals et al., 2019](#)), and perform robotic grasping ([Kalashnikov et al., 2018](#)). Deep RL research is primarily centered around questions of function approximation of value ([Mnih et al., 2015](#); [Lillicrap et al., 2015](#)), generalization over environments ([Cobbe et al., 2018](#)) and tasks ([Oh et al., 2017](#)), and exploration ([Pathak et al., 2017](#)). However, these tasks and benchmarks all assume a well-defined action space. In contrast, the action spaces humans act in can be dynamic, unseen, and intricately nuanced, which makes finding the optimal action challenging.

Let’s return to the example of humans cooking in an unfamiliar kitchen and identify the core action space challenges for intelligent agents.

1. We often use unknown tools or improvise if a common tool is missing, for example, by using a fork as a whisk when one is unavailable. Thus, to utilize **unseen** actions, agents must have the ability to understand the behavior of the given actions.
2. We adapt our tool-selection strategy depending on the toolset available by considering how the tools might interact. For instance, we can get warm water by simply filling a mug and heating it in a microwave, but in the absence of a microwave, we would instead fill a pot with water and heat it on a stove before finally filling our mug. Thus, an agent must be able to adapt its strategy according to the action space that is **varying**.
3. Once we select a tool (seen or unseen), there are several imperfect ways to manipulate it, like grasping it in various orientations. Yet, we can figure out the best way to achieve the desired result, like finding the optimal angle to chop vegetables efficiently. Similarly, an agent must be able to optimize the best action in the **presence of suboptimal actions**.
4. When learning to manipulate multiple new tools simultaneously, we perform trial-and-error by re-using behaviors across tools that seem similar. Such behavior reuse between tasks can ease searching for the best actions because it might already be learned in another task. Thus, an agent must find the best action more efficiently in multi-task RL than in single-task RL by effectively re-using **shareable** actions between the different tasks.

Thus, the goal of my research is to enable decision-making in action spaces that are unseen, varying, non-convex to optimize, or shareable with other tasks. Such action space complexities are common in various scenarios, including robotics, recommender systems, and physical reasoning. Like humans, we leverage the relationship of each action with its long-term effect on the task (Q-value) and reason about the most promising actions. Concretely, we address each problem by formalizing the relationship of the Q-value over actions, a Q-function, in an architecture that enables us to maximize and find the optimal action easily. In summary, this thesis addresses decision-making in complex action spaces, including (1) generalization to unseen and varying action spaces using

action representations (Part I) and (2) optimizing the action in large or continuous action spaces with several suboptimal actions in single-task and multi-task RL (Part II).

1.1 Generalization to Unseen and Varying Action Spaces

In Part I, we enable decision-making in unseen and varying discrete action choices. Conventional RL benchmarks *assume* a fixed action set, such as Atari games — where actions like moving left, right, or firing are predefined (Mnih et al., 2015). However, the available inventory of an agent can change, such as in tool use, skill composition, and recommendation systems. We posit that **action representations** — continuous vectors to characterize actions — enable generalization over actions.

In Chapter 2, we leverage action metadata, such as videos of action behavior, and train a dataset VAE (Edwards and Storkey, 2017) to encode the representation of actions like tools and skills. We introduce an RL algorithm that uses these representations to compute each action’s long-term utility (analogous to Q-value). We develop the Chain REaction Tool Environment (CREATE) to evaluate generalizable tool selection in physics puzzles. When faced with new actions, the agent can utilize previously learned representations to make informed decisions, facilitating effective zero-shot generalization to unseen actions. This research was published in Jain et al. (2020).

In Chapter 3, we observe that varying action sets necessitates reasoning about the relations between the actions available, akin to how selecting a nail is only useful if a hammer is available. We extend our action generalization framework to incorporate graph attention networks (GATs) (Veličković et al., 2017) to explicitly learn and utilize relationships between the available actions’ representations for decision-making. An agent informed about its action space zero-shot adapts its strategies under different action availabilities, including completely unseen action sets. This research was published in Jain et al. (2021).

1.2 Near-optimal Action in Non-convex Q-functions

In Part II, we improve decision-making in non-convex Q-functions containing multiple local optima, making finding the optimal action challenging. Actor-critic RL methods following [Silver et al. \(2014\)](#) learn a policy that ascends the gradient of the Q-function and thus can get stuck in locally optimal actions. The key insight to finding near-optimal actions is obtaining multiple action candidates and explicitly computing the Q-value’s maximum. We realize this insight in both single-task and multi-task RL.

In Chapter 4, we iteratively prune the Q-function landscape below the current best Q-value, inspired by tabu search ([Glover, 1990](#)), resulting in a refined landscape with fewer local optima. This enables successive gradient ascent to find actions with a higher Q-value. Finally, we combine multiple policies through an $\arg \max$ operation on their Q-values to construct a superior policy, leading to near-optimal decision-making in algorithms like TD3 ([Fujimoto et al., 2018](#)). This work appeared in [Jain et al. \(2024\)](#).

In Chapter 5, we address this non-convexity with candidate actions for $\arg \max$ coming from different task policies in multi-task reinforcement learning (MTRL). We derive the objective that must be maximized in soft actor-critic (SAC) algorithm ([Haarnoja et al., 2018](#)) and prove that the resulting MTRL algorithm achieves better sample efficiency. Consequently, we achieve a new way of sharing information between the tasks in MTRL, sharing actions, which is complementary to prior MTRL paradigms such as parameter sharing and data sharing. This research appeared in [Zhang et al. \(2023\)](#).

Finally, we conclude by discussing open challenges in complex action spaces and some future directions in Chapter 6.

Part I

Generalization to Unseen and Varying Action Spaces

Chapter 2

Generalization to New Actions via Action Representations

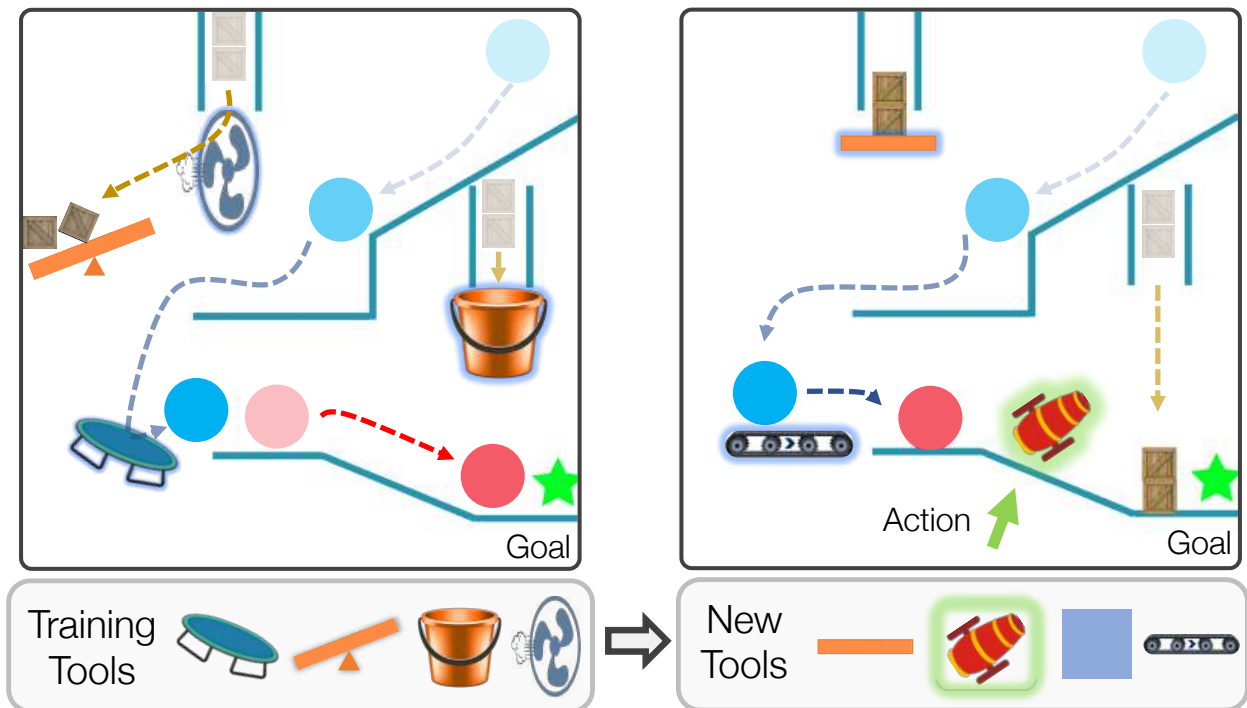


Figure 2.1: An illustration of zero-shot generalization to new actions in a sequential decision-making task, CREATE. (Left) Learning to select and place the right tools for reaching the goal. (Right) Generalizing the learned policy to a previously unseen set of tools.

2.1 Introduction

Imagine making a salad with an unfamiliar set of tools. Since tools are characterized by their behaviors, you would first inspect the tools by interacting with them. For instance, you can observe

a blade has a thin edge and infer that it is sharp. Afterward, when you need to cut vegetables for the salad, you decide to use this blade because you know sharp objects are suitable for cutting. Like this, humans can make selections from a novel set of choices by observing the choices, inferring their properties, and finally making decisions to satisfy the requirements of the task.

From a reinforcement learning perspective, this motivates an important question of how agents can adapt to solve tasks with previously unseen actions. Prior work in deep reinforcement learning has explored generalization of policies over environments (Cobbe et al., 2018; Nichol et al., 2018), tasks (Finn et al., 2017; Parisi et al., 2018), and agent morphologies (Wang et al., 2018; Pathak et al., 2019a). However, zero-shot generalization of policies to new discrete actions has not yet been explored. The primary goal of this work is to propose the problem of generalization to new actions. In this setup, a policy that is trained on one set of discrete actions is evaluated on its ability to solve tasks zero-shot with new actions that were unseen during training.

Addressing this problem can enable robots to solve tasks with a previously unseen toolkit, recommender systems to make suggestions from newly added products, and hierarchical reinforcement learning agents to use a newly acquired skill set. In such applications, retraining with new actions would require prohibitively costly environment interactions. Hence, zero-shot generalization to new actions without retraining is crucial to building robust agents. To this end, we propose a framework and benchmark it on using new tools in the CREATE physics environment (Figure 2.1), stacking of towers with novel 3D shapes, reaching goals with unseen navigation skills, and recommending new articles to users.

We identify three challenges faced when generalizing to new actions. Firstly, an agent must observe or interact with the actions to obtain data about their characteristics. This data can be in the form of videos of a robot interacting with various tools, images of inspecting objects from different viewpoints, or state trajectories observed when executing skills. In present work, we assume such action observations are given as input since acquiring them is domain-specific. The second key challenge is to extract meaningful properties of the actions from the acquired action observations, which are diverse and high-dimensional. Finally, the task-solving policy architecture must be flexible

to incorporate new actions and be trained through a procedure that avoids overfitting (Hawkins, 2004) to training actions.

To address these challenges, we propose a two-stage framework of representing the given actions and using them for a task. First, we employ the hierarchical variational autoencoder (Edwards and Storkey, 2017) to learn action representations by encoding the acquired action observations. In the reinforcement learning stage, our proposed policy architecture computes each given action’s utility using its representation and outputs a distribution. We observe that naive training leads to overfitting to specific actions. Thus, we propose a training procedure that encourages the policy to select diverse actions during training, hence improving its generalization to unseen actions.

Our main contribution is introducing the problem of generalization to new actions. We propose four new environments to benchmark this setting. We show that our proposed two-stage framework can extract meaningful action representations and utilize them to solve tasks by making decisions from new actions. Finally, we examine the robustness of our method and show its benefits over retraining on new actions.

2.2 Related Work

Generalization in Reinforcement Learning. Our proposed problem of zero-shot generalization to new discrete action-spaces follows prior research in deep reinforcement learning (RL) for building robust agents. Previously, state-space generalization has been used to transfer policies to new environments (Cobbe et al., 2018; Nichol et al., 2018; Packer et al., 2018), agent morphologies (Wang et al., 2018; Sanchez-Gonzalez et al., 2018a; Pathak et al., 2019a), and visual inputs for manipulation of unseen tools (Fang et al., 2018; Xie et al., 2019). Similarly, policies can solve new tasks by generalizing over input task-specifications, enabling agents to follow new instructions (Oh et al., 2017), demonstrations (Xu et al., 2017), and sequences of subtasks (Andreas et al., 2017). Likewise, our work enables policies to adapt to previously unseen action choices.

Unsupervised Representation Learning. Representation learning of high-dimensional data can make it easier to extract useful information for downstream tasks (Bengio et al., 2013). Prior work

has explored downstream tasks such as classification and video prediction (Denton and Birodkar, 2017), relational reasoning through visual representation of objects (Steenbrugge et al., 2018), domain adaptation in RL by representing image states (Higgins et al., 2017b), and goal representation in RL for better exploration (Laversanne-Finot et al., 2018) and sample efficiency (Nair et al., 2018b). In this work, we leverage unsupervised representation learning of action observations to achieve generalization to new actions in the downstream RL task.

Learning Action Representations. In prior work, Chen et al. (2019b); Chandak et al. (2019); Kim et al. (2019) learn a latent space of discrete actions during policy training by using forward or inverse models. Tennenholtz and Mannor (2019) use expert demonstration data to extract contextual action representations. However, these approaches require a predetermined and fixed action space. Thus, they cannot be used to infer representations of previously unseen actions. In contrast, we learn action representations by encoding action observations acquired independent of the task, which enables zero-shot generalization to novel actions.

Applications of Action Representations. Continuous representations of discrete actions have been primarily used to ease learning in large discrete action spaces (Dulac-Arnold et al., 2015; Chandak et al., 2019) or exploiting the shared structure among actions for efficient learning and exploration (He et al., 2015; Tennenholtz and Mannor, 2019; Kim et al., 2019). Concurrent work from Chandak et al. (2020b) learns to predict in the space of action representations, allowing efficient finetuning when new actions are added. In contrast, we utilize action representations learned separately, to enable zero-shot generalization to new actions in RL.

2.3 Problem Formulation

In order to build robust decision-making agents, we introduce the problem setting of generalization to new actions. A policy that is trained on one set of actions is evaluated on its ability to utilize unseen actions without additional retraining. Such zero-shot transfer requires additional input that can illustrate the general characteristics of the actions. Our insight is that action choices, such as tools, are characterized by their general behaviors. Therefore, we record a collection of an action’s

behavior in diverse settings in a separate environment to serve as action observations. The action information extracted from these observations can then be used by the downstream task policy to make decisions. For instance, videos of an unseen blade interacting with various objects can be used to infer that the blade is sharp. If the downstream task is cutting, an agent can then reason to select this blade due to its sharpness.

2.3.1 Reinforcement Learning

We consider the problem family of episodic Markov Decision Processes (MDPs) with discrete action spaces. MDPs are defined by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma\}$ of states, actions, transition probability, reward function, and discount factor. At each time step t in an episode, the agent receives a state observation $s_t \in \mathcal{S}$ from the environment and responds with an action $a_t \in \mathcal{A}$. This results in a state transition to s_{t+1} and a state-conditioned reward $\mathcal{R}(s_{t+1})$. The objective of the agent is to maximize the expected discounted reward $R = \sum_{t=1}^T \gamma^{t-1} \mathcal{R}(s_t)$ in an episode of length T .

2.3.2 Generalization to New Actions

The setting of generalization to new actions consists of two phases: training and evaluation. During training, the agent learns to solve tasks with a given set of actions $\mathbb{A} = \{a_1, \dots, a_N\}$. During each evaluation episode, the trained agent is evaluated on a new action set \mathcal{A} sampled from a set of unseen actions \mathbb{A}' . The objective is to learn a policy $\pi(a|s, \mathcal{A})$, which maximizes the expected discounted reward using any given action set $\mathcal{A} \subset \mathbb{A}'$,

$$R = \mathbb{E}_{\mathcal{A} \subset \mathbb{A}', a \sim \pi(a|s, \mathcal{A})} \left[\sum_{t=1}^T \gamma^{t-1} \mathcal{R}(s_t) \right]. \quad (2.1)$$

For each action $a \in \mathbb{A} \cup \mathbb{A}'$, the set of acquired action observations is denoted with $\mathcal{O} = \{o_1, \dots, o_n\}$. Here, each $o_j \in \mathcal{O}$ is an observation for the action like a state-trajectory, a video, or an image, indicating the action's behavior. For the set of training actions \mathbb{A} , we denote the set of associated actions observations as $\mathbb{O} = \{\mathcal{O}_1, \dots, \mathcal{O}_N\}$.

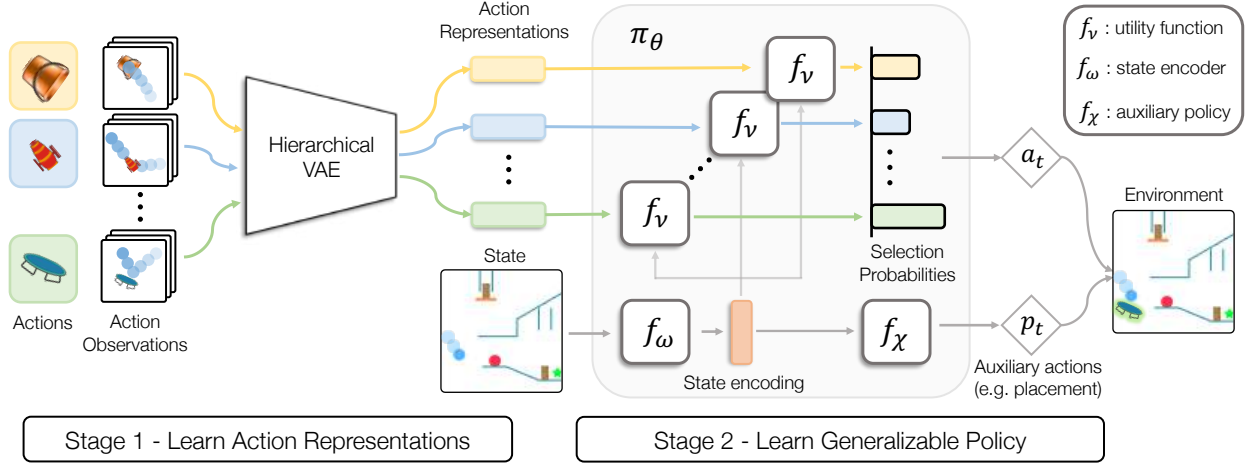


Figure 2.2: Two-stage framework for generalization to new actions through action representations. (1) For each available action, a hierarchical VAE module encodes the action observations into action representations and is trained with a reconstruction objective. (2) The policy π_θ encodes the state with state encoder $f_\omega(s)$ and pairs it with each action representation using the utility function f_v . The utility scores are computed for each action and output to a categorical distribution. The auxiliary network takes the encoded state and outputs environment-specific auxiliary actions such as tool placement in CREATE. The policy architecture is trained with policy gradients.

2.4 Approach

Our approach for generalization to new actions is based on the intuition that humans make decisions from new options by exploiting prior knowledge about the options (Gershman and Niv, 2015). First, we infer the properties of each action from the action observations given as prior knowledge. Second, a policy learns to make decisions based on these inferred action properties. When a new action set is given, their properties are inferred and exploited by the policy to solve the task. Formally, we propose a two-stage framework:

1. **Learning Action Representations:** We use unsupervised representation learning to encode each set of action observations into an action representation. This representation expresses the latent action properties present in the set of diverse observations (Section 2.4.1).
2. **Learning Generalizable Policy:** We propose a flexible policy architecture to incorporate action representations as inputs, which can be trained through RL (Section 2.4.2). We

provide a training procedure to control overfitting to the training action set, making the policy generalize better to unseen actions (Section 2.4.3).

2.4.1 Unsupervised Learning of Action Representations

Our goal is to encode each set of action observations into an action representation that can be used by a policy to make decisions in a task. The main challenge is to extract the shared statistics of the action’s behavior from high-dimensional and diverse observations.

To address this, we employ the hierarchical variational autoencoder (HVAE) by [Edwards and Storkey \(2017\)](#). HVAE first summarizes the entire set of an action’s observations into a single action latent. This action latent then conditions the encoding and reconstruction of each constituent observation through a conditional VAE. Such hierarchical conditioning ensures that the observations for the same action are organized together in the latent space. Furthermore, the action latent sufficiently encodes the diverse statistics of the action. Therefore, this action latent is used as the action’s representation in the downstream RL task (Figure 2.2).

Formally, for each training action $a_i \in \mathbb{A}$, HVAE encodes its associated action observations $\mathcal{O}_i \in \mathbb{O}$ into a representation c_i by mean-pooling over the individual observations $o_{i,j} \in \mathcal{O}_i$. We refer to this action encoder as the action representation module $q_\phi(c_i|\mathcal{O}_i)$. The action latent c_i sampled from the action encoder is used to condition the encoders $q_\psi(z_{i,j}|o_{i,j}, c_i)$ and decoders $p(o_{i,j}|z_{i,j}, c_i)$ for each individual observation $o_{i,j} \in \mathcal{O}_i$. The entire HVAE framework is trained with reconstruction loss across the individual observations, along with KL-divergence regularization of encoders q_ϕ and q_ψ with their respective prior distributions $p(c)$ and $p(z|c)$. For additional details on HVAE, refer to Appendix D.3.1 and [Edwards and Storkey \(2017\)](#). The final training objective requires maximizing the ELBO:

$$\mathcal{L} = \sum_{\mathcal{O} \in \mathbb{O}} \left[\mathbb{E}_{q_\phi(c|\mathcal{O})} \left[\sum_{o \in \mathcal{O}} \mathbb{E}_{q_\psi(z|o,c)} \log p(o|z, c) - D_{KL}(q_\psi||p(z|c)) \right] - D_{KL}(q_\phi||p(c)) \right]. \quad (2.2)$$

Algorithm 1 Two-stage Training Framework

- 1: **Inputs:** Training actions \mathbb{A} , action observations \mathbb{O}
 - 2: Randomly initialize HVAE and policy parameters
 - 3: **for** epoch = 1, 2, ... **do**
 - 4: Sample batch of action observations $\mathcal{O}_i \sim \mathbb{O}$
 - 5: Train HVAE parameters with gradient ascent on Eq. 2.2
 - 6: **end for**
 - 7: Infer action representations: $c_i = q_\phi^\mu(\mathcal{O}_i), \forall a_i \in \mathbb{A}$
 - 8: **for** iteration = 1, 2, ... **do**
 - 9: **while** episode not done **do**
 - 10: Subsample action set $\mathcal{A} \subset \mathbb{A}$ of size m
 - 11: Sample action $a_t \sim \pi_\theta(s, \mathcal{A})$ using Eq. 2.3
 - 12: $s_{t+1}, r_t \leftarrow \text{ENV}(s_t, a_t)$
 - 13: Store experience (s_t, a_t, s_{t+1}, r_t) in replay buffer
 - 14: **end while**
 - 15: Update and save policy θ using PPO on Eq. 2.4
 - 16: **end for**
 - 17: Select θ with best validation performance
-

For action observations consisting of sequential data, $o = \{x_0, \dots, x_m\}$ like state trajectories or videos, we augment HVAE to extract temporally extended behaviors of actions. We accomplish this by incorporating insights from trajectory autoencoders (Wang et al., 2017; Co-Reyes et al., 2018) in HVAE. Bi-LSTM (Schuster and Paliwal, 1997) is used in the encoders and LSTM is used as the decoder $p(x_1, \dots, x_m | z, c, x_0)$ to reconstruct the trajectory given the initial state x_0 . Explicitly for video observations, we also incorporated temporal skip connections (Ebert et al., 2017) by predicting an extra mask channel to balance contributions from the predicted and first video frame of the video. We set the representation for an action as the mean of the inferred distribution $q_\phi(c_i | \mathcal{O}_i)$ as done in Higgins et al. (2017a); Steenbrugge et al. (2018).

2.4.2 Adaptable Policy Architecture

To enable decision-making with new actions, we develop a policy architecture that can adapt to any available action set \mathcal{A} by taking the list of action representations as input. Since the action representations are learned independently of the downstream task, a task-solving policy must learn to extract the relevant task-specific knowledge.

Algorithm 2 Generalization to New Actions

- 1: **Inputs:** New actions $\mathcal{A} = \{a_1, \dots, a_M\}$, observations $\{\mathcal{O}_1, \dots, \mathcal{O}_M\}$. Trained networks q_ϕ and π_θ
 - 2: Infer action representations: $c_i = q_\phi^\mu(\mathcal{O}_i), \forall a_i \in \mathcal{A}$
 - 3: **while** not done **do**
 - 4: Sample action $a_t \sim \pi_\theta(s, \mathcal{A})$ using Eq. 2.3
 - 5: $s_{t+1}, r_t \leftarrow \text{ENV}(s_t, a_t)$
 - 6: **end while**
-

The policy $\pi(a|s, \mathcal{A})$ receives a set of available actions $\mathcal{A} = \{a_1, \dots, a_k\}$ as input, along with the action representations $\{c_1, \dots, c_k\}$. As shown in Figure 2.2, the policy architecture starts with a state encoder f_ω . The utility function f_ν is applied to each given action’s representation c_i and the encoded state $f_\omega(s)$ (Eq. 2.3). The utility function estimates the score of an action at the current state, through its action representation, just like a Q-function (Watkins and Dayan, 1992). Action utility scores are converted into a probability distribution through a softmax function:

$$\pi(a_i|s, \mathcal{A}) = \frac{e^{f_\nu[c_i, f_\omega(s)]}}{\sum_{j=1}^k e^{f_\nu[c_j, f_\omega(s)]}}. \quad (2.3)$$

In many physical environments, the choice of a discrete action is associated with auxiliary parameterizations, such as the intended position of tool usage or a binary variable to determine episode termination. We incorporate such hybrid action spaces (Hausknecht and Stone, 2015), through an auxiliary network f_χ , which takes the encoded state and outputs a distribution over the auxiliary actions¹. An environment action is taken by sampling the auxiliary action from this distribution and the discrete action from Eq. 2.3. The policy parameters $\theta = \{\nu, \omega, \chi\}$ are trained end-to-end using policy gradients (Sutton et al., 2000).

2.4.3 Generalization Objective and Training Procedure

Our final objective is to find policy parameters θ to maximize reward on held-out action sets $\mathcal{A} \subset \mathbb{A}'$ (Eq. 2.1), while being trained on a limited set of actions \mathbb{A} . We study this generalization problem based on statistical learning theory (Vapnik, 1998; 2013) in supervised learning. Particularly,

¹Alternatively, the auxiliary network can take the discrete selection as input as tested in Appendix C.4

generalization of machine learning models is expected when their training inputs are independent and identically distributed (Bousquet et al., 2003). However, in RL, a policy typically acts in the environment to collect its own training data. Thus when a policy overexploits a specific subset of the training actions, this skews the policy training data towards those actions. To avoid this form of overfitting and be robust to diverse new action sets, we propose the following regularizations to approximate i.i.d. training:

- **Subsampled action spaces:** To limit the actions available in each episode of training, we randomly subsample action sets, $\mathcal{A} \subset \mathbb{A}$ of size m , a hyperparameter. This avoids overfitting to any specific actions by forcing the policy to solve the task with diverse action sets.
- **Maximum entropy regularization:** We further diversify the policy’s actions during training using the maximum entropy objective (Ziebart et al., 2008). We add the entropy of the policy $\mathcal{H}[\pi_\theta(a|s)]$ to the RL objective with a hyperparameter weighting β . While this objective has been widely used for exploration, we find it useful to enable generalization to new actions.
- **Validation-based model selection:** During training, the models are evaluated on held-out validation sets of actions, and the best performing model is selected. Just like supervised learning, this helps to avoid overfitting the policy during training. Note that the validation set is also used to tune hyperparameters such as entropy coefficient β and subsampled action set size m . There is no overlap between test and validation sets, hence the test actions are still completely unseen at evaluation.

The final policy training objective is:

$$\max_{\theta} \mathbb{E}_{\mathcal{A} \subset \mathbb{A}, a \sim \pi_\theta(\cdot|s, \mathcal{A})} [R(s) + \beta \mathcal{H}[\pi_\theta(a|s, \mathcal{A})]]. \quad (2.4)$$

The training procedure is described in Algorithm 1. The HVAE is trained using RAdam optimizer (Liu et al., 2019), and we use PPO (Schulman et al., 2017a) to train the policy with Adam Optimizer (Kingma and Ba, 2014). Additional implementation and experimental details, including the hyperparameters searched, are provided in Appendix D. The inference process is described in

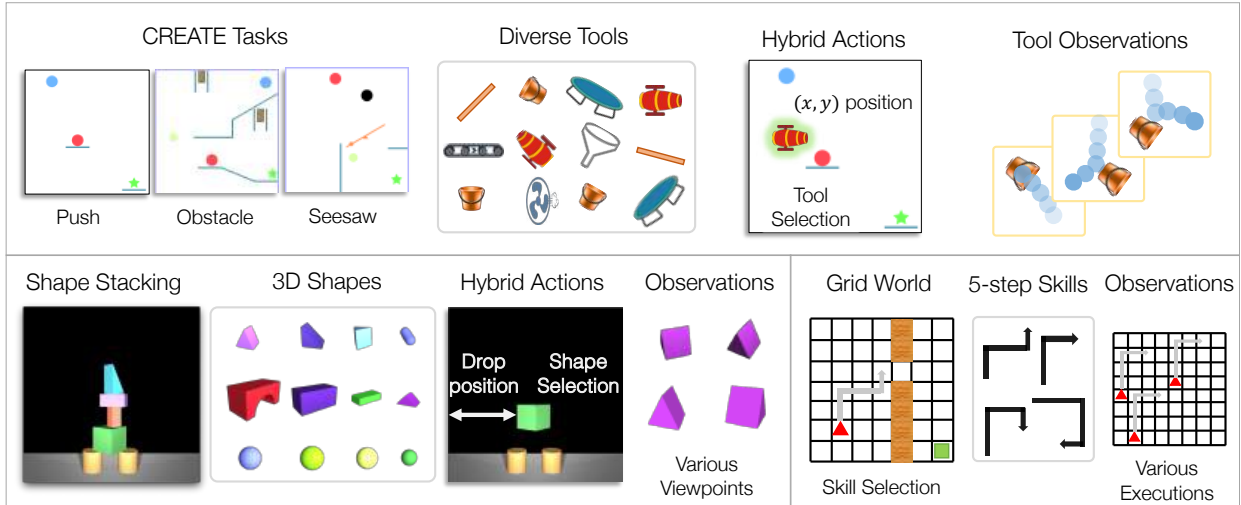


Figure 2.3: Benchmark environments for evaluating generalization to new actions. (Top) In CREATE, an agent selects and places various tools to move the red ball to the goal. Other moving objects can serve as help or obstacles. Some tasks also have subgoals to help with exploration (Appendix C.3 shows results with no subgoal rewards). The tool observations consist of trajectories of a test ball interacting with the tool. (Left) In Shape Stacking, an agent selects and places 3D shapes to stack a tower. The shape observations are images of the shape from different viewpoints. (Right) In Grid World, an agent reaches the goal by choosing from 5-step navigation skills. The skill observations are collected on an empty grid in the form of agent trajectories resulting from skill execution from random locations.

Algorithm 2. When given a new set of actions, we can infer the action representations with the trained HVAE module. The policy can also generalize to utilize these actions since it has learned to map a list of action representations to an action probability distribution.

2.5 Experimental Setup

2.5.1 Environments

We propose four sequential decision-making environments with diverse actions to evaluate and benchmark the proposed problem of generalization to new actions. These test the action representation learning method on various types of action observations. The long-horizon nature of the environments presents a challenge to use new actions correctly to solve the given tasks consistently. Figure 2.3 provides an overview of the task, types of actions, and action observations

in three environments. In each environment, the train-test-validation split is approximately 50-25-25%. Complete details on each environment, action observations, and train-validation-test splits can be found in Appendix A.

Grid World. In the Grid world environment (Chevalier-Boisvert et al., 2018), an agent navigates a 2D lava maze to reach a goal using predefined skills. Each skill is composed of a 5-length sequence of left, right, up or down movement. The total number of available skills is 4^5 . Action observations consist of state sequences of an agent observed by applying the skill in an empty grid. This environment acts as a simple demonstration of generalization to unseen skill sets.

Recommender System. The Recommender System environment (Rohde et al., 2018) simulates users responding to product recommendations. Every episode, the agent makes a series of recommendations for a new user to maximize their click-through rate (CTR). With a total of 10,000 products as actions, the agent is evaluated on how well it can recommend previously unseen products to users. The environment specifies predefined action representations. Thus we only evaluate our policy framework on it, not the action encoder.

CREATE. We develop the Chain REAction Tool Environment (CREATE) as a challenging benchmark to test generalization to new actions². It is a physics-based puzzle where the agent must place tools in real-time to manipulate a specified ball’s trajectory to reach a goal position (Figure 2.3). The environment features 12 different tasks and 2,111 distinct tools. Moreover, it tests physical reasoning since every action involves selecting a tool and predicting the 2D placement for it, making it a hybrid action-space environment. Action observations for a tool consist of a test ball’s trajectories interacting with the tool from various directions and speeds. CREATE tasks evaluate the ability to understand complex functionalities of unseen tools and utilize them for various tasks. We benchmark our framework on all 12 CREATE tasks with the extended results in Appendix C.1.

Shape Stacking. We develop a MuJoCo-based (Todorov et al., 2012) Shape Stacking environment, where the agent drops blocks of different shapes to build a tall and stable tower. Like in CREATE, the discrete selection of shape is parameterized by the coordinates of where to place the

²CREATE environment: <https://clvrai.com/create>

selected shape and a binary action to decide whether to stop stacking. This environment evaluates the ability to use unseen complex 3D shapes in a long horizon task and contains 810 shapes.

2.5.2 Experiment Procedure

We perform the following procedure for each action generalization experiment³.

1. *Collect action observations* for all the actions using a supplemental play environment that is task-independent.
2. *Split the actions* into train, validation, and test sets.
3. *Train HVAE* on the train action set by autoencoding the collected action observations.
4. *Infer action representations* for all actions using the HVAE encoder on action observations.
5. *Train policy* on the task environment with RL. In each episode, an action set is randomly sampled from the train actions. The policy acts by using inferred action representations as an input list.
6. **Evaluation:** In each episode, an action set is subsampled from the test (or validation) action set. The trained policy uses the inferred representations of these actions to act in the environment zero-shot. The performance metric (e.g. success rate) is averaged over multiple such episodes.
 - (a) Perform hyperparameter tuning & model selection by evaluating on *validation action set*.
 - (b) Report final performance on the *test action set*.

2.5.3 Baselines

We validate the design choices of the proposed action encoder and policy architecture. For action encoder, we compare with a policy using action representations from a non-hierarchical

³Complete code available at <https://github.com/clvrail/new-actions-rl>

encoder. For policy architecture, we consider alternatives that select actions using distances in the action representation space instead of learning a utility function.

- **Non-hierarchical VAE:** A flat VAE is trained over the individual action observations. An action’s representation is taken as the mean of encodings of the constituent action observations.
- **Continuous-output:** The policy architecture outputs a continuous vector in the action representation space, following [Dulac-Arnold et al. \(2015\)](#). From any given action set, the action closest to this output is selected.
- **Nearest-Neighbor:** A standard discrete action policy is trained. The representation of this policy’s output action is used to select the nearest neighbor from new actions.

2.5.4 Ablations

We individually ablate the two proposed regularizations:

- **Ours without subsampling:** Train over the entire set of training actions without subsampling.
- **Ours without entropy:** Train without entropy regularization by setting the coefficient to zero.

2.6 Results and Analysis

Our experiments aim to answer the following questions about the proposed problem and framework: (1) Can the HVAE extract meaningful action characteristics from the action observations? (2) What are the contributions of the proposed action encoder, policy architecture, and regularizations for generalization to new actions? (3) How well does our framework generalize to varying difficulties of test actions and types of action observations? (4) How inefficient is finetuning to a new action space as compared to zero-shot generalization?

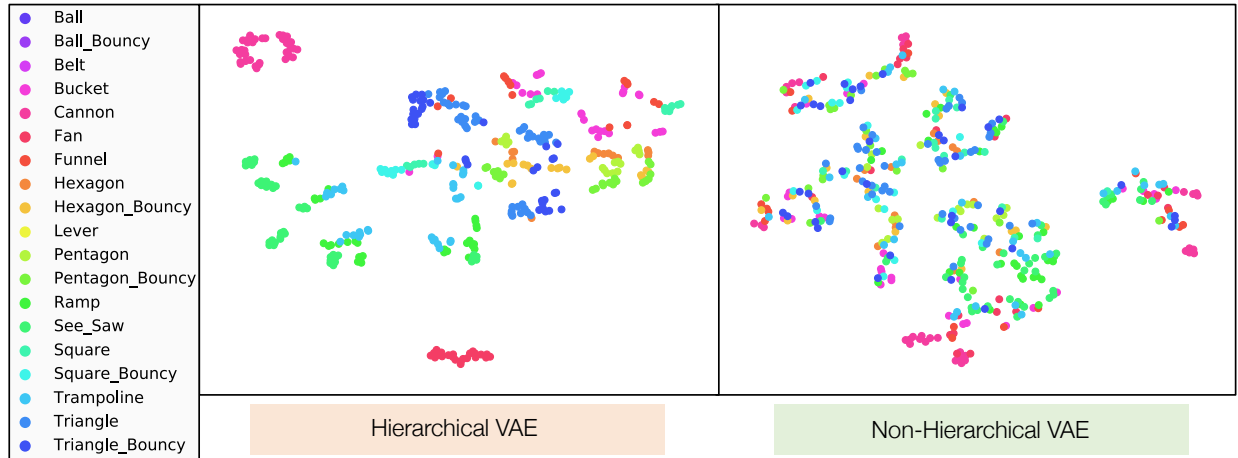


Figure 2.4: t-SNE visualization of action representations for held-out tools in CREATE inferred using a trained HVAE (left) and a VAE (right). The color indicates the tool class (e.g. cannons, buckets). The HVAE encoder learns to organize semantically similar tools together, in contrast to the flat VAE, which shows less structure.

2.6.1 Visualization of Inferred Action Representations

To investigate if the HVAE can extract important characteristics from observations of new actions, we visualize the inferred action representations for unseen CREATE tools. In Figure 2.4, we observe that tools from the same class are clustered together in the HVAE representations. Whereas in the absence of hierarchy, the action representations are less organized. This shows that encoding action observations independently, and averaging them to obtain a representation can result in the loss of semantic information, such as the tool’s class. In contrast, hierarchical conditioning on action representation enforces various constituent observations to be encoded together. This helps to model the diverse statistics of the action’s observations into its representation.

2.6.2 Results and Comparisons

Baselines. Figure 2.5 shows that our framework outperforms the baselines (Section 2.5.3) in zero-shot generalization to new actions on six tasks. The non-hierarchical VAE baseline has lower policy performance in both training and testing. This shows that HVAE extracts better representations from action observations that facilitate easier policy learning.

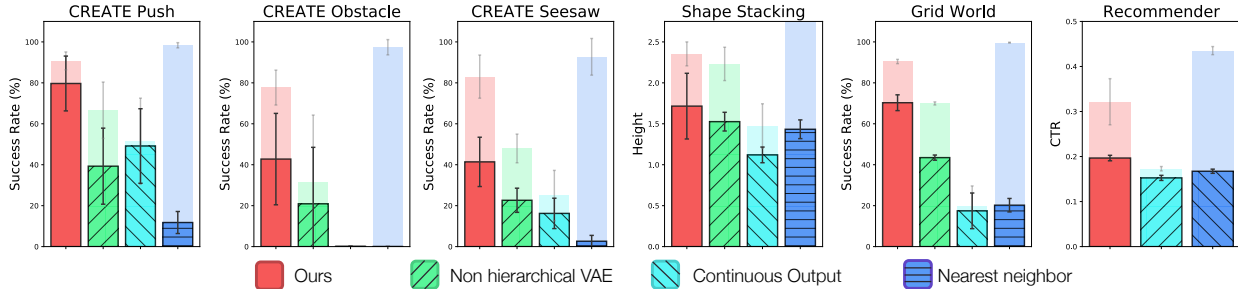


Figure 2.5: Comparison against baseline action representation and policy architectures on 6 environments, 3 of which are CREATE tasks. The solid bar denotes the test performance and the transparent bar the training performance, to observe the generalization gap. The results are averaged over 5000 episodes across 5 random seeds, and the error bars indicate the standard deviation (8 seeds for Grid World). All learning curves are present in Figure A.14. Results on 9 additional CREATE tasks can be found in Appendix C.1.

The continuous-output baseline suffers in training as well as testing performance. This is likely due to the complex task of indirect action selection. The distance metric used to find the closest action does not directly correspond to the task relevance. Therefore the policy network must learn to adjust its continuous output, such that the desired discrete action ends up closest to it. Our method alleviates this through the utility function, which first extracts task-relevant features to enable an appropriate action decision. The nearest-neighbor baseline achieves high training performance since it is merely discrete-action RL with a fixed action set. However, at test time, the simple nearest-neighbor in action representation space does not correspond to the actions’ task-relevance. This results in poor generalization performance.

Ablations. Figure 2.6 assesses the contribution of the proposed regularizations to avoid overfitting to training actions. Entropy regularization usually leads to better training and test performance due to better exploration. In the recommender environment, the generalization gap is more substantial without entropy regularization. Without any incentive to diversify, the policy achieved high training performance by overfitting to certain products. We observe a similar effect in the absence of action subsampling across all tasks. It achieves a higher training performance, due to the ease of training in non-varying action space. However, its generalization performance is weak because it is easy to overfit when the policy has access to all the actions during training.

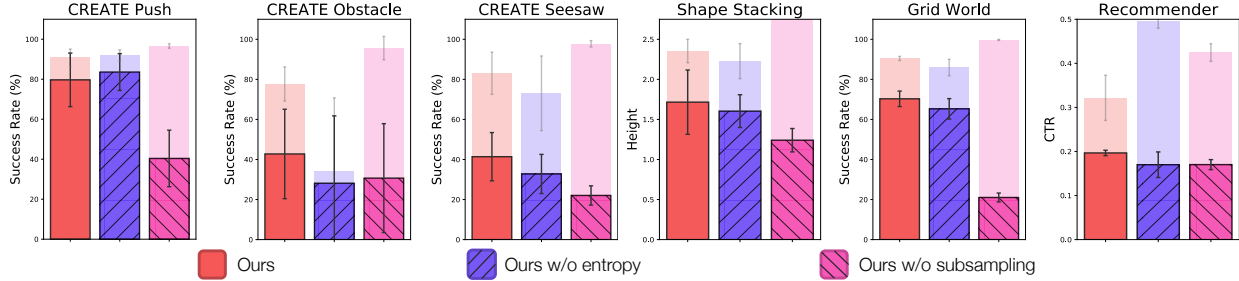


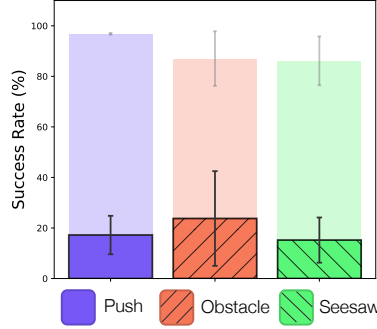
Figure 2.6: Analyzing the importance of the proposed action space subsampling and entropy regularization in our method. Training and evaluation follow Figure 2.5.

2.6.3 Analyzing the Limits of Generalization

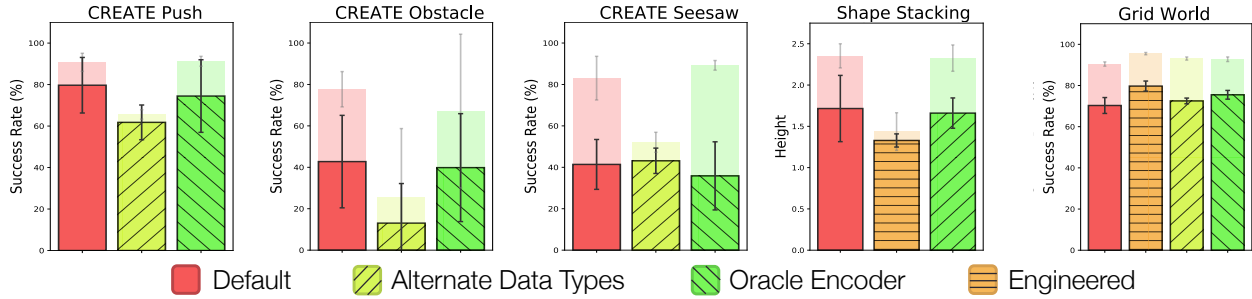
Generalization to Unseen Action Classes. Our method is expected to generalize when new actions are within the distribution of those seen during training. However, what happens when we test our approach on completely unseen action classes? Generalization is still expected because the characteristic action observations enable the representation of actions in the same space. Figure 2.7a evaluates our approach on held-out tool classes in the CREATE environment. Some tool classes like trampolines and cannons are only seen during training, whereas others like fans and conveyor belts are only used during testing. While the generalization gap is more substantial than before, we still observe reasonable task success across the 3 CREATE tasks. The performance can be further improved by increasing the size and diversity of training actions. Appendix C.6 shows a similar experiment on Shape Stacking.

Alternate Action Representations. In Figure 2.7b, we study policy performance for various action representations. See Appendix B for t-SNE visualizations.

- **Alternate Data Types** of action observations are used to learn representations. For CREATE, we use video data instead of the state trajectory of the test ball (see Figure 2.3). For Grid World, we test with a one-hot vector of agent location instead of (x, y) coordinates. The policy performance using these representations is comparable to the default. This shows that HVAE is suitable for high-dimensional action observations, such as videos.



(a) Unseen tool classes in CREATE



(b) Alternate action representations

Figure 2.7: Additional analyses. (a) Our method achieves decent performance on out-of-distribution tools in 3 CREATE tasks, but the generalization gap is more pronounced. (b) Various action representations can be successfully used with our policy architecture.

- **Oracle** HVAE is used to get representations by training on the test actions. The performance difference between default and oracle HVAE is negligible. This shows that HVAE generalizes well to unseen action observations.
- **Hand-Engineered** action representations are obtained for Stacking and Grid World using ground-truth information about the actions. In Stacking, HVAE outperforms these representations, since it is hard to specify the information about shape geometry manually. In contrast, it is easy to specify the complete skill in Grid World. Nevertheless, HVAE representations perform comparably.

Varying the Difficulty of Generalization. Figure 2.8 shows a detailed study of generalization on various degrees of differences between the train and test actions in 3 CREATE tasks. We vary the following parameters:

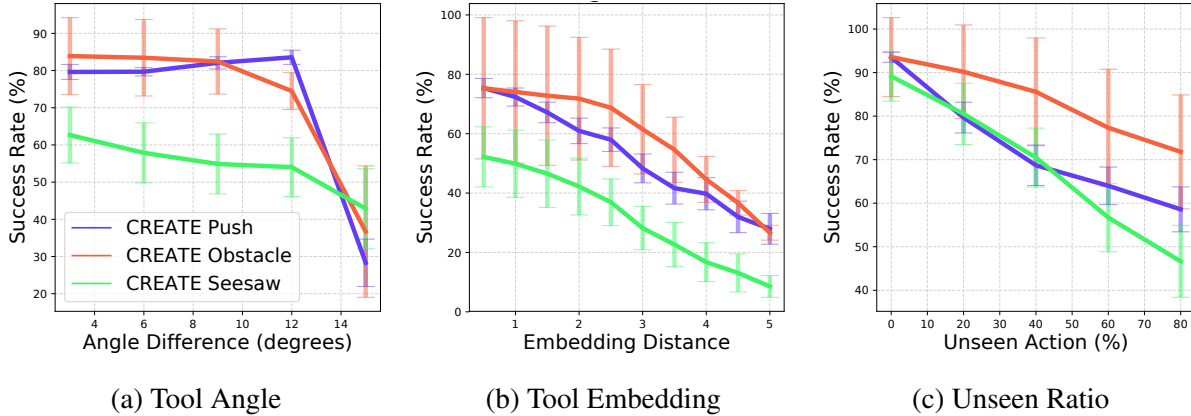


Figure 2.8: Varying the test action space. An increasing x-axis corresponds to more difficult generalization conditions. Each value plotted is the average test performance over 5 random seeds with the error bar corresponding to the standard deviation.

- a) **Tool Angle**: Each sampled test tool is at least θ degrees different from the most similar tool seen during training.
- b) **Tool Embedding**: Each test tool’s representation is at least d Euclidean distance away from each training tool.
- c) **Unseen Ratio**: The test action set is a mixture of seen and unseen tools, with $x\%$ unseen.

The results suggest a gradual decrease in generalization performance as the test actions become more different from training actions. We chose the hardest settings for the main experiments: 15° angle difference and 100% unseen actions.

Qualitative Analysis. Figure 2.9 shows success and failure examples when using unseen actions in the CREATE and Stacking environments. In CREATE, our framework correctly infers the directional pushing properties of unseen tools like conveyor belts and fans from their action observations and can utilize them to solve the task. Failure examples include placements being off and misrepresenting the direction of a belt. Collecting more action observations can improve the representations.

In Shape Stacking, the geometric properties of 3D shapes are correctly inferred from image action observations. The policy can act in the environment by selecting the appropriate shapes to

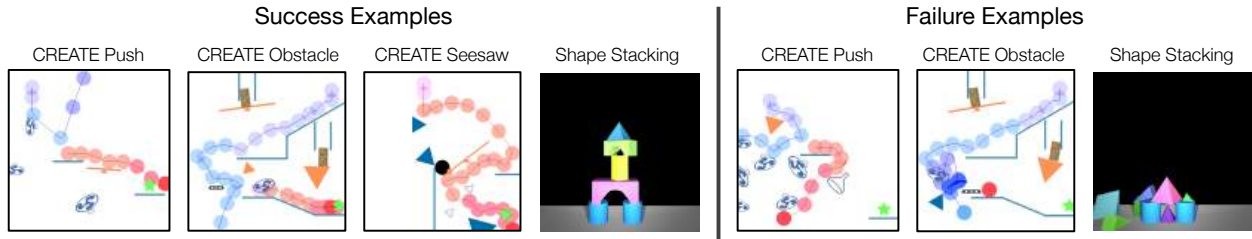


Figure 2.9: Evaluation results showing the trajectories of objects in CREATE and the final tower in Shape Stacking. Our framework is generally able to infer the dynamic properties of tools and geometry of shapes and subsequently use them to make the right decisions.

drop based on the current tower height. Failures include greedily selecting a tall but unstable shape in the beginning, like a pyramid.

2.6.4 The Inefficiency of Finetuning on New Actions

In Figure 2.10, we examine various approaches to continue training on a particular set of new actions in CREATE Push. First, we train a policy from scratch on the new actions either with our adaptable policy architecture (Ours Scratch) or a regular discrete policy (Discrete Scratch). These take around 3 million environment steps to achieve our pretrained method’s zero-shot performance (Ours Zero-Shot). Next, we consider ways to transfer knowledge from training actions. We train a regular discrete policy and finetune on new actions by re-initializing the final layer (Discrete Fine-Tune). While this approach transfers some task knowledge, it disregards any relationship between the old and new actions. It still takes over 1 million steps to reach our zero-shot performance. This shows how expensive retraining is on a single action set. Clearly, this retraining process is prohibitive in scenarios where the action space frequently changes. This demonstrates the significance of addressing the problem of zero-shot generalization to new actions. Finally, we continue training our pretrained policy on the new action set with RL (Ours Fine-Tune). We observe fast convergence to optimal performance, because of its ability to utilize action representations to transfer knowledge from the training actions to the new actions. Finetuning results for all other environments are in Figure A.9.

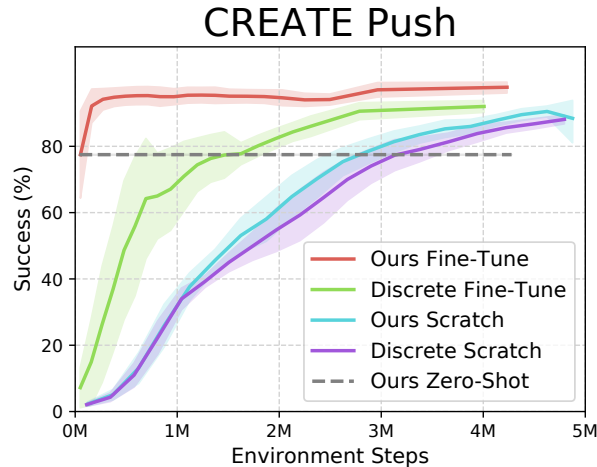


Figure 2.10: Finetuning or training policies from scratch on the new action space. The horizontal line is the zero-shot performance of our method. Each line is the average test performance over 5 random seeds, while the shaded region is the standard deviation.

2.7 Conclusion

Generalization to novel circumstances is vital for robust agents. In this chapter, we propose the problem of enabling RL policies to generalize to new action spaces. Our two-stage framework learns action representations from acquired action observations and utilizes them to make the downstream RL policy flexible. We propose four challenging benchmark environments and demonstrate the efficacy of hierarchical representation learning, policy architecture, and regularizations. Exciting directions for future research include building general problem-solving agents that can adapt to new tasks with new action spaces and autonomously acquiring informative action observations in the physical world.

Chapter 3

Know Your Action Set in Varying Action Spaces via Action Relations

Relations

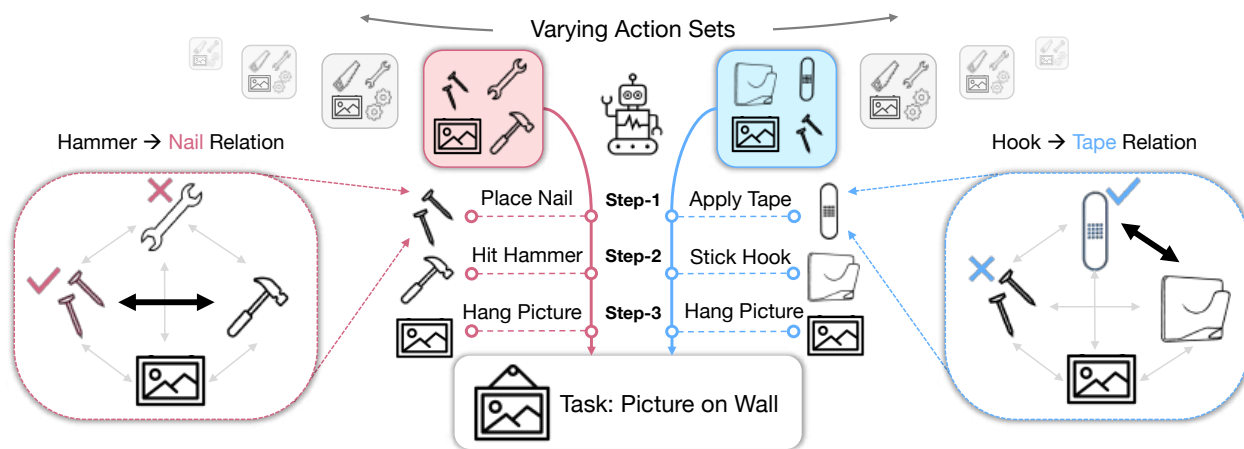


Figure 3.1: Picture hanging task with varying sets of tool-actions. The strategy with each action set depends on all the pairwise action relations. (Left) The agent infers that a nail and a hammer are strongly related (bold line). Thus, it takes nail-action in Step-1 because hammer-action is available for Step-2. (Right) With a different action set, the nail-action is no longer useful due to the absence of a hammer. So, the agent must use an adhesive-tape in Step-1 since its related action of the hook is available for later use. We show that a policy with GAT over actions can learn such action relations.

3.1 Introduction

Imagine you want to hang a picture on the wall. You may start by placing a nail on the wall and then use a hammer to nail it in. However, if you do not have access to a hammer, you would not use

the nail. Instead, you would try alternative approaches such as using a hook and adhesive-strips or a screw and a drill (Figure 3.1). In general, we solve tasks by choosing actions that interact with each other to achieve a desired outcome in the environment. Therefore, the best action decision depends not only on the environment but also on what other actions are available to use.

This work addresses the setting of varying action space in sequential decision-making. Typically reinforcement learning (RL) assumes a fixed action space, but recent work has explored variations in action space when adding or removing actions (Boutillier et al., 2018; Chandak et al., 2020a;c) or for generalization to unseen actions (Jain et al., 2020). These assume that the given actions can be treated independently in decision-making. But this assumption often does not hold. As the picture hanging example illustrates, the optimality of choosing the nail is dependent on the availability of a hammer. Therefore, our goal is to address this problem of learning the interdependence of actions.

Addressing this problem is vital for many varying action space applications, such as recommender systems where articles to recommend vary everyday and physical reasoning where decision-making must be robust to any given set of tools, objects, or skills. In this work, we benchmark three such scenarios where learning action interdependence is crucial for solving the task optimally: (i) shortcut-actions in grid navigation, which can shorten the path to the goal when available, (ii) co-dependent actions in tool reasoning, where tools need other tools to be useful (like nail-hammer), and (iii) list-actions or slate-actions (Sunebag et al., 2015) in simulated and real-data recommender systems, where user response is a collective effect of the recommended list.

There are three key challenges in learning action interdependence for RL with varying action space. First, since all the actions are not known in advance, the policy framework must be flexible to work with action representations. Second, the given action space is an additional variable, like state observations in RL. Therefore, the policy framework must incorporate a variably sized set of action representations as part of the input. Finally, an agent’s decision for each action’s utility (Q-value or probability) should explicitly model its relationship with other available actions.

We propose a novel policy architecture to address these challenges: AGILE, Action Graph for Interdependence Learning. AGILE builds on the utility network proposed in Jain et al. (2020) to

incorporate action representations. Its key component is a graph attention network (GAT) (Veličković et al., 2017) over a fully connected graph of actions. This serves two objectives: summarizing the action set input and computing the action utility with relational information of other actions.

Our primary contribution is introducing the problem of learning action interdependence for RL with varying action space. We demonstrate our proposed policy architecture, AGILE, learns meaningful action relations. This enables optimal decision-making in varying action space tasks such as simulated and real-world recommender systems and reasoning with skills and tools.

3.2 Related Work

Stochastic Action Sets. In prior work, Boutilier et al. (2018) provide the theoretical foundations of MDPs with Stochastic Action Sets (SAS-MDPs), where actions are sampled from a known base set of actions. They propose a solution with Q-learning which Chandak et al. (2020a) extend to policy gradients. An instance of SAS-MDPs is when certain actions become invalid, like in games, they are masked out from the output action probability distribution (Huang and Ontañón, 2020; Ye et al., 2020; Kanervisto et al., 2020). However, the assumption of knowing the finite base action set limits the practical applicability of SAS-MDPs. E.g., recommender agents often receive unseen items to recommend and a robotic agent does not know beforehand what tools it might encounter in the future. We work with action representations to alleviate this limitation. Furthermore, in SAS-MDPs the action set can vary at any timestep of an episode. Thus, the learned policy will only be optimal on average over all the possible action sets (Qin et al., 2020). Whereas in our setting, the action set only changes at the beginning of a task instance and stays constant over the episode. This is a more practical setup and raises the challenge of solving a task optimally with any given action space.

Action Representations. In discrete action RL, action representations have enabled learning in large action spaces (Dulac-Arnold et al., 2015; Chandak et al., 2019), transfer learning to a different action space (Chen et al., 2019b), and efficient exploration by exploiting the shared structure among actions (He et al., 2015; Tennenholtz and Mannor, 2019; Kim et al., 2019). Recently, Chandak

et al. (2020c) use them to accelerate adaptation when new actions are added to an existing action set. In contrast, our setting requires learning in a constantly varying action space where actions can be added, removed, or completely replaced in an episode. Closely related to our work, Jain et al. (2020) assume a similar setting of varying action space while training to generalize to unseen actions. Following their motivation, we use action representations to avoid assuming knowledge of the base action set. However, their policy treats each action independently, which we demonstrate leads to suboptimal performance.

List-wise Action Space. The action space is combinatorial in list size in listwise RL (or slate RL). Commonly it has applications in recommendation systems (Sunehag et al., 2015; Zhao et al., 2017; 2018; Ie et al., 2019b; Gong et al., 2019; Liu et al., 2021b; Jiang et al., 2018; Song et al., 2020). In recent work, Chen et al. (2019a) proposed Cascaded DQNs (CDQN) framework, which learns a Q-network for every index in the list and trains them all with a shared reward. For our experiments on listwise RL, we utilize CDQN as the algorithm and show its application with AGILE as the policy architecture.

Relational Reinforcement Learning. Graph neural networks (Battaglia et al., 2018) have been explored in RL tasks with a rich relational structure, such as morphological control (Wang et al., 2018; Sanchez-Gonzalez et al., 2018b; Pathak et al., 2019b), multi-agent RL (Tacchetti et al., 2019), physical construction (Hamrick et al., 2018), and structured perception in games like StarCraft (Zambaldi et al., 2018). In this paper, we propose that the set of actions possess a relational structure that enables the actions to interact and solve tasks in the environment. Therefore, we leverage a graph attention network (Veličković et al., 2017) to learn these action relations and show that it can model meaningful action interactions.

3.3 Problem Formulation

A hallmark of intelligence is the ability to be robust in an ever-changing environment. To this end, we consider the setting of RL with a varying action space, where an agent receives a different action set in every task instance. Our key problem is to learn the interdependence of actions so the

agent can act optimally with any given action set. Figure 3.1 illustrates that for a robot with the task of hanging a picture on the wall, starting with a nail is optimal only if it can access a hammer subsequently.

3.3.1 Reinforcement Learning with Varying Action Space

We consider episodic Markov Decision Processes (MDPs) with discrete action spaces, supplemented with action representations. The MDP is defined by a tuple $\{\mathcal{S}, \mathbb{A}, \mathcal{T}, \mathcal{R}, \gamma\}$ of states, actions, transition probability, reward function, and a discount factor, respectively. The base set of actions \mathbb{A} can be countably infinite. To support infinite base actions, we use D -dimensional action representations $c_a \in \mathbb{R}^D$ to denote an action $a \in \mathbb{A}$. These can be image or text features of a recommendable item, behavior characteristics of a tool, or simply one-hot vectors for a known and finite action set.

In each instance of the MDP, a subset of actions $\mathcal{A} \subset \mathbb{A}$ is given to the agent, with associated representations \mathcal{C} . Thereafter, at each time step t in the episode, the agent receives a state observation $s_t \in \mathcal{S}$ from the environment and acts with $a_t \in \mathcal{A}$. This results in a state transition to s_{t+1} and a reward r_t . The objective of the agent is to learn a policy $\pi(a|s, \mathcal{A})$ that maximizes the expected discounted reward over evaluation episodes with potentially unseen actions, $\mathbb{E}_{\mathcal{A} \subset \mathbb{A}} [\sum_t \gamma^{t-1} r_t]$.

3.3.2 Challenges of Varying Action Space

1. **Using Action Representations:** The policy framework should be flexible to take a set of action representations \mathcal{C} as input and output corresponding Q-values or probability distribution for RL.
2. **Action Set as part of State:** When the action set varies, the original state space \mathcal{S} is not anymore Markovian. For example, the state of a robot hanging the picture is under-specified without knowing if its toolbox contains a hammer or not. The MDP can be preserved by reformulating the state space as $S' = \{s \circ \mathcal{C}_{\mathcal{A}} : s \in \mathcal{S}, \mathcal{A} \subset \mathbb{A}\}$ to include the representations

$\mathcal{C}_{\mathcal{A}}$ associated with the available actions \mathcal{A} (Boutillier et al., 2018). Thus, the policy framework must support the input of $s \circ \mathcal{C}_{\mathcal{A}}$, where \mathcal{A} is a variably sized action set.

3. **Interdependence of Actions:** The optimal choice of an action $a_t \in \mathcal{A}$ is dependent on the action choices that would be available in the future steps of the episode, $a_{t'} \in \mathcal{A}$. Recalling Figure 3.1, a nail should only be picked initially from the toolbox if a hammer is accessible later. Thus, an optimal agent must explicitly model the relationships between the characteristics of the current action c_{a_t} and the possible future actions $c_{a_i} \forall a_i \in \mathcal{A}$.

3.4 Approach

Our goal is to design a policy framework that is optimal for any given action set by addressing the challenges in Sec. 3.3.2. We build on the utility network proposed by Jain et al. (2020) that acts in parallel on each action’s representation. Our central insight is to use graph neural networks for summarizing the set of action representations as a state component and learning action relations.

3.4.1 AGILE: Action Graph for Interdependence Learning

We propose AGILE, a relational framework for learning action interdependence in RL with a varying action space. Given a list of action representations \mathcal{C} , AGILE builds a fully-connected action graph. A graph attention network (GAT) (Veličković et al., 2017) processes the action graph and learns action relations. The attention weights in the GAT would be high for closely related actions such as a nail and a hammer in Figure 3.1. A utility network uses the GAT’s resulting relational action representations, the state, and a pooled together action set summary to compute each action’s Q-value or probability logit for policy gradient methods (Figure 3.2).

Action Graph: The input to our policy framework consists of the state s and a list $\mathcal{C} = [c_{a_0}, \dots, c_{a_k}]$ of action representations for each action $a_i \in \mathcal{A}$. We build a fully connected action graph \mathcal{G} with vertices corresponding to each available action. If certain action relations are predefined via domain knowledge, we can reduce some edges to ease training (Appendix B.1). We note that

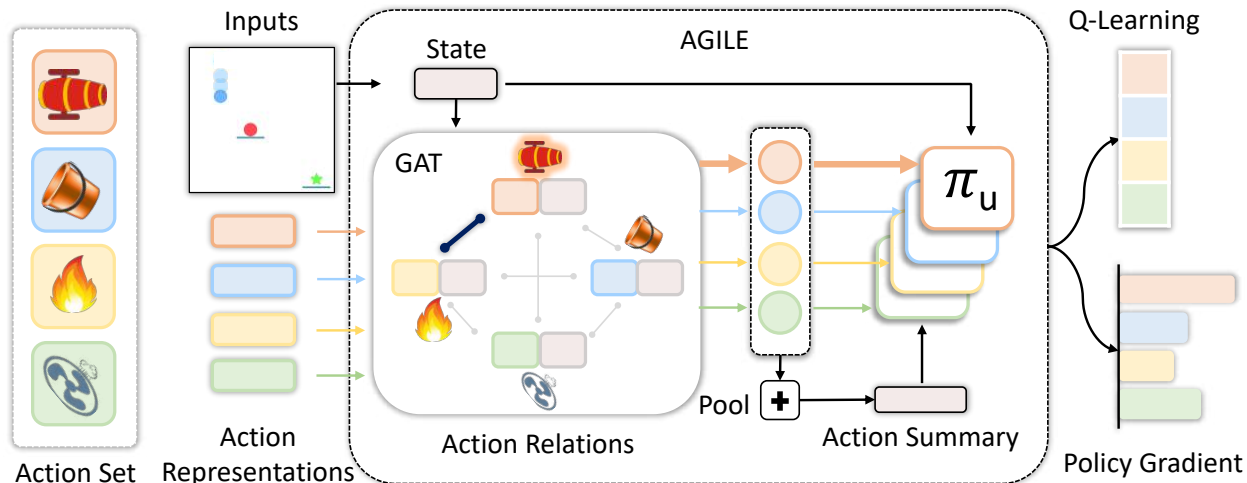


Figure 3.2: Given an action set, AGILE builds a complete graph where each node is composed of an action representation and the state encoding. A graph attention network (GAT) learns action relations by attending to other relevant actions for the current state. For example, the attention weight between cannon and fire is high because fire can activate the cannon. The GAT outputs more informed relational action representations than the original inputs. Finally, a utility network computes each action’s value or selection probability in parallel using its relational action representation, the state, and a mean-pooled summary vector of all available actions’ relational features.

the action relations can vary depending on the environment state. For instance, a screwdriver is related to a screw for furniture repair, but a drill machine is more related when the screw is for a wall. Therefore, we join the state and action representations, $c'_{a_i} = (s, c_{a_i})$ to obtain the nodes of the graph. Sec. 3.6.3 validates that learning state-dependent action relations leads to more optimal solutions.

Graph Attention Network: The action graph \mathcal{G} is input to a GAT. Since the graph is fully-connected, we choose an attention-based graph network that can learn to focus on the most relevant actions in the available action set. A similar insight was employed by [Zambaldi et al. \(2018\)](#) where the entities inferred from the visual observation are assumed to form a fully connected graph. To enable propagation of sufficient relational information between actions, we use two graph attention layers with an ELU activation in between ([Clevert et al., 2015](#)). We found a residual connection after the second GAT layer was crucial in experiments, while multi-headed attention did not help.

Action Set Summary: The output of the GAT is a list of relational action representations $\mathcal{C}^R = \{c_{a_0}^R, \dots, c_{a_k}^R\}$ which contain the information about the presence of other available actions and their relations. To represent the input action set as part of the state, we compute a compact action set summary by mean-pooling the relational action features, $\bar{c}^R = \frac{1}{K} \sum_{i=1}^K c_{a_i}^R$.

Action Utility: To use the relational action representations with RL, we follow the utility network architecture π_u from Jain et al. (2020). It takes the relational action representation, the state, and the action set summary as input for each available action in parallel. It outputs a utility score $\pi_u(c_a^R, s, \bar{c}^R)$ for how useful an action a is for the current state and in relation to the other available actions. The utility scores can be used as a Q-value directly for value-based RL or as a logit fed into a softmax function to form a probability distribution over the available actions for policy-based RL.

3.4.2 Training AGILE framework with Reinforcement Learning

The AGILE architecture can be trained with both policy gradient and value-based RL methods.

Policy Gradient (PPO) : For every action decision step, we take the output action utility and use a softmax over all available actions to get a probability distribution. We use PPO (Schulman et al., 2017b) to train the policy. PPO requires a value function $V(s')$ as a baseline. We represent the effective state as the concatenation of the state encoding s and the action set summary \bar{c}^R inferred from the GAT, $s' = (s, \bar{c}^R)$. An important implementation detail to make the training faster and stable was to *not* share the GAT weights used for the actor $\pi(a|s', c_a^R)$ and the critic $V(s')$ networks.

Value-based RL (DQN) The output action utility can be directly treated as the $Q(s', a)$ value of the action a at the current effective state s' with the available action set \mathcal{A} . This can be trained using standard Deep Q-learning Bellman backup (Mnih et al., 2015).

Listwise RL (CDQN): For tasks with listwise actions, we follow the Cascaded DQN (CDQN) framework of Chen et al. (2019a). The main challenge is that building the action list all at once is not feasible due to a combinatorial number of possible list-actions. Therefore, the key is to build the list incrementally, one action at a time. Thus, each list index can be treated as an individual action decision trained with independent Q-networks. We replace the Q-network of CDQN with AGILE

to support a varying action space. Sharing the weights of the cascaded Q-networks led to better performance. Algorithm 5 provides complete details on CDQN for listwise AGILE.

3.5 Environments

We evaluate AGILE on three varying action set scenarios requiring learning action interdependence: (i) shortcut actions in goal-reaching, which can shorten the optimal path to the goal in a 2D Grid World when available, (ii) co-dependent actions in tool reasoning, which require other tools to activate their functionality, and (iii) list-actions in simulated and real-data recommender systems where the cumulative list affects the user response. Figure 3.3 provides an overview of the tasks, the base and varying action space, and an illustration of the action interdependence. More environment details such as tasks, action representations, and data collection are present in Appendix A.

3.5.1 Dig Lava Grid Navigation

We modify the grid world environment (Chevalier-Boisvert et al., 2018) where an agent navigates a 2D maze with two lava rivers to reach a goal. The agent always has access to four direction movements and a *turn-left* skill (Figure 3.5). There are two additional actions randomly sampled out of four special skills: *turn-right*, *step-forward*, *dig-orange-lava* and *dig-pink-lava*. If the agent enters the lava, it will die unless it uses the matching *dig-lava* skill to remove the lava in the immediately next timestep. Thus, when available, *dig-lava* skills can be used to create shortcut paths to the goal and receive a higher reward. We use PPO for all experiments in this environment.

3.5.2 Chain REAction Tool Environment: CREATE

The CREATE environment (Jain et al., 2020) is a challenging physical reasoning benchmark with a large variety of tools as actions, supporting evaluation with unseen actions. The objective is to sequentially place tools to help push the red ball towards the green goal. An action is a hybrid of a discrete tool-selection from varying toolsets and a continuous (x, y) coordinate of tool-placement on

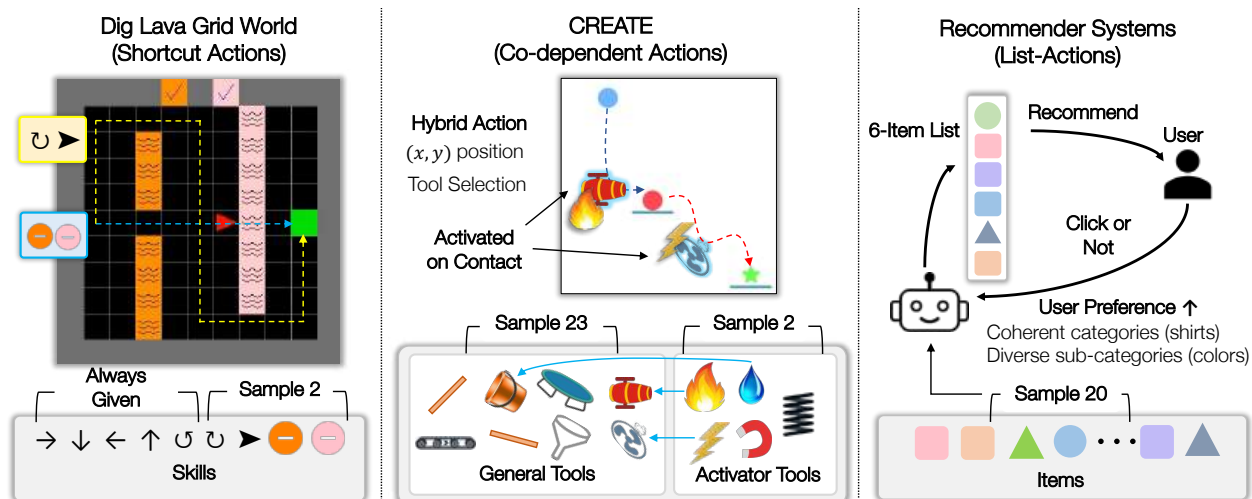


Figure 3.3: **Environment Setup.** (Left) In Grid World, the red agent must avoid orange and pink lava to reach the green goal. The *dig-lava* skills enable shortcut paths to the goal when available. (Middle) In CREATE, tools are selected and placed sequentially to push the red ball to the green goal. General tools (cannon, fan) require activator tools (fire, electric) to start functioning. The choice of general tools depends on what activators are available and vice-versa. (Right) In simulated and real-data recommender systems, the agent selects a list of items. Lists with coherent categories and complementary sub-categories improve user satisfaction (e.g., click likelihood).

the screen. An auxiliary policy network decides the tool-placement based on the effective state s' as input, following Jain et al. (2020). To emphasize action relations, we augment the environment with special activator tools (e.g., fire) that general tools (e.g., cannon) need in contact to be functional. Thus, a general tool can be useful only if its activator is also available. Action representations of general tools encode their physical behavior (Jain et al., 2020), while those of activator tools are one-hot vectors. We train AGILE and the auxiliary policy jointly with PPO.

3.5.3 Recommender Systems

Recommender system (RecSys) is a natural application of varying action space RL — for instance, news articles or videos to recommend are updated daily. Action interdependence is distinctly apparent in the case of listwise actions. A recommended list of diverse videos is more likely to get a user click than videos about the same thing (Zhou et al., 2010). We experiment with the listwise metric of Complementary Product Recommendation (CPR) (Hao et al., 2020).

Complementary Product Recommendation (CPR) is a scenario where user response for a recommended list is more favorable if the list is diverse at a low level but coherent at a high level. In our experiments, each item has a primary category (such as shirt or pant) and a subcategory (such as its color). We define the CPR of an item-list as, $\frac{\text{Entropy of subcategory}}{\text{Entropy of category}}$. This encourages diversity in subcategories (colors) and similarity in the main category (all shirts). We maximize CPR (i) implicit in the click-behaviors of simulated users and (ii) explicit in the reward computed on real-world data.

Simulated Recommender System: RecSim. We use RecSim (Je et al., 2019a) to simulate user interactions and extend it to the listwise recommendation task. We have a base action set of 250 train and 250 test items, and 20 items are sampled as actions for the agent in each episode. The agent recommends a list-action of size six at each step. We assume a fully observable environment with the state as the user preference vector and the action representations as item characteristics. The objective implicitly incorporates CPR by boosting the probability of a user clicking any item proportional to the list CPR. The implicit CPR objective exemplifies realistic scenarios where the entire list influences user response. One way to optimize CPR is to identify the most common category in the available action set and recommend most items from that category. Such counting of categories requires relational reasoning over all items available in the action set. We train CDQN-based models to maximize the number of clicks in a user session.

Real-Data recommender system. We collect four-week interaction data in a listwise online campaign recommender system. Users are represented by attributes such as age, occupation, and localities. Item attributes include text features, image features, and reward points of campaigns. We train a VAE (Kingma and Welling, 2013) to learn item representations. We create a representative RL environment by training two click-estimation models using data from the first two weeks for training and the last two weeks for evaluation. The training environment consists of 68,775 users and 57 items, while testing has 82,445 users and 58 items, with an overlap of 30 items. The reward function combines the user-click and CPR value of the list. The explicit CPR reward is a representative scenario for when the designer has listwise objectives in addition to user satisfaction. We train with CDQN and report the test reward.

3.6 Experiments

We design experiments to answer the following questions in the context of RL with varying action space: (1) How effective is AGILE compared to prior works that treat actions independently or assume a fixed action set? (2) How effective are AGILE’s relational action representations for computing action set summary and action utility score? (3) Does the attention in AGILE represent meaningful action relations? (4) Is attention necessary in the graph network of AGILE? (5) Is learning state-dependent action relations important for solving general varying action space tasks?

3.6.1 Effectiveness of AGILE in Varying Action Spaces

We evaluate baselines from prior work in varying action spaces, which either assume a fixed action set or act independently of other actions. We ablate the importance of relational action features by replacing them with original action representations and computing the action set summary in different ways. Refer to the Appendix for detailed comparisons (C.1) and visualizations (Figure B.10) of all baselines and ablations, hyperparameter-tuning (D.2,D.3) and network designs(C.3).

Baselines.

- **Mask-Output** (No action representations, No input action set): Assumes a fixed action space output, Q-values or policy probabilities are masked out for unavailable actions, follows SAS-MDP works: [Boutillier et al. \(2018\)](#); [Chandak et al. \(2020a\)](#); [Huang and Ontañón \(2020\)](#).
- **Mask-Input-Output** (No action representations): Augments *Mask-Output* with action set input via a binary availability vector: having 1s at available action indices and 0s otherwise.
- **Utility-Policy** (No input action set): [Jain et al. \(2020\)](#) propose a parallel architecture to compute each action’s utility using action representations, but ignores any action interdependence.
- **Simple DQN** (No cascade, No input action set): A non-cascaded DQN baseline for listwise RL that selects top-K items instead of reasoning about the entire list. Thus, it ignores two action interdependences: (i) on other items in the list and (ii) on other available actions.

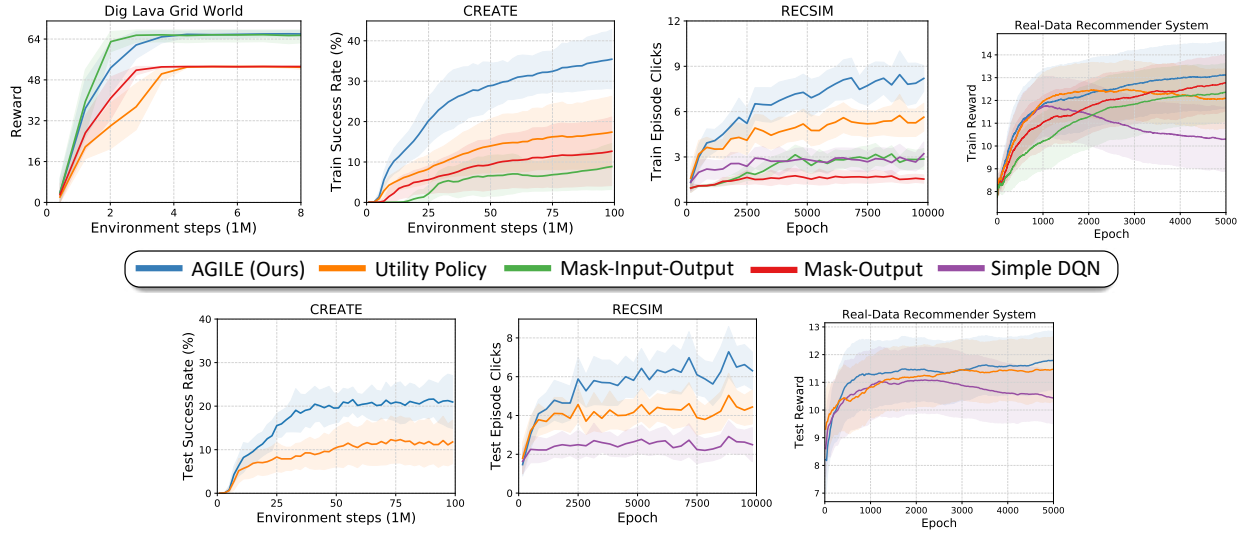


Figure 3.4: We evaluate AGILE against baselines on training actions (top) and unseen testing actions (bottom). Generalization is enabled by continuous action representations, except Grid World (Appendix A). All architectures share the same RL algorithm (PPO or CDQN). The results are averaged over 5 seeds, with seed variance shown with shading. AGILE outperforms all baselines that assume a fixed action set (cannot generalize) or treat actions independently (suboptimal).

Ablations.

- **Summary-LSTM:** A Bi-LSTM (Huang et al., 2015) encodes the list of action representations.
- **Summary-Deep Set:** A deep set architecture (Zaheer et al., 2017a) with mean pooling is used to aggregate the available action representation list.
- **Summary-GAT:** The relational action representations output from the GAT in AGILE is used only for the action set summary but not for the utility network. This does not scale to tasks with many diverse inter-action relations because the summary vector has a limited modeling capacity.

Results.

Baselines: Figure 3.4 shows baseline results on train and test actions. Grid world has no unseen actions, so we report only train reward. **Grid World:** all methods learn to reach the goal with perfect success but achieve different rewards due to the length of the path. The methods that do not take action set as input, *Mask-Output* and *Utility Policy*, resort to the safe strategy of going

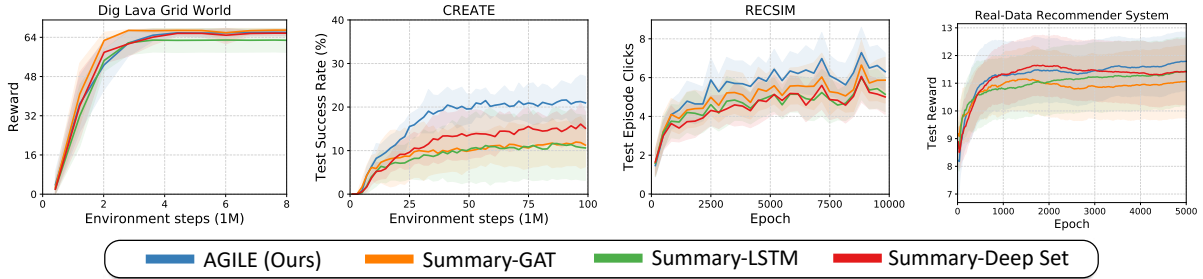


Figure 3.5: Test performance of AGILE against ablations with utility network using only action set summary, but not the relational action representations. The difference is most pronounced in CREATE, where there are several diverse action (tool) relations for each action decision.

around the lava rivers (Figure 3.6(b)) because the agent must enter the lava and then dig it to take a shortcut. So before taking the *move-right* action into lava, the agent must know whether that *dig-lava* skill is available. **CREATE**: *Mask* baselines do not support generalization due to fixed action set assumption and train poorly as they do not exploit the structure in action representations. *Utility policy* outperforms the other baselines since it can exploit learning with action representations. Finally, AGILE outperforms all the baselines, demonstrating that relational knowledge of other available actions is crucial for an optimal policy. **RecSim** and **Real RecSys**: result trends are consistent with CREATE, but less pronounced for Real RecSys. Additionally, DQN is worse than CDQN-based architectures because the top-K greedy list-action ignores intra-list dependence.

Ablations: Figure 3.5 shows ablation results on test actions. **Grid World**: all ablations utilize the action set summary as input, aggregated via different mechanisms. Thus, they can identify which *dig-lava* skills are available and enter lava accordingly to create shortcuts. In such small action spaces with simple action relations, summary-ablations are on par with AGILE. This trend also holds for **RecSim** and **Real RecSys**, where the summary can find the most common category and its items are then selected to maximize CPR (e.g., Figure 3.6(c)). Therefore, we observe only 5 – 20% gains of AGILE over the ablations. To test the consistency of results, we further evaluate two more RecSim tasks. (i) Direct CPR: the agent receives additional explicit CPR metric reward on top of click/no-click reward (Sec. B.3), and (ii) pairing environment: the task is to recommend pairs of associated items based on predefined pairings (Sec. B.4). We reproduce the trend that $AGILE \geq$ ablations. However, the difference is most pronounced (30 – 50%) in **CREATE**, where

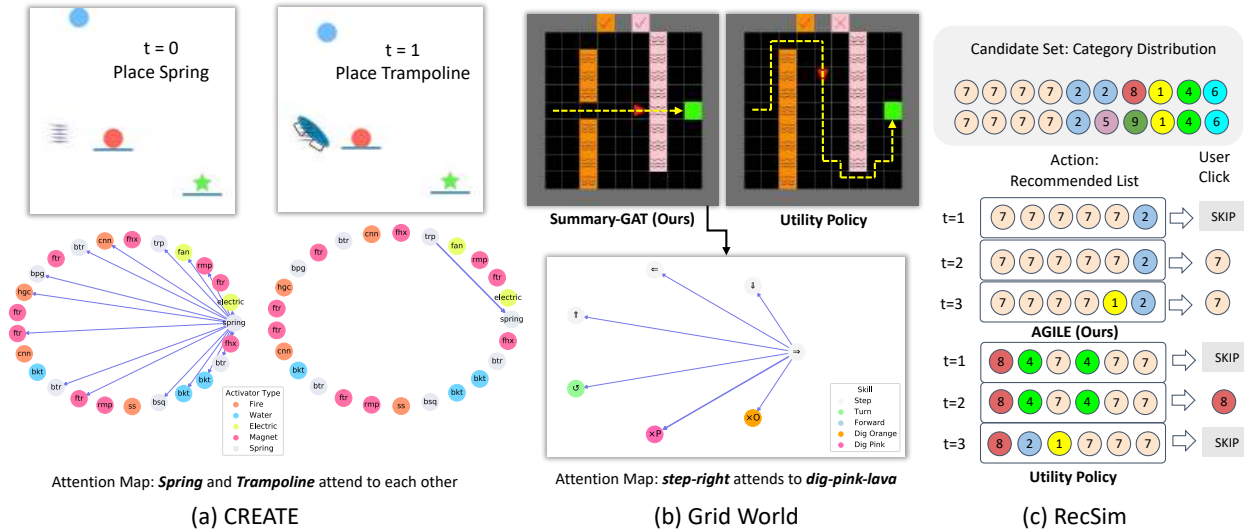


Figure 3.6: Qualitative Analysis. (a,b) The attention maps from GAT show the reasoning behind an action decision. The nodes show available actions, and edge widths are proportional to attention weight (thresholded for clarity). (b) Utility Policy learns the same suboptimal solution for any given action set, while Summary-GAT (like AGILE) adapts to the best strategy by exploiting dig-skills. (c) AGILE can optimize CPR by identifying the most common item category available, unlike Utility Policy. We provide qualitative video results for Grid World and CREATE on the project page <https://sites.google.com/view/varyingaction>.

each action decision relies on relations between various tools and activators. In Figure 3.6(a), the decision of *Spring* relates with all other tools that spring can activate, and the decision of *trampoline* relates with its activator, *Spring*. In ablations, the action set summary must model all the complex and diverse action relations with a limited representation capacity. In contrast, the relational action representations in AGILE model each action’s relevant relations, and the summary models the global relations.

3.6.2 Does the Attention in AGILE Learn Meaningful Action Relations?

In Figure 3.6, we analyze the agent performance qualitatively. (a) In CREATE, at $t = 0$, the selected action *spring* in AGILE’s GAT attends to various other tools, especially the tools that get activated with *spring*, such as *trampoline*. At $t = 1$, the *trampoline* tool is selected with strong attention on *spring*. This shows that for selecting the *trampoline*, the agent checks for its activator,

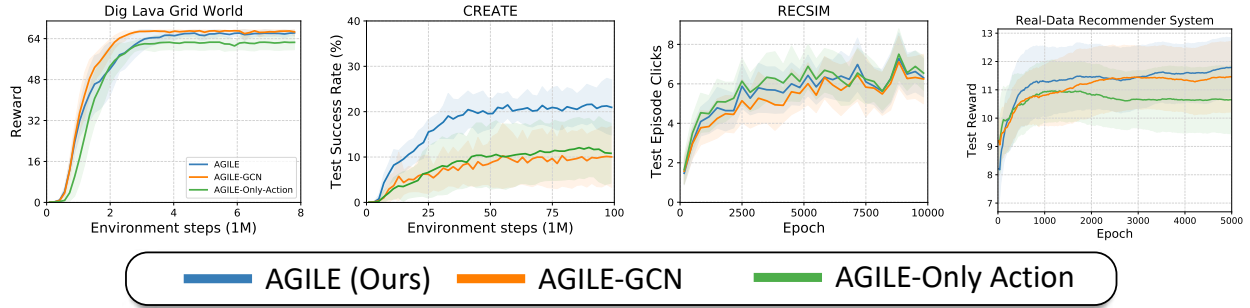


Figure 3.7: Analyses. (i) GAT v/s GCN (ii) state-action relations v/s action-only relations.

spring, to ensure that it is possible to place *spring* before or after the trampoline. (b) In Grid World, we visualize the inter-action attention in *Summary-GAT*'s summarizer. We consider the case where both *dig* – *lava* skills are available. The agent goes right, digs the orange lava, and is about to enter the pink lava. At this point, the *Right* action attends with a large weight to the *Dig* – *Pink* skill, checking for its presence before making an irreversible decision of entering the lava. In contrast, the *Utility Policy* always follows the safe suboptimal path as it is blind to the knowledge of dig-skills before entering lava. (c) In RecSim, we observe that the agent can maximize the CPR score by selecting 5 out of 6 items in the list from the same primary category. In contrast, *Utility Policy* cannot determine the most common available category and is unable to maximize CPR.

3.6.3 Additional Analyses

Importance of Attention in the Graph Network. We validate the choice of using graph attention network as the relational architecture. In Figure 3.7, we compare GAT against a graph convolutional network (GCN) (Kipf and Welling, 2016) to act over AGILE's action graph. We observe that GCN achieves optimal performance for the grid world and RecSys tasks. GCN can learn simple action relations even though the edge weights are not learned. However, it suffers in CREATE and RecSim-pairing (Figure B.9), where the action relations are diverse and plenty. Moreover, we believe that the attention in GAT makes the graph sparse to ease RL training, which in contrast, is difficult in a fully-connected GCN.

Importance of state-dependent learning of action relations. We evaluate a version of AGILE where the GAT only receives action representations as input and no state. Thus, the action relations are inferred independently of the state. Figure 3.7 shows a drop in performance for Grid World and CREATE, where the relevant action relations change based on the state. However, this effect is less apparent on RecSim because CPR requires only knowing the most common category, independent of user state.

3.7 Conclusion

In this chapter, we present AGILE, a policy architecture for leveraging action relations for reinforcement learning with varying action spaces. AGILE builds a complete graph of available actions' representations and utilizes a graph attention network to learn the interdependence between actions. We demonstrate that using the knowledge of available actions is crucial for optimal decision-making and relational action features ease learning in four environments, including a real-data recommendation task.

Part II

Near-optimal Action in Non-convex Q-functions

Chapter 4

Optimizing Action in Non-Convex Q-functions via Successive Actors

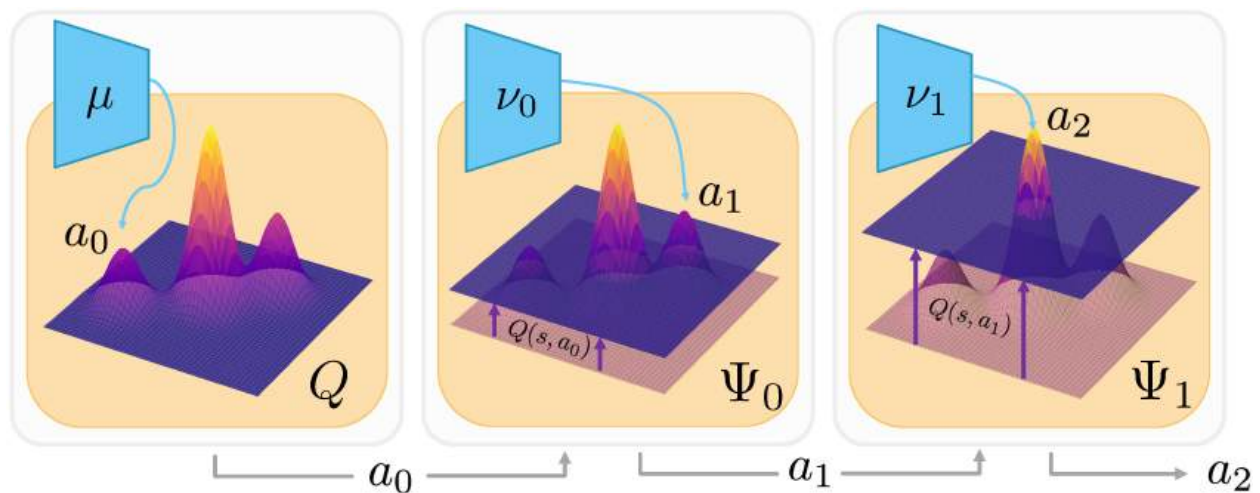


Figure 4.1: An actor μ trained with gradient ascent on a challenging Q-landscape gets stuck in local optima. Our approach learns a sequence of surrogates Ψ_i of the Q-function that successively prune out the Q-landscape below the current best Q-values, resulting in fewer local optima. Thus, the actors ν_i trained to ascend on these surrogates produce actions with a more optimal Q-value.

4.1 Introduction

In sequential decision-making, the goal is to build an optimal agent that maximizes the expected cumulative returns (Sondik, 1971; Littman, 1996). Value-based reinforcement learning (RL)

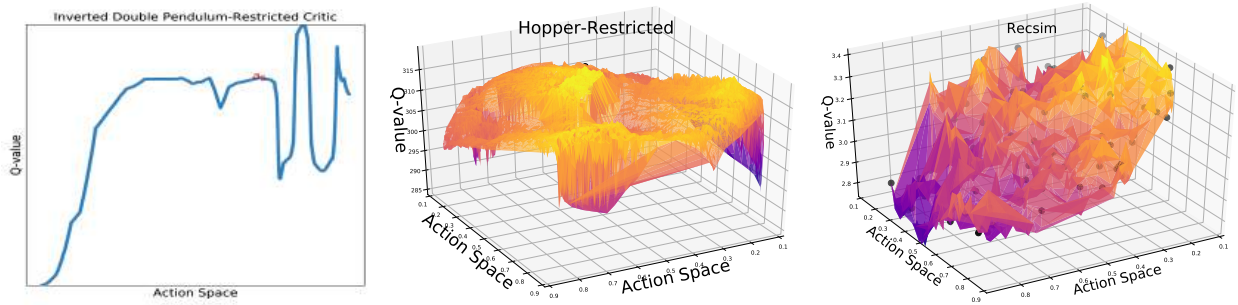


Figure 4.2: **Complex Q-landscapes.** We plot Q-value versus action a for some state. In control of Inverted-Double-Pendulum-Restricted (left) and Hopper-Restricted (middle), certain action ranges are unsafe, resulting in various locally optimal action peaks. In a large discrete-action recommendation system (right), there are local peaks at actions representing real items (black dots).

approaches learn each action’s expected returns with a Q-function and maximize it (Sutton and Barto, 1998). However, in continuous action spaces, evaluating the Q-value of every possible action is impractical. This necessitates an actor to globally maximize the Q-function and efficiently navigate the vast action space (Grondman et al., 2012). But this is particularly challenging in tasks like restricted locomotion, where the *non-convex* Q-function has many local optima (Figure 4.2).

Can we build an actor architecture to find near-optimal actions in such complex Q-landscapes? Prior methods perform a search over the action space with evolutionary algorithms like CEM (De Boer et al., 2005; Kalashnikov et al., 2018; Shao et al., 2022), but this requires numerous costly re-evaluations of the Q-function. To avoid this, deterministic policy gradient (DPG) algorithms (Silver et al., 2014), such as DDPG (Lillicrap et al., 2015) and TD3 (Fujimoto et al., 2018) train a parameterized actor with gradient ascent to output actions to maximize the Q-function *locally*.

A significant challenge arises in environments where the Q-function has many local optima, such as Figure 4.2. An actor trained via gradient ascent may converge to a local optimum with a much lower Q-value than the global maximum. This leads to *suboptimal* decisions during deployment and *sample-inefficient* training, as the agent fails to explore high-reward trajectories (Kakade, 2003).

To improve actors’ ability to identify optimal actions in complex, non-convex Q-function landscapes, we propose the Successive Actors for Value Optimization (SAVO) algorithm. SAVO leverages two key insights: (1) combining multiple policies using an arg max on their Q-values to

construct a superior policy (§4.4.1), and (2) simplifying the Q-landscape by excluding lower Q-value regions based on high-performing actions, inspired by tabu search (Glover, 1990), thereby reducing local optima and facilitating gradient-ascent (§4.4.2). By iteratively applying these strategies through a sequence of simplified Q-landscapes and corresponding actors, SAVO progressively finds more optimal actions.

We evaluate SAVO in complex Q-landscapes such as (i) *continuous* control in dexterous manipulation (Rajeswaran et al., 2017) and restricted locomotion (Todorov et al., 2012), and (ii) *discrete* decision-making in the large action spaces of simulated (Ie et al., 2019a) and real-data recommender systems (Harper and Konstan, 2015), and gridworld mining expedition (Chevalier-Boisvert et al., 2018). We reframe large discrete action RL to continuous action RL following Van Hasselt and Wiering (2009) and Dulac-Arnold et al. (2015), where a policy acts in continuous actions, such as the feature space of recommender items (Figure 4.2), and the nearest discrete action is executed.

Our key contribution is SAVO, an actor architecture to find better optimal actions in complex non-convex Q-landscapes (Section 4.4). In experiments, we visualize how SAVO’s successively learned Q-landscapes have fewer local optima (Section 4.6.2), making it more likely to find better action optima with gradient ascent. This enables SAVO to outperform alternative actor architectures, such as sampling more action candidates (Dulac-Arnold et al., 2015) and learning an ensemble of actors (Osband et al., 2016) (Section 4.6.1) across continuous and discrete action RL.

4.2 Related Work

Q-learning (Watkins and Dayan, 1992; Tesauro et al., 1995) is a fundamental value-based RL algorithm that iteratively updates Q-values to make optimal decisions. Deep Q-learning (Mnih et al., 2015) has been applied to tasks with manageable discrete action spaces, such as Atari (Mnih et al., 2013; Espeholt et al., 2018; Hessel et al., 2018), traffic control (Abdoos et al., 2011), and small-scale recommender systems (Chen et al., 2019a). However, scaling Q-learning to continuous or large discrete action spaces requires specialized techniques to efficiently maximize the Q-function.

Analytical Q-optimization. Analytical optimization of certain Q-functions, such as wire fitting algorithm (Baird and Klopff, 1993) and normalized advantage functions (Gu et al., 2016; Wang et al., 2019), allows closed-form action maximization without an actor. Likewise, Amos et al. (2017) assume that the Q-function is convex in actions and use a convex solver for action selection. In contrast, the Q-functions considered in this paper are inherently non-convex in action space, making such an assumption invalid. Generally, analytical Q-functions lack the expressiveness of deep Q-networks (Hornik et al., 1989), making them unsuitable to model complex tasks like in Figure 4.2.

Evolutionary Algorithms for Q-optimization. Evolutionary algorithms like simulated annealing (Kirkpatrick et al., 1983), genetic algorithms (Srinivas and Patnaik, 1994), tabu search (Glover, 1990), and the cross-entropy method (CEM) (De Boer et al., 2005) are employed in RL for global optimization (Hu et al., 2007). Approaches such as QT-Opt (Kalashnikov et al., 2018; Lee et al., 2023; Kalashnikov et al., 2021a) utilize CEM for action search, while hybrid actor-critic methods like CEM-RL (Pourchot and Sigaud, 2018), GRAC (Shao et al., 2022), and Cross-Entropy Guided Policies (Simmons-Edler et al., 2019) combine evolutionary techniques with gradient descent. Despite their effectiveness, CEM-based methods require numerous Q-function evaluations and struggle with high-dimensional actions (Yan et al., 2019). In contrast, SAVO achieves superior performance with only a few (e.g., three) Q-evaluations, as demonstrated in experiments (Section 4.6).

Actor-Critic Methods with Gradient Ascent. Actor-critic methods can be on-policy (Williams, 1992; Schulman et al., 2015; 2017b) primarily guided by the policy gradient of expected returns, or off-policy (Silver et al., 2014; Lillicrap et al., 2015; Fujimoto et al., 2018; Chen et al., 2020) primarily guided by the Bellman error on the critic. Deterministic Policy Gradient (DPG) (Silver et al., 2014) and its extensions like DDPG Lillicrap et al. (2015), TD3 (Fujimoto et al., 2018) and REDQ (Chen et al., 2020) optimize actors by following the critic’s gradient. Soft Actor-Critic (SAC) (Haarnoja et al., 2018) extends DPG to stochastic actors. However, these methods can get trapped in local optima within the Q-function landscape. SAVO addresses this limitation by enhancing gradient-based actor training. This issue also affects stochastic actors, where a local

optimum means an *action distribution* (instead of a single action) that fails to minimize the KL divergence from the Q-function density fully, and is a potential area for future research.

Sampling-Augmented Actor-Critic. Sampling multiple actions and evaluating their Q-values is a common strategy to find optimal actions. Greedy actor-critic (Neumann et al., 2018) samples high-entropy actions and trains the actor towards the best Q-valued action, yet remains susceptible to local optima. In large discrete action spaces, methods like Wolpertinger (Dulac-Arnold et al., 2015) use k-nearest neighbors to propose actions, requiring extensive Q-evaluations on up to 10% of total actions. In contrast, SAVO efficiently generates high-quality action proposals through successive actor improvements without being confined to local neighborhoods.

Ensemble-Augmented Actor-Critic. Ensembles of policies enhance exploration by providing diverse action proposals through varied initializations (Osband et al., 2016; Chen and Peng, 2019; Song et al., 2023; Zheng12 et al., 2018; Huang et al., 2017). The best action is selected based on Q-value evaluations. Unlike ensemble methods, SAVO systematically eliminates local optima, offering a more reliable optimization process for complex tasks (Section 4.6).

4.3 Problem Formulation

Our work tackles the effective optimization of the Q-value landscape in off-policy actor-critic methods for continuous and large-discrete action RL. We model a task as a Markov Decision Process (MDP), defined by a tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{T}, R, \gamma\}$ of states, actions, transition probabilities, reward function, and a discount factor. The action space $\mathcal{A} \subseteq \mathbb{R}^D$ is a D -dimensional *continuous* vector space. At every step t in the episode, the agent receives a state observation $s_t \in \mathcal{S}$ from the environment and acts with $a_t \in \mathcal{A}$. Then, it receives the new state after transition s_{t+1} and a reward r_t . The objective of the agent is to learn a policy $\pi(a | s)$ that maximizes the expected discounted reward, $\max_{\pi} \mathbb{E}_{\pi} [\sum_t \gamma^t r_t]$.

4.3.1 Deterministic Policy Gradients (DPG)

DPG (Silver et al., 2014) is an off-policy actor-critic algorithm that trains a deterministic actor μ_ϕ to maximize the Q-function. This happens via two steps of generalized policy iteration, GPI (Sutton and Barto, 1998): policy evaluation estimates the Q-function (Bellman, 1966) and policy improvement greedily maximizes the Q-function. To approximate the $\arg \max$ over continuous actions in Eq. (4.2), DPG proposes the policy gradient to update the actor locally in the direction of increasing Q-value,

$$Q^\mu(s, a) = r(s, a) + \gamma \mathbb{E}_{s'} [Q^\mu(s', \mu(s'))], \quad (4.1)$$

$$\mu(s) = \arg \max_a Q^\mu(s, a), \quad (4.2)$$

$$\nabla_\phi J(\phi) = \mathbb{E}_{s \sim \rho^\mu} \left[\nabla_a Q^\mu(s, a) \Big|_{a=\mu(s)} \nabla_\phi \mu_\phi(s) \right]. \quad (4.3)$$

DDPG (Lillicrap et al., 2015) and TD3 (Fujimoto et al., 2018) made DPG compatible with deep networks via techniques like experience replay and target networks to address non-stationarity of online RL, twin critics to mitigate overestimation bias, target policy smoothing to prevent exploitation of errors in the Q-function, and delayed policy updates so critic is reliable to provide actor gradients.

4.3.2 The Challenge of an Actor Maximizing a Complex Q-landscape

DPG-based algorithms train the actor following the chain rule in Eq. (4.3). Specifically, its first term, $\nabla_a Q^\mu(s, a)$ involves gradient ascent in Q-versus- a landscape. This Q-landscape is often highly non-convex (Figures 4.2, 4.3) and non-stationary because of its own training. This makes the actor's output $\mu(s)$ get stuck at suboptimal Q-values, thus leading to insufficient policy improvement in Eq. (4.2). We can define the suboptimality of the μ w.r.t. Q^μ at state s as

$$\Delta(Q^\mu, \mu, s) = \arg \max_a Q^\mu(s, a) - Q^\mu(s, \mu(s)) \geq 0. \quad (4.4)$$

Suboptimality in actors is a crucial problem because it leads to (i) **poor sample efficiency** by slowing down GPI, and (ii) **poor inference performance** even with an optimal Q-function, Q^* as seen in Figure 4.3 where a TD3 actor gets stuck at a locally optimum action a_0 in the final Q-function.

This challenge fundamentally differs from the well-studied field of non-convex optimization, where non-convexity arises in the *loss function w.r.t. the model parameters* (Goodfellow, 2016). In those cases, stochastic gradient-based optimization methods like SGD and Adam (Kingma and Ba, 2014) are effective at finding acceptable local minima due to the smoothness and high dimensionality of the parameter space, which often allows for escape from poor local optima (Choromanska et al., 2015). Moreover, overparameterization in deep networks can lead to loss landscapes with numerous good minima (Neyshabur et al., 2017).

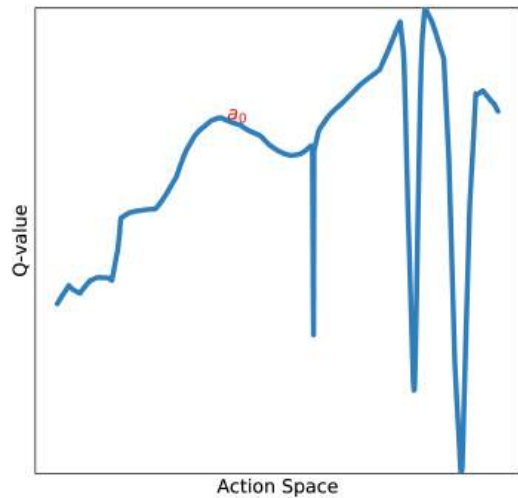


Figure 4.3: Non-convex Q-landscape in Inverted-Pendulum-Restricted leads to a suboptimally converged actor.

In contrast, our challenge involves non-convexity in the *Q-function w.r.t. the action space*. The actor’s task is to find, for every state s , the action a that maximizes $Q^\mu(s, a)$. Since the Q-function can be highly non-convex and multimodal in a , the gradient ascent step $\nabla_a Q^\mu(s, a)$ used in Eq. (4.3) may lead the actor to converge to suboptimal local maxima in action space. Unlike parameter space optimization, the actor cannot rely on high dimensionality or overparameterization to smooth out the optimization landscape in action space because the Q-landscape is determined by the task’s reward. Furthermore, the non-stationarity of the Q-function during training compounds this challenge. These properties make our non-convex challenge unique, requiring a specialized actor to navigate the complex Q-landscape.

Tasks with several local optima in the Q-function include restricted inverted pendulum as shown in Figure 4.3, where certain regions of the action space are invalid or unsafe, leading to a rugged

Q-landscape (Florence et al., 2022). Dexterous manipulation exhibits discontinuous behaviors like inserting a precise peg in place with a small region of high-valued actions (Rajeswaran et al., 2017) and surgical robotics have a high variance in Q-values of nearby motions (Barnoy et al., 2021).

Large Discrete Action RL Reframed as Continuous Action RL. We discuss another practical domain where non-convex Q-functions are present. In large discrete action tasks like recommender systems (Zhao et al., 2018; Zou et al., 2019; Wu et al., 2017), a common approach (Van Hasselt and Wiering, 2009; Dulac-Arnold et al., 2015) is to use continuous representations of actions as a medium of decision-making. Given a set of actions, $\mathcal{I} = \{\mathcal{I}_1, \dots, \mathcal{I}_N\}$, a predefined module $\mathcal{R} : \mathcal{I} \rightarrow \mathcal{A}$ assigns each $\mathcal{I} \in \mathcal{I}$ to its representation $\mathcal{R}(\mathcal{I})$, e.g., text embedding of a given movie (Zhou et al., 2010). A continuous action policy $\pi(a | s)$ is learned in the action representation space, with each $a \in \mathcal{A}$ converted to a discrete action $\mathcal{I} \in \mathcal{I}$ via nearest neighbor,

$$f_{\text{NN}}(a) = \arg \min_{\mathcal{I}_i \in \mathcal{I}} \|\mathcal{R}(\mathcal{I}_i) - a\|_2.$$

Importantly, the nearest neighbor operation creates a challenging piece-wise continuous Q-function with suboptima at various discrete points as shown in Figure 4.2 (Jain et al., 2021; 2020).

4.4 Approach: Successive Actors for Value Optimization (SAVO)

Our objective is to design an actor architecture that efficiently discovers better actions in complex, non-convex Q-function landscapes. We focus on gradient-based actors and introduce two key ideas:

1. **Maximizing Over Multiple Policies:** By combining policies using an $\arg \max$ over their Q-values, we can construct a policy that performs at least as well as any individual policy (Section 4.4.1).
2. **Simplifying the Q-Landscape:** Drawing inspiration from tabu search (Glover, 1990), we propose using actions with good Q-values to eliminate or “tabu” the Q-function regions with lower Q-values, thereby reducing local optima and facilitating gradient-based optimization (Section 4.4.2).

4.4.1 Maximizer Actor over Action Proposals

We first show how additional actors can improve DPG’s policy improvement step. Given a policy μ being trained with DPG over Q , consider k additional arbitrary policies ν_1, \dots, ν_k , where $\nu_i : \mathcal{S} \rightarrow \mathcal{A}$ and let $\nu_0 = \mu$. We define a maximizer actor μ_M for $a_i = \nu_i(s)$ for $i = 0, 1, \dots, k$,

$$\mu_M(s) := \arg \max_{a \in \{a_0, a_1, \dots, a_k\}} Q(s, a), \quad (4.5)$$

Here, μ_M is shown to be a better maximizer of $Q(s, a)$ in Eq. (4.2) than $\mu \forall s$:

$$Q(s, \mu_M(s)) = \max_{a_i} Q(s, a_i) \geq Q(s, a_0) = Q(s, \mu(s)).$$

Therefore, by policy improvement theorem (Sutton and Barto, 1998), $V^{\mu_M}(s) \geq V^\mu(s)$, proving that μ_M is better than a single μ for a given Q . Appendix A proves the following theorem by showing that policy evaluation and improvement with μ_M converge.

Theorem 4.4.1 (Convergence of Policy Iteration with Maximizer Actor). *A modified policy iteration algorithm where $\nu_0 = \mu$ is the current policy learned with DPG and maximizer actor μ_M defined in Eq. (4.5), converges in the tabular setting to the optimal policy.*

This algorithm is valid for arbitrary ν_1, \dots, ν_k . We experiment with ν ’s obtained by **sampling** from a Gaussian centered at μ or **ensembling** on μ to get diverse actions. However, in high-dimensionality, *randomness* around μ is not sufficient to get action proposals to significantly improve μ .

4.4.2 Successive Surrogates to Reduce Local Optima

To train additional policies ν_i that can improve upon μ_M , we introduce *surrogate* Q-functions with fewer local optima, inspired by the principles of tabu search (Glover, 1990), which is an optimization technique that uses memory structures to avoid revisiting previously explored inferior solutions, thereby enhancing the search for optimal solutions. Similarly, our surrogate functions act

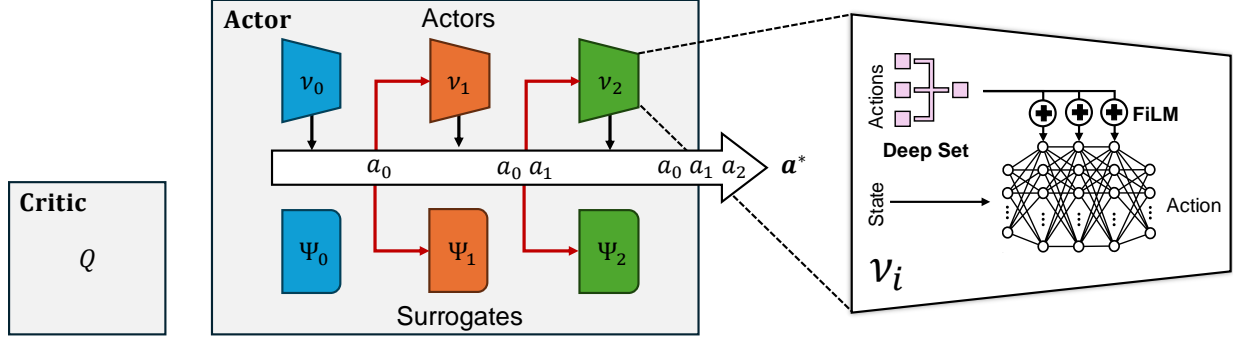


Figure 4.4: **SAVO Architecture.** (left) Q-network is unchanged. (center) Instead of a single actor, we learn a sequence of actors and surrogate networks connected via action predictions. (right) Conditioning on previous actions is done with the help of a deep-set summarizer and FiLM modulation.

as memory mechanisms that “tabu” certain regions of the Q-function landscape deemed suboptimal based on previously identified good actions. Given a known action a^\dagger , we define a surrogate function that elevates the Q-values of all inferior actions to $Q(s, a^\dagger)$, which serves as a constant threshold:

$$\Psi(s, a; a^\dagger) = \max\{Q(s, a), Q(s, a^\dagger)\}. \quad (4.6)$$

Extending this idea, we define a sequence of surrogate functions using the actions from previous policies. Let $a_{<i} = \{a_0, a_1, \dots, a_{i-1}\}$. The i -th surrogate function is:

$$\Psi_i(s, a; a_{<i}) = \max\left\{Q(s, a), \max_{j<i} Q(s, a_j)\right\}. \quad (4.7)$$

Theorem 4.4.2. *For a state $s \in \mathcal{S}$ and surrogates Ψ_i defined as above, the number of local optima decreases with each successive surrogate:*

$$N_{opt}(Q(s, \cdot)) \geq N_{opt}(\Psi_1(s, \cdot; a_0)) \geq \dots \geq N_{opt}(\Psi_k(s, \cdot; a_{<k})),$$

where $N_{opt}(f)$ denotes the number of local optima of function f over \mathcal{A} .

Proof Sketch. As $\Psi_i \rightarrow \Psi_{i+1}$, the anchor Q-value in Eq. (4.7) weakly increases, $\max_{j<i} Q(s, a_j) \leq \max_{j<(i+1)} Q(s, a_j)$, thus, eliminating more local minima below it (proof in Appendix B.1). \square

4.4.3 Successive Actors for Surrogate Optimization

To effectively reduce local optima using the surrogates Ψ_1, \dots, Ψ_k , we design the policies ν_i to optimize their respective surrogates $\Psi_i(s, a; a_{<i})$. Each ν_i focuses on regions where $Q(s, a) \geq \max_{j < i} Q(s, a_j)$, allowing it to find better optima than previous policies. The actor ν_i is conditioned on previous actions $\{a_0, \dots, a_{i-1}\}$, summarized using deep sets (Zaheer et al., 2017b) (Figure 4.4). The maximizer actor μ_M (Eq. (4.5)) selects the best action among all proposals.

We train each actor ν_i using gradient ascent on its surrogate Ψ_i , similar to DPG:

$$\nabla_{\phi_i} J(\phi_i) = \mathbb{E}_{s \sim \rho^{\mu_M}} \left[\nabla_a \Psi_i(s, a; a_{<i}) \Big|_a \nabla_{\phi_i} \nu_i(s; a_{<i}) \right]. \quad (4.8)$$

4.4.4 Approximate Surrogate Functions

The surrogates Ψ_i have zero gradients when $Q(s, a) < \tau$, where $\tau = \max_{j < i} Q(s, a_j)$,

$$\nabla_a \Psi_i(s, a; a_{<i}) = \begin{cases} \nabla_a Q^{\mu_M}(s, a) & \text{if } Q(s, a) \geq \tau, \\ 0 & \text{if } Q(s, a) < \tau. \end{cases}$$

This means the policy gradient only updates ν_i when $Q(s, a) \geq \tau$, which may slow down learning. To address this issue, we ease the gradient flow by learning a smooth lossy approximation $\hat{\Psi}_i$ of Ψ_i .

We approximate each surrogate Ψ_i with a neural network $\hat{\Psi}_i$. This approach leverages the universal approximation theorem (Hornik et al., 1989; Cybenko, 1989) and benefits from empirical evidence that deep networks can effectively learn non-smooth functions (Imaizumi and Fukumizu, 2019). The smooth surrogate $\hat{\Psi}_i$ enables continuous gradient propagation, which is essential for optimizing the actors ν_i . We train $\hat{\Psi}_i$ to approach Ψ_i by minimizing the mean squared error at two critical points:

1. $\tilde{\mu}_M(s)$ is the action selected by the current maximizer actor μ_M , having a high Q-value,
2. $\nu_i(s; a_{<i})$ is the action proposed by the i -th actor conditioned on previous actions $a_{<i}$,

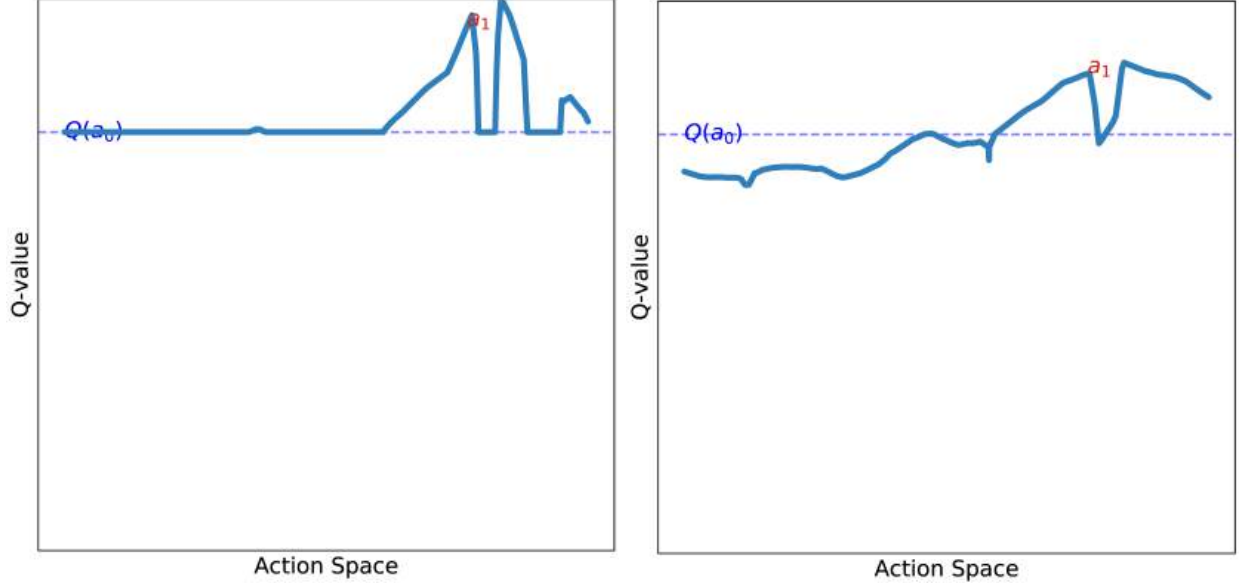


Figure 4.5: In restricted inverted pendulum, given an anchor $Q(a_0)$ value, Ψ (left) has some zero-gradient surfaces which $\hat{\Psi}$ (right) *approximately* follows while allowing non-zero gradients towards high Q-values to flow into its actor ν .

$$\mathcal{L}_{\text{approx}} = \mathbb{E}_{s \sim \rho^{\mu, M}} \left[\sum_{a \in \{\hat{\mu}_M(s), \nu_i(s; a_{<i})\}} \left\| \hat{\Psi}_i(s, a; a_{<i}) - \Psi_i(s, a; a_{<i}) \right\|_2^2 \right]. \quad (4.9)$$

This design ensures $\hat{\Psi}_i$ is updated on high Q-value actions, and thus, the landscape is biased towards those values. This makes the gradient flow trend in the direction of high Q-values. So, even when a_i from ν_i falls in a region of zero gradients for Ψ_i , in $\hat{\Psi}_i$ would provide a policy gradient in a higher Q-value direction if it exists. Figure 4.5 shows Ψ_1 and $\hat{\Psi}_1$ in restricted inverted pendulum.

4.4.5 SAVO-TD3 Algorithm and Design Choices

While the SAVO architecture (Figure 4.4) can be integrated with any off-policy actor-critic algorithm, we choose to implement it with TD3 (Fujimoto et al., 2018) due to its compatibility with continuous and large-discrete action RL (Dulac-Arnold et al., 2015). Using the SAVO actor in TD3 enhances its ability to find better actions in complex Q-function landscapes. Algorithm 3

Algorithm 3 SAVO-TD3

Initialize $Q, Q_2, \mu, \hat{\Psi}_1, \dots, \hat{\Psi}_k, \nu_1, \dots, \nu_k$
Initialize target networks $Q' \leftarrow Q, Q'_2 \leftarrow Q_{\text{twin}}$
Initialize replace buffer \mathcal{B} .
for timestep $t = 1$ to T **do**
 Select Action:
 $a_0 = \mu(s), a_i = \nu_i(s; a_{<i})$ for $i = 1, \dots, k$
 Add perturbations with OU Noise $\hat{a}_i = a_i + \epsilon_i$
 Evaluate $\mu_M(s) = \arg \max_{a \in \{\hat{a}_0, \dots, \hat{a}_k\}} Q(s, a)$
 Exploration action $a = \tilde{\mu}_M(s) = \mu_M(s) + \epsilon$
 Observe reward r and new state s'
 Store $(s, a, \{\hat{a}_i\}_{i=0}^K, r, s')$ in \mathcal{B}
 Update:
 Sample N transitions $(s, a, \{\hat{a}_i\}_{i=0}^K, r, s')$ from \mathcal{B}
 Compute target action $\tilde{a} = \mu_M(s')$
 Update $Q, Q_2 \leftarrow r + \gamma \min\{Q'(s', \tilde{a}), Q'_2(s', \tilde{a})\}$
 Update $\hat{\Psi}_i$ with Eq. 4.9 $\forall i = 1, \dots, k$
 Update actor μ with Eq. 4.3
 Update actor ν_i with Eq. 4.8 $\forall i = 1, \dots, k$
end for

depicts SAVO (highlighted) applied to TD3. We discuss design choices in SAVO and validate them in Section 4.6.

1. **Removing policy smoothing:** We eliminate TD3’s policy smoothing, which adds noise to the target action \tilde{a} during critic updates. In non-convex landscapes, nearby actions may have significantly different Q-values and noise addition might obscure important variations.

2. **Exploration in Additional Actors:** Added actors ν_i explore the surrogate landscapes for high-reward regions by adding OU (Lillicrap et al., 2015) or Gaussian (Fujimoto et al., 2018) noise to their actions.

3. **Twin Critics for Surrogates:** To prevent overestimation bias in surrogates $\hat{\Psi}_i$, we use twin critics to compute the target of each surrogate, mirroring TD3.

4. **Conditioning on Previous Actions:** Actors ν_i and surrogates $\hat{\Psi}_i$ are conditioned on preceding actions via FiLM layers (Perez et al., 2018b) as in Figure 4.4.

5. **Discrete Action Space Tasks:** We apply 1-nearest-neighbor f_{NN} before Q-value evaluation to ensure the Q-function is only queried at in-distribution actions.

SAVO-TD3 employs SAVO actor to systematically reduce the local optima in its base algorithm TD3. We empirically validate the proposed design improvements.

4.5 Environments

We evaluate SAVO on discrete and continuous action space environments with challenging Q-value landscapes. More environment details are presented in Appendix C and Figure C.1.

Locomotion in Mujoco. We evaluate on MuJoCo (Todorov et al., 2012) environments of Hopper-v4, Walker2D-v4, Inverted Pendulum-v4, and Inverted Double Pendulum-v4.

Locomotion in Restricted Mujoco. We create a restricted locomotion suite of the same environments as in MuJoCo. A *hard* Q-landscape is realized via high-dimensional discontinuities that restrict the action space. Concretely, a set of predefined hyper-spheres (as shown in Figure 4.6) in the action space are sampled and set to be valid actions, while the other invalid actions have a null effect if selected. The complete details can be found in Appendix C.3.1.

Adroit Dexterous Manipulation. Rajeswaran et al. (2017) propose manipulation tasks with a dexterous multi-fingered hand.

Door: In this task, a robotic hand is required to open a door with a latch. The challenge lies in the precise manipulation needed to unlatch and swing open the door using the fingers. *Hammer:* the robotic hand must use a hammer to drive a nail into a board. This task tests the hand’s ability to grasp the hammer correctly and apply force accurately to achieve the goal. *Pen:* This task involves the robotic hand manipulating a pen to reach a specific goal position and rotation. The objective

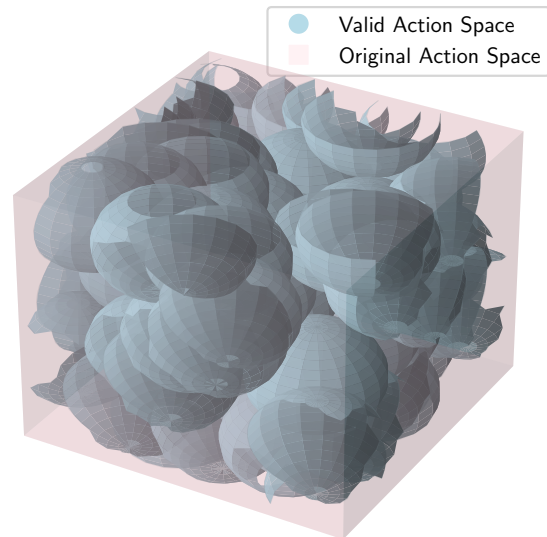


Figure 4.6: Restricted Hopper’s 3D visualization of Action Space.

is to control the pen’s orientation and position using fingers, which demands fine motor skills and coordination.

Mining Expedition in Grid World. We develop a 2D Mining grid world environment (Chevalier-Boisvert et al., 2018) where the agent (Figure C.1) navigates a 2D maze to reach the goal, removing mines with correct pick-axe tools to reach the goal in the shortest path. The action space includes navigation and tool-choice actions, with a procedurally-defined action representation space. The Q-landscape is non-convex because of the diverse effects of nearby action representations.

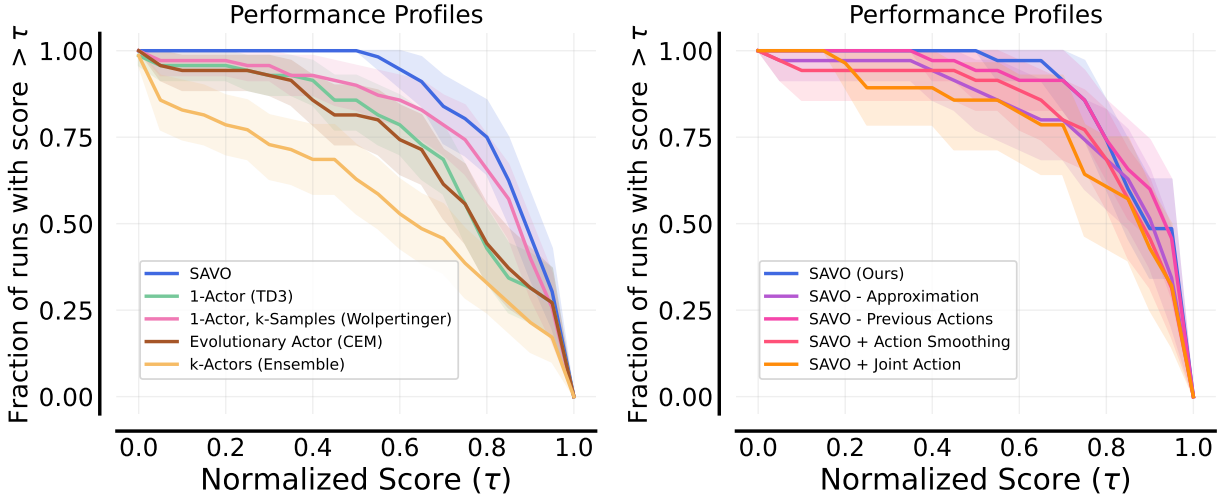
Simulated and Real-Data Recommender Systems. RecSim (Ie et al., 2019a) simulates sequential user interactions in a recommender system with a large discrete action space. The agent must recommend the most relevant item from 10,000 items based on user preference information. The action representations are simulated item characteristic vectors in simulated and movie review embeddings in the real-data task based on MovieLens (Harper and Konstan, 2015) for items.

4.6 Experiments

4.6.1 Effectiveness of SAVO in challenging Q-landscapes

We compare SAVO against the following baseline actor architectures, where $k = 3$:

- **1-Actor (TD3):** Conventional single actor architecture which is susceptible to local optima.
- **1-Actor, k samples (Wolpertinger):** Gaussian sampling centered on actor’s output. For discrete actions, we select k -NN discrete actions around the continuous action (Dulac-Arnold et al., 2015).
- **k -Actors (Ensemble):** Best Q-value among actions from ensemble (Osband et al., 2016).
- **Evolutionary actor (CEM):** CEM search over the action space (Kalashnikov et al., 2018).
- **Greedy-AC:** Greedy Actor Critic (Neumann et al., 2018) trains a high-entropy proposal policy and primary actor trained from best proposals with gradient updates.
- **Greedy TD3:** Our version of Greedy-AC with TD3 exploration and update improvements.



(a) SAVO versus baseline actor architectures.

(b) SAVO versus ablations of SAVO

Figure 4.7: Aggregate performance profiles using normalized scores over 7 tasks and 10 seeds each.

- **SAVO:** Our method with k successive actors and surrogate Q-landscapes.

We ablate the crucial components and design decisions in SAVO:

- **SAVO - Approximation:** removes the approximate surrogates (Section 4.4.4), using Ψ_i instead of $\hat{\Psi}_i$.
- **SAVO - Previous Actions:** removes conditioning on $a_{<i}$ in SAVO’s actors and surrogates.
- **SAVO + Action Smoothing:** TD3’s policy smoothing (Fujimoto et al., 2018) is used to compute Q-targets.
- **SAVO + Joint Action:** trains an actor with a joint action space of $3 \times D$. The k action samples are obtained by splitting the joint action into D dimensions. Validates successive nature of SAVO.

Aggregate performance. We use performance profiles (Agarwal et al., 2021) to aggregate results across different environments in Figure 4.7a (evaluation mechanism detailed in Appendix G.1). SAVO consistently outperforms baseline actor architectures like single-actor (TD3) and sampling-augmented actor (Wolpertinger), showing wide robustness across challenging Q-landscapes. In Figure 4.7b, SAVO outperforms its ablations, validating each component and design decision.

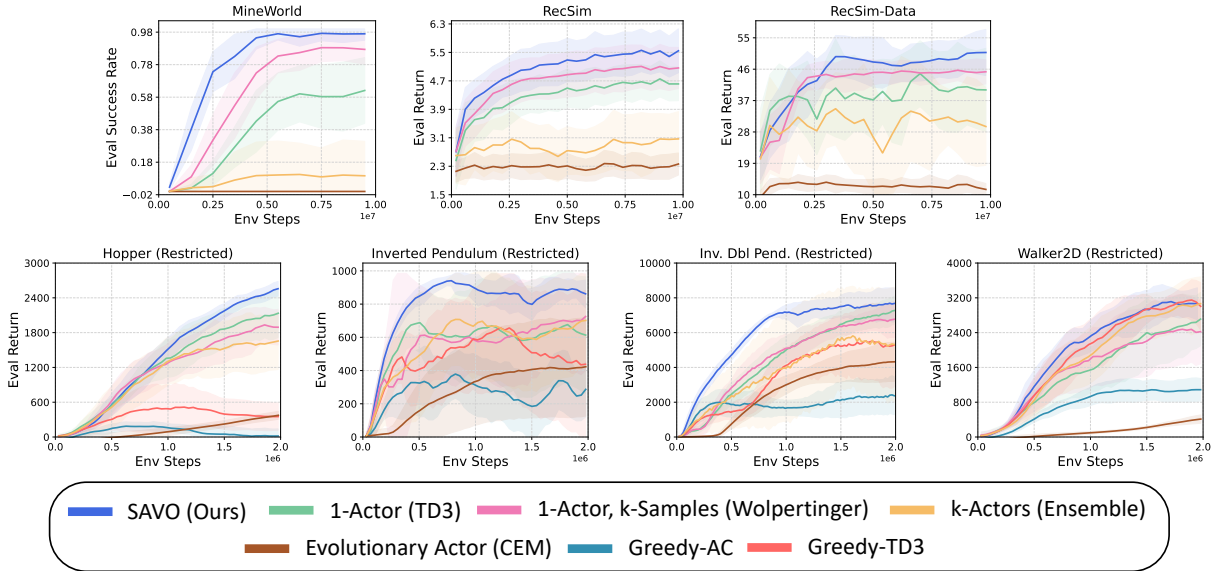


Figure 4.8: SAVO against baselines on discrete and continuous tasks. Results over 10 seeds.

Per-environment results. In discrete action tasks, the Q-value landscape is only well-defined at exact action representations and nearby discrete actions might have very different values (Section 4.3.2). This makes the Q-value landscape uneven, with multiple peaks and valleys (Figure 4.2). For example, actions in Mining Expedition involve both navigation and tool-selection which are quite different, while RecSim and RecSim-Data have many diverse items to choose from. Methods like [Wolpertinger](#) that sample many actions a local neighborhood perform better than [TD3](#) which considers a single action (Figure 4.8). However, SAVO achieves the best performance by directly simplifying the non-convex Q-landscape. In restricted locomotion, there are several good actions that are far apart. SAVO actors can search and explore widely to optimize the Q-landscape better than only nearby sampled actions. Figure C.4 ablates SAVO in all 7 tasks and shows that the most critical features are its successive nature, removing policy smoothing, and approximate surrogates.

4.6.2 Q-Landscape Analysis: Do successive surrogates reduce local optima?

Figure 4.9 visualizes surrogate landscapes in Inverted-Pendulum-Restricted for a given state s . Successive pruning and approximation smooth the Q-landscapes, reducing local optima. A single actor gets stuck at a local optimum a_0 (left), but surrogate $\hat{\Psi}_1$ uses a_0 as an anchor to find a better

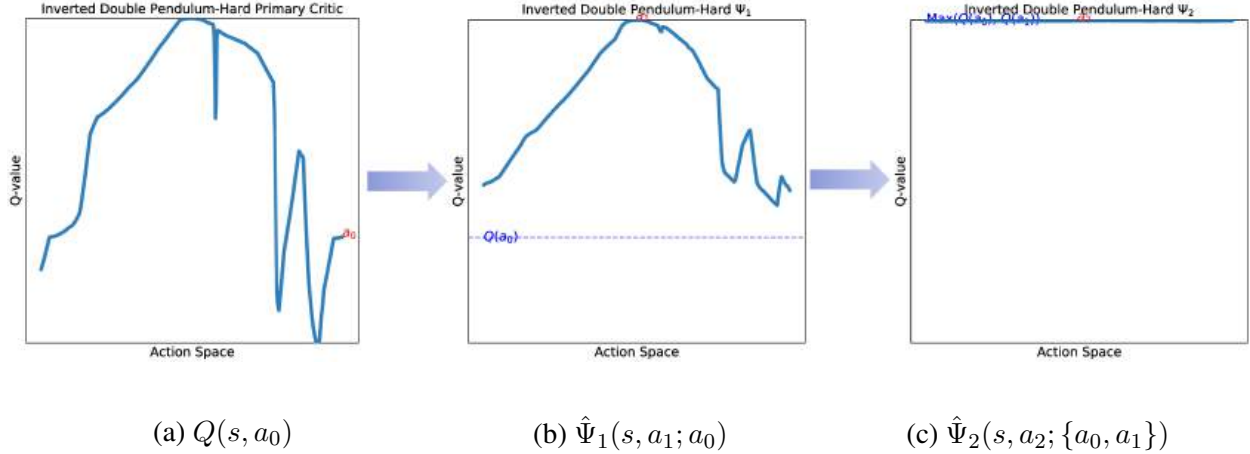


Figure 4.9: SAVO helps a single actor escape the local optimum a_0 in the Restricted Inverted Pendulum Task. Each successive surrogate learns a Q-landscape with fewer local optima and thus is easier to optimize by its actor.

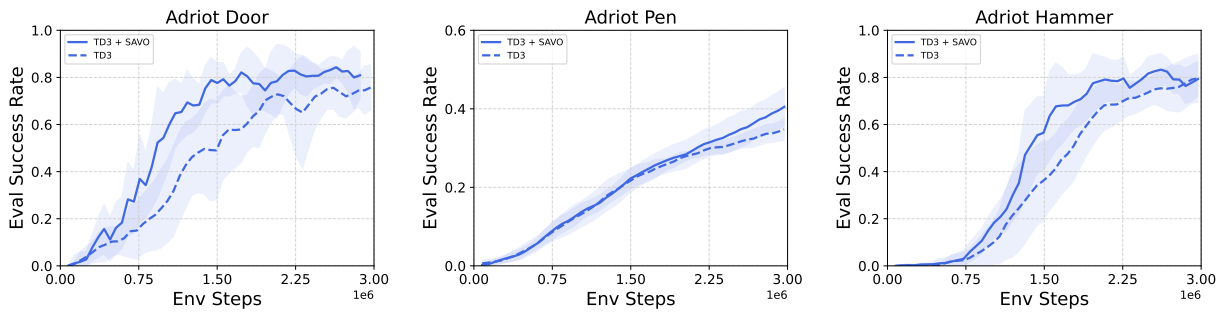


Figure 4.10: SAVO improves the sample-efficiency of TD3 on Adroit dexterous manipulation tasks. optimum a_1 . The maximizer policy finally selects the highest Q-valued action among a_0, a_1, a_2 . Figure C.12 extends this visualization to Inverted-Double-Pendulum-Restricted. Figure C.11 shows how one actor is sufficient in the *convex* Q-landscape of unrestricted Inverted-Pendulum-v4. Figures C.13, C.14 show how Hopper-v4 Q-landscape provides a path to global optimum, while Hopper-Restricted is non-convex.

4.6.3 Challenging Dexterous Manipulation (Adroit)

In Adroit dexterous manipulation tasks (Door, Pen, Hammer) (Rajeswaran et al., 2017), SAVO improves sample efficiency of TD3 (Figure 4.10). The non-convexity in Q-landscape likely arises from nearby actions having high variance outcomes like grasping, missing, dropping, or no impact.

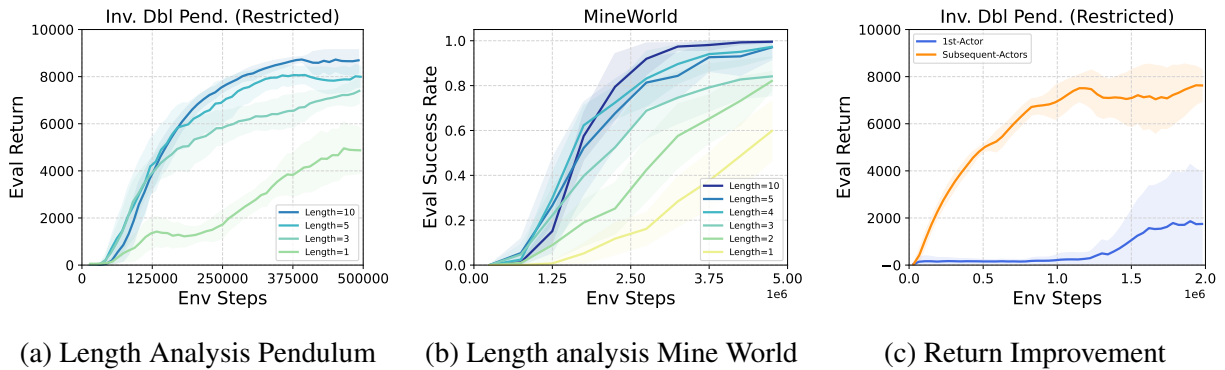


Figure 4.11: (L) More successive actor-surrogates are better, (R) SAVO v/s single-actor on inference.

4.6.4 Quantitative Analysis: The Effect of Successive Actors and Surrogates

We investigate the effect of increasing the number of successive actor-surrogates in SAVO in Pendulum (Figure 4.11a) and MineWorld (Figure 4.11b). Additional actor-surrogates significantly help to reduce severe local optima initially. However, the improvement saturates as the suboptimality gap reduces.

Next, we show that successive actors are needed because a single actor can get stuck in local optima even with an optimal Q-function. In Figure 4.11c, we consider a SAVO agent trained to optimality with 3 actors. When we remove the additional actors, the remaining single-actor agent resembles TD3 trained to maximize an “optimal” Q-function. However, the significant performance gap indicates that the single actor could not find optimal actions for the given Q-function.

4.6.5 Does RL with Resets address the issue of Q-function Optimization?

Primacy bias (Nikishin et al., 2022; Kim et al., 2024) occurs when an agent is trapped in suboptimal behaviors from early training. To mitigate this, methods like resetting parameters and re-learning from the replay buffer aim to reduce reliance on initial samples. We run TD3 in MineEnv with either a full-reset or last-layer reset every 200k, 500k, or 1 million iterations. None of these versions outperformed the original TD3 algorithm without resets. This is because resetting does not help an actor to navigate the Q-landscape better and can even cause an otherwise optimal actor to get stuck in a suboptimal solution during retraining. In contrast, the SAVO actor architecture

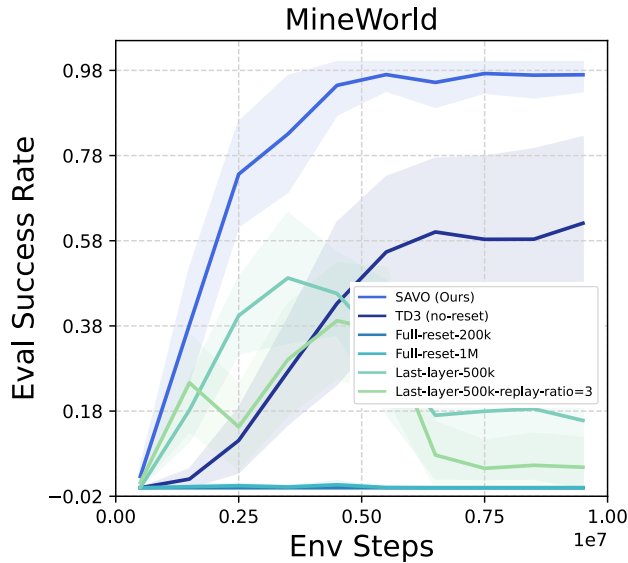


Figure 4.12: Reset (primacy bias) does not improve Q-optimization.

specifically addresses the non-convex Q-landscapes, being a more robust method to finding closer to optimal actions.

4.6.6 Further experiments to validate SAVO

- **Unrestricted locomotion.** Figure C.3 shows that both SAVO and baselines achieve optimal performance in simple Q-landscapes, confirming effective hyperparameter tuning (Section G.4, Section G.3) and indicating that the baselines underperform due to the complexity introduced in Q-landscapes.
- **SAVO orthogonal to SAC.** Figure C.5 shows that SAVO+TD3 ζ SAC ζ TD3, indicating that SAC’s stochastic policy does not address non-convexity, but can itself suffer from local optima (Figure C.6)
- **Design Choices.** Figure C.8 shows that LSTM, DeepSet, and Transformers are all valid choices as summarizers of preceding actions $a_{<i}$ in SAVO. Figure C.9 shows that FiLM conditioning on $a_{<i}$ especially helps for discrete action space tasks but has a smaller effect in continuous action space. In Figure C.10a, we find Ornstein-Uhlenbeck (OU) noise and Gaussian noise to be largely equivalent.

Method	GPU Mem.	Return	Time
TD3	619MB	1107.795	0.062s
SAVO k=3	640MB	2927.149	0.088s
SAVO k=5	681MB	3517.319	0.122s

Table 4.1: Compute v/s Performance Gain (Mujoco)

- **Massive Discrete Actions.** SAVO outperforms in RecSim with 100k and 500k actions (Figure C.7).

4.7 Limitations and Conclusion

Introducing more actors in SAVO has negligible influence on GPU memory, but leads to longer inference time (Table 4.1). However, even for 3 actor-surrogates, SAVO achieves significant improvements in all our experiments. Further, for tasks with a simple convex Q-landscape, a single actor does not get stuck in local optima, making the gain from SAVO negligible.

In this chapter, we improve Q-landscape optimization in deterministic policy gradient RL with Successive Actors for Value Optimization (SAVO) in both continuous and large discrete action spaces. We demonstrate with quantitative and qualitative analyses how the improved optimization of Q-landscape with SAVO leads to better sample efficiency and performance.

Chapter 5

Sharing Actions in Multi-Task Reinforcement Learning via Q-switch Mixture of Policies

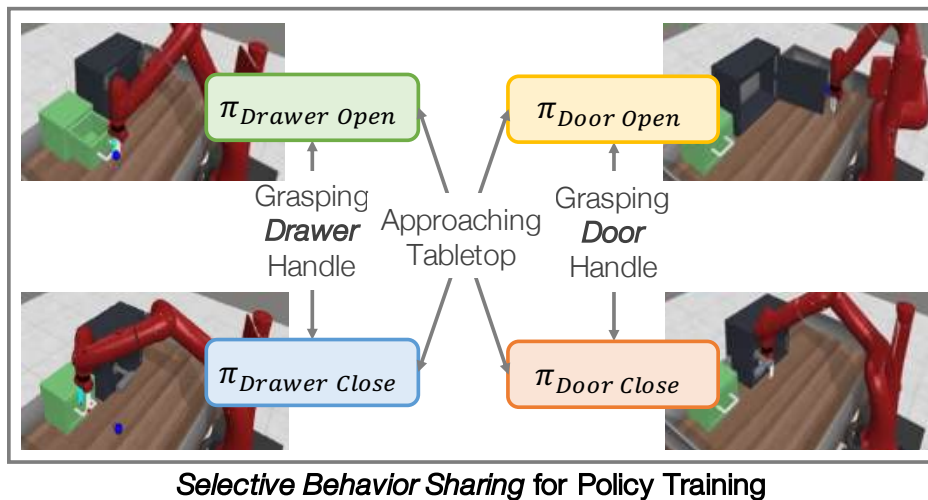


Figure 5.1: We propose a sample-efficient MTRL framework that selectively shares behaviors by acting with other task policies for data collection. For example, Drawer Open and Drawer Close can share behaviors for grasping the drawer handle, while Drawer Open and Door Close share behaviors for approaching the tabletop.

5.1 Introduction

In multi-task reinforcement learning, each task has something to learn from the others. Consider a task set where a robot is simultaneously learning to open and close a drawer and a door on a tabletop, as illustrated in Figure 5.1. These tasks share many similar behaviors, like approaching

the tabletop or grasping the object handle. While learning to open a drawer, the robot can benefit by exploring behaviors found rewarding in other related tasks (such as grasping the drawer or door handle), then incorporating the helpful behaviors into its own policy, instead of randomly exploring the entire action space. Can we craft a framework to learn from similar behaviors across tasks to accelerate overall learning?

Most multi-task reinforcement learning (MTRL) methods share task information via policy parameters (Vithayathil Varghese and Mahmoud, 2020) or data relabeling (Kaelbling, 1993). We propose a new framework for MTRL: *share behaviors* between tasks to improve data collection by employing useful policies from other tasks for more informative training data. This approach offers a simple, general, and sample-efficient approach that complements existing MTRL methods.

Prior works (Teh et al., 2017; Ghosh et al., 2018) share behaviors between task policies uniformly by regularizing to one shared distilled policy (Rusu et al., 2015). This introduces a bias towards the mean behavior and causes negative interference when tasks might require differing optimal behaviors from the same state. In contrast, reusing other policies for data collection does not introduce any bias.

We propose *selective behavioral policy sharing* as a novel and general mechanism to improve sample efficiency in any MTRL architecture. Our key insight is that behaviors being acquired in other tasks can help when appropriately selected and shared, as shown in human learners (Tomov et al., 2021). In the Drawer Open task, while learning to approach the drawer handle, the robot should share behaviors between the Drawer policies, but avoid Door policies which would lead it to the wrong object.

The key question with selective behavioral policy sharing is how to identify helpful behaviors from other policies in a principled way. We propose a principled way of selecting shared behaviors: a Q-switch Mixture of Policies (QMP). At each state, one policy from a mixture of all policies is selected to collect data. The Q-switch makes this selection based on which policy best optimizes the current task’s soft Q-value because that is an estimate of the most helpful behavior for the current task. We prove that this selection mechanism preserves the convergence guarantees of

the underlying RL algorithm and potentially improves sample efficiency. Crucially, QMP uses other tasks’ policies only for data collection, allowing policy training to remain unbiased under any off-policy RL algorithm.

Our primary contribution is introducing behavioral policy sharing for MTRL as a novel avenue of information sharing between tasks and addressing the problem of principled selective behavior sharing. Our proposed framework, Q-switch Mixture of Policies (QMP), can effectively identify shareable behaviors between tasks and incorporates them to gather more informative training data. We prove that QMP’s behavior sharing not only preserves the policy convergence of the underlying RL algorithm, but is at least as sample efficient. We demonstrate that QMP provides complementary gains to other forms of MTRL in a range of manipulation, locomotion, and navigation tasks and performs well over diverse task families when compared to other behavior sharing methods.

5.2 Related Work

Information Sharing in Multi-Task RL. There are multiple, mostly complementary ways to share information in MTRL, including sharing *data*, sharing *parameters* or *representations*, and sharing *behaviors*. In offline MTRL, prior works selectively share *data* between tasks (Yu et al., 2021; 2022). Sharing parameters across policies can speed up MTRL through shared *representations* (Xu et al., 2020; D’Eramo et al., 2020; Yang et al., 2020; Sodhani et al., 2021; Misra et al., 2016; Perez et al., 2018a; Devin et al., 2017; Vuorio et al., 2019; Rosenbaum et al., 2019; Yu et al., 2023; Cheng et al., 2023; Hong et al., 2022) and can be easily combined with other types of information sharing. Most similar to our work, Teh et al. (2017) and Ghosh et al. (2018) share *behaviors* between multiple policies through policy distillation and regularization. Vuong et al. (2019) identify which states between tasks share optimal behavior and regularize to each other there. These works share behaviors through regularization, biasing the policy objective when tasks have differing optimal behaviors. In contrast, our work selectively shares behavioral policies without modifying the training objective.

Multi-Task Learning for Diverse Task Families. Multi-task learning in diverse task families is susceptible to *negative transfer* between dissimilar tasks, hindering training. Prior works combat this by measuring task relatedness through validation loss on tasks (Liu et al., 2022; Ackermann et al., 2021) or influence of one task to another (Fifty et al., 2021; Standley et al., 2020) to find task groupings for training. Other works focus on the challenge of multi-objective optimization (Sener and Koltun, 2018; Hessel et al., 2019; Yu et al., 2020; Liu et al., 2021a; Schaul et al., 2019; Chen et al., 2018; Kurin et al., 2022). Similar to these works, we identify that prior behavior-sharing MTRL approaches are susceptible to negative transfer. However, we avoid the challenge of negative transfer entirely by selectively sharing behaviors only during off-policy data collection.

Exploration in Multi-Task Reinforcement Learning. Our approach of modifying the behavioral policy to leverage shared task structures can be seen as a form of MTRL exploration, which we discuss further in Appendix Section D.16c. Bangaru et al. (2016) encourage agents to increase their state coverage by providing an exploration bonus. Zhang and Wang (2021) study sharing information between agents to encourage exploration under tabular MDPs. Kalashnikov et al. (2021b) directly leverage data from policies of other specialized tasks (like grasping a ball) for their general task variant (like grasping an object).

In contrast to these approaches, we do not require a pre-defined task similarity measure or exploration bonus; we demonstrate in Section 5.6 that QMP works across many tasks and domains without these additional measures. Skill learning can be seen as behavior sharing in a single task setting such as learning options for exploration or heirarchical RL (Machado et al., 2017; Jinnai et al., 2019b;a; Hansen et al., 2019; Riemer et al., 2018). We also discuss the difference to single-task exploration in Appendix Section G.3.

Using Q-functions as filters. Yu et al. (2021) uses Q-functions to filter which data should be shared between tasks in a multi-task setting. In the imitation learning setting, Nair et al. (2018a) and Sasaki and Yamashina (2020) use Q-functions to filter out low-quality demonstrations, so they are not used for training. In both cases, the Q-function is used to evaluate some data that can be used for training. Zhang et al. (2022) reuses pre-trained policies to learn a new task, using a Q-function as a

filter to choose which pre-trained policies to regularize to as guidance. In contrast to prior works, our method uses a Q-function to *evaluate* different task policies to gather training data.

5.3 Problem Formulation

Multi-task reinforcement learning (MTRL) addresses sequential decision-making tasks, where an agent learns a policy to act optimally in an environment (Kaelbling et al., 1996; Wilson et al., 2007). Therefore, in addition to typical multi-task learning techniques, MTRL can also share *behaviors*, i.e., actions, to improve sample efficiency. However, current approaches share behaviors uniformly (Section 5.2), which assumes that different tasks’ behaviors do not conflict. To address this limitation, we seek to develop a selective behavior-sharing method that can be applied in more general task families for sample-efficient MTRL.

Multi-Task RL with Behavior Sharing. We aim to simultaneously learn a set $\{\mathbb{T}_1, \dots, \mathbb{T}_N\}$ of N tasks. Each task \mathbb{T}_i is a Markov Decision Process (MDP) defined by state space \mathcal{S} , action space \mathcal{A} , transition probabilities \mathcal{T}_i , reward functions \mathcal{R}_i , initial state distribution ρ_i , and discount factor $\gamma \in [0, 1]$. While we use \mathcal{S} to denote shared state spaces for simplicity, our formulation extends to tasks with different state spaces as it complements policy architectures that share state encoders. The agent learns a set of N policies $\{\pi_1, \dots, \pi_N\}$, where each policy $\pi_i(a|s)$ represents the behavior on task \mathbb{T}_i . The objective is to maximize the average expected return over all tasks,

$$\{\pi_1^*, \dots, \pi_N^*\} = \max_{\{\pi_1, \dots, \pi_N\}} \frac{1}{N} \sum_{i=1}^N \left[\mathbb{E}_{a_t \sim \pi_i} \sum_{t=0}^{\infty} \gamma^t \mathcal{R}_i(s_t, a_t) \right].$$

Unlike prior works, our tasks can exhibit conflicting optimal behaviors: for any s , $\pi_i^*(a|s)$ may differ from $\pi_j^*(a|s)$. Thus, prior methods that bias policy learning objectives like direct policy sharing (Kalashnikov et al., 2021a) or behavior regularization (Teh et al., 2017) would be suboptimal.

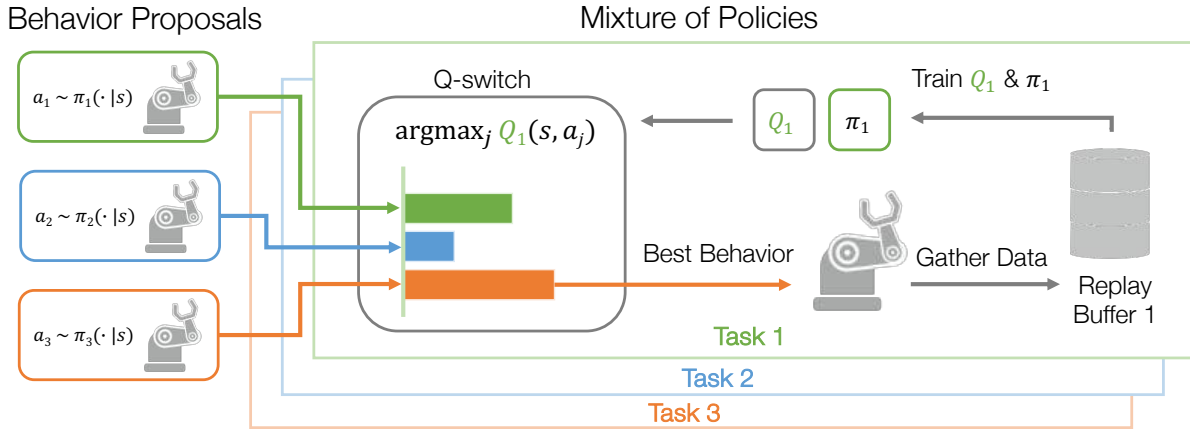


Figure 5.2: Our method (QMP) shares behavior between task policies in the data collection phase using a mixture of these policies. For example, in Task 1, each task policy proposes an action a_j . The task-specific Q-switch evaluates each $Q_1(s, a_j)$ and selects the best scored policy to gather reward-labeled data to train Q_1 and π_1 . Thus, Task 1 will be boosted by incorporating high-reward shareable behaviors into π_1 and improving Q_1 for subsequent Q-switch evaluations.

5.4 Approach

To improve the sample efficiency of multi-task RL, we propose a framework that *selectively* incorporates behaviors from policies of other tasks without introducing bias into the RL objective for the current task. We achieve this by using a mixture of all policies as the behavioral policy for the current task, thereby modifying only its *off-policy training data*. However, naively mixing other policies into the current task’s behavioral policy does not necessarily improve its sample efficiency. To address this, we derive a specific definition of this mixture, named Q-switch Mixture of Policies (QMP), that selects a policy based on the current task’s Q-function (see Figure 5.2 and Algorithm 4) and prove that QMP guarantees greater than or equal sample efficiency than using the current task’s policy alone.

5.4.1 Multi-Task Behavior Sharing via Off-Policy Data Collection

MTRL methods like Teh et al. (2017) use *regularization to a common average policy* to enforce task policies to share behaviors. However, this introduces bias to each policy’s RL objective, leading to suboptimal actions in states where tasks require different actions. To address this, we propose

using a *mixture of policies for off-policy data collection* as the means of behavior-sharing. At each state in any given task, one of the task policies is selected to gather training data as the current behavioral policy. This approach is compatible with any off-policy RL algorithm (Watkins and Dayan, 1992) because the environment rewards help determine the best actions from the collected data. However, an effective mixture policy must choose the behavioral policies in a selective and principled way.

Definition 5.4.1 (Mixture of Policies). For each task \mathbb{T}_i , the mixture policy $\pi_i^{\text{mix}}(a | s)$ is defined as $\pi_i^{\text{mix}}(a | s) = \pi_{f_i(s, \pi_1, \dots, \pi_N)}(a | s)$, where $f_i(s, \pi_1, \dots, \pi_N) : \mathcal{S} \times \Pi^N \rightarrow \{1, \dots, N\}$ is a mixture-switch function that selects one of the policies π_1, \dots, π_N based on the current state s .

Our intuition of policy mixture shares inspiration with hierarchical RL (Çelik et al., 2021; Daniel et al., 2016; End et al., 2017; Goyal et al., 2019) where a *mixture* of options is learned according to the downstream task(s). However, a key difference in an MTRL mixture is that each policy is optimized for its own specific task and not designed to fit the task where the mixture is employed.

5.4.2 Q-switch Mixture of Policies (QMP)

We aim to derive a principled mixture-switch function f_i such that the mixture policy π_i^{mix} selectively incorporates behaviors from other policies while being guaranteed to be no worse than the current task’s policy π_i . We recall the generalized policy iteration procedure (Sutton and Barto, 2018) underlying single-task SAC (Haarnoja et al., 2018): policy evaluation learns Q by minimizing the bellman error on the collected data, and policy improvement follows Q by minimizing the KL divergence between the new policy and the exponential of the current Q -function, $Q^{\pi^{\text{old}}}$:

$$\pi^{\text{new}} = \arg \min_{\pi' \in \Pi} D_{\text{KL}} \left(\pi'(\cdot | s_t) \left\| \frac{\exp \left(\frac{1}{\alpha} Q^{\pi^{\text{old}}}(s_t, \cdot) \right)}{Z^{\pi^{\text{old}}}(s_t)} \right. \right) \quad (5.1)$$

In practice, the gradient updates in SAC are gradual and do not instantly achieve this optimization in Eq. 5.1, leaving a suboptimality gap to catch up to the Q -function. Thus, a mixture policy π_i^{mix} that selects the best policy from a set of all given policy candidates, *including the current policy*,

Algorithm 4 Q-switch Mixture of Policies (QMP)

Input: Task Set $\{\mathbb{T}_1, \dots, \mathbb{T}_N\}$
Initialize $\{\pi_i\}_{i=1}^N, \{Q_i\}_{i=1}^N$, Data buffers $\{\mathcal{D}_i\}_{i=1}^N$
for each epoch **do**
 for $i = 1$ to N **do**
 while Task \mathbb{T}_i episode not terminated **do**
 Observe state s
 Compute π_i^{mix} as in Eq. 5.3.
 Take action proposal from $a \sim \pi_i^{\text{mix}}$
 $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup (s, a, r_i, s')$
 end while
 end for
 for $i = 1$ to N **do**
 Update π_i, Q_i using \mathcal{D}_i with SAC
 end for
end for
Output: Trained policies $\{\pi_i\}_{i=1}^N$

ensures that π_i^{mix} is at least as good as π_i for the current state s , while potentially being a better optimizer of Eq. 5.1 due to shareable behaviors from the other task policies:

$$\min_{\pi' \in \{\pi_1, \dots, \pi_N\}} D_{\text{KL}} \left(\pi'(\cdot | s) \left\| \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(s, \cdot))}{Z^{\pi_i}(s)} \right\| \right) \leq D_{\text{KL}} \left(\pi_i(\cdot | s_t) \left\| \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(s, \cdot))}{Z^{\pi_i}(s)} \right\| \right) \quad (5.2)$$

We simplify the expression on the left in Appendix B, deriving the following definition.

Definition 5.4.2 (Q-switch Mixture of Policies: QMP). For a task \mathbb{T}_i and available candidate policies $\{\pi_1, \dots, \pi_N\}$, the QMP $\pi_i^{\text{mix}}(a | s)$ selects a policy according to:

$$\pi_i^{\text{mix}} = \arg \max_{\pi' \in \{\pi_1, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} [Q^{\pi_i}(s, a)] + \alpha \mathcal{H} [\pi'(\cdot | s)] \quad (5.3)$$

Algorithm 4 shows that QMP can be plugged into any MTRL framework, making it complementary with various MTRL frameworks like parameter-sharing and data relabeling (see Section 5.7.1). In practice, we estimate the expectation in Eq. 5.3 by evaluating the Q-value for the mean action of each task policy’s distribution $\pi'(\cdot | s)$ ignoring the entropy term. We do not find any empirical difference when using a sampled estimate of the expectation (see Appendix G.2) or including the entropy term, as the Q-value is the primary distinguishing factor between policies. In terms of

compute, sampling from QMP’s $\pi_i^{\text{mix}}(a|s)$ does require more policy and Q-function evaluations. However, evaluations are parallelized and impact runtime negligibly, as shown in Appendix G.4.

While π_i^{mix} can mistakenly choose a poor policy due to estimation error in Q^{π_i} , this is identical to Q-learning or SAC, where the Q-function would be inaccurate at less-seen states. In both Q-learning and QMP, this is corrected with online interactions where the Q-function is trained to be more accurate in a subsequent iteration. Furthermore, π_i^{mix} actually better maximizes Q^{π_i} than π_i , which is the objective under generalized policy iteration. Note that QMP does **not** exacerbate the problem of overestimation because the soft policy evaluation step stays the same, i.e., it uses π_i and not π_i^{mix} .

5.5 Why QMP Works: Theory and Didactic Example

5.5.1 QMP: Convergence and Improvement Guarantees

QMP performs simultaneous MTRL by collecting data using a Q-switch guided mixture of policies π_i^{mix} . In Appendix C, we prove that QMP with underlying RL algorithm Soft-Actor Critic (SAC) (Haarnoja et al., 2018) shares the same convergence guarantees in a tabular setting. Particularly, we show that under QMP, policy evaluation converges because QMP only modifies data collection of off-policy RL, policy improvement guarantees are preserved (Theorem 5.5.1), and policy iteration converges to an optimal policy at least as sample-efficiently (Theorem C.2).

The key reason for *better policy improvement* of QMP over the current task policy π_i is the $\arg \max$ operation in Eq. 5.3, which ensures that the selected policy $\pi_i^{\text{mix}} \in \{\pi_j\}_{j=1}^N$ optimizes the SAC objective at least as well as π_i itself. We formalize this in Theorem 5.5.1 with proof in Appendix C.1. Due to the suboptimality gap in Eq. 5.1 in SAC, QMP can actually achieve better policy improvement when there are shareable behaviors between policies.

Theorem 5.5.1 (Mixture Soft Policy Improvement). *Consider π_i^{old} and its associated Q-function Q_i . Apply SAC’s policy improvement $\pi_i^{\text{old}} \rightarrow \pi_i$ and then $\pi_i \rightarrow \pi_i^{\text{mix}}$ from Eq. 5.3. Then, $Q^{\pi_i^{\text{mix}}}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_i}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t)$ for all tasks \mathbb{T}_i and for all $(s_t, a_t) \in \mathcal{S} \times \mathcal{A}$ with $|\mathcal{A}| < \infty$.*

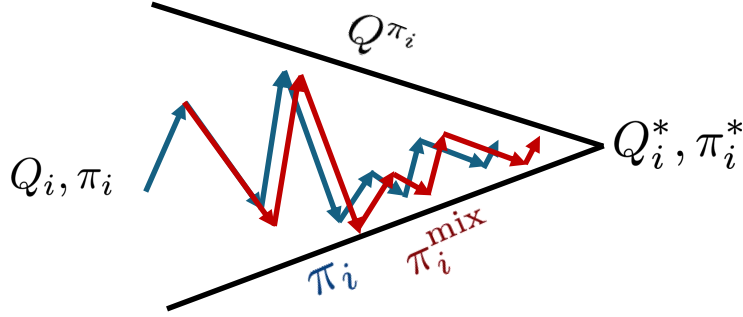


Figure 5.3: QMP generalized policy iteration

While QMP in Def. 5.4.2 applies to any set of candidate policies $\{\pi_1, \dots, \pi_N\}$, one expects π_i^{mix} to improve over π_i when some $\pi_j \neq \pi_i$ proposes an action candidate better than π_i for Task \mathbb{T}_i . This is more likely in MTRL policies that *share structure between tasks* than an arbitrary set of policies. For instance, if \mathbb{T}_i and \mathbb{T}_j share a subtask that appears early in the episodes for \mathbb{T}_j , then π_j would have already learned it before π_i and be a better policy for certain states of \mathbb{T}_i , according to Q_i .

QMP making bigger policy improvement results in each iteration of generalized policy iteration progressing more towards optimality. This reduces the number of iterations required to converge, improving the sample efficiency of the algorithm as illustrated in Fig. 5.3 and proved in Theorem C.2.

5.5.2 Illustrative Example: 2D Point Reaching

We demonstrate when QMP can *utilize alternate policy candidates* $\{\pi_1, \dots, \pi_N\}$ to more effectively learn a policy by bridging a *policy improvement suboptimality gap* as π tries to follow Q in Eq. 5.1. Consider a 2D point-reaching task where the agent must navigate from the bottom-left corner $(0, 0)$ to the goal in the top-right corner $(10, 10)$. The point agent receives dense rewards based on its proximity to the goal and takes incremental 2D actions $(\Delta x, \Delta y) \in [-1, 1]^2$.

Figure 5.5 shows that the SAC policy π converges to a suboptimal solution. Fig. 5.4a confirms that the data collected by SAC policy never reaches the goal. This shows that if the suboptimality gap in π is not successfully bridged, it can make the entire algorithm converge suboptimally.

To illustrate QMP’s effect, we add 3 fixed gaussian policies centered on $(\uparrow \rightarrow \swarrow)$ or $(\uparrow \rightarrow \nearrow)$, and only let π be trainable. Fig. 5.4b, 5.4c show that π_i^{mix} employs alternate policies at many states

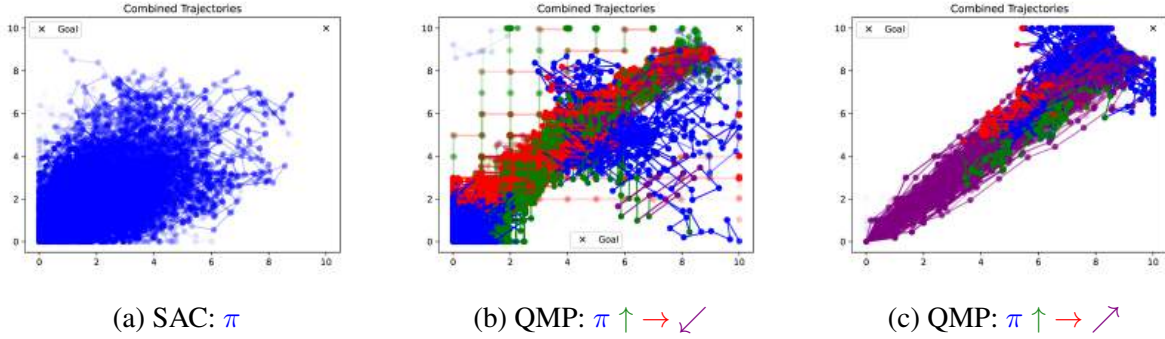


Figure 5.4: **2D Point Reaching.** We visualize the training trajectories of π with different sets of task policies (fixed but stochastic) and color each step with the policy that proposed it. **(a)** The single-task SAC policy cannot reach the goal. **(b)** With 3 diverse policies ($\uparrow \rightarrow \searrow$), QMP often selects other policies, showing the suboptimality gap from Q in Eq. 5.1. **(c)** When a highly relevant \nearrow policy is added, QMP often selects \nearrow as it is likely to best optimize the learned Q -function.

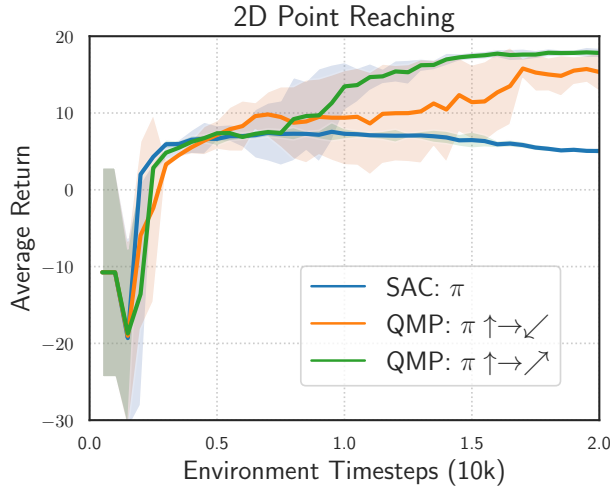


Figure 5.5: QMP improves performance using other policies, increasingly so when task-relevant.

in data collection as they optimize Eq. 5.3 better than π itself. This *selectivity* enables π_i^{mix} to generate more effective goal-reaching trajectories by bridging the suboptimality gap, resulting in better performance in Fig. 5.5. A policy like \nearrow relevant to the underlying task leads to a larger gain.

The same principle extends to the simultaneous multi-task RL setting. In MTRL, each task’s policy continuously improves and can serve as a valuable candidate in the mixture for other tasks. QMP enables tasks to selectively share their behaviors, allowing each task to benefit from the progress of others. This mutual assistance accelerates learning across all tasks, as the mixture policy π_i^{mix} for each task \mathbb{T}_i selects the most promising action proposals from all available policies

according to the task-specific Q-function, guaranteed to be at least as good as π_i itself. Consequently, MTRL combined with QMP leverages the collective knowledge of all tasks to bridge suboptimality gaps more efficiently, leading to improved sample efficiency and overall performance.

5.6 Experiments

5.6.1 Environments

We evaluate our method in 7 multi-task designs in manipulation, navigation, and locomotion environments, shown in Figure 5.6. These tasks vary in the degree of shared and conflicting behaviors between tasks and the number of tasks in the set. Further details in Appendix Section D.

Multistage Reacher: A 6 DoF Jaco arm learns 5 tasks with ordered subgoals. The first 4 tasks share some subgoals, while the 5th *conflicting* task requires the agent to stay at its initial position.

Maze Navigation: Building on point mass maze navigation (Fu et al., 2020), we define 10 tasks with various start-goal locations exhibiting coinciding and conflicting segments in the optimal paths.

Meta-World Manipulation: We use three task sets based on the Meta-World environment (Yu et al., 2019). **Meta-World MT10** and **Meta-World MT50** are sets of 10 and 50 table-top manipulation tasks involving different objects and behaviors. **Meta-World CDS** is a 4-task set proposed in Yu et al. (2021), which places the door and drawer next to each other on the same tabletop so all 4 tasks (door open & close, drawer open & close) are solvable in a simultaneous multi-task setup.

Walker2D: Walker2D is a 9 DoF bipedal walker agent with the multi-task set containing 4 locomotion tasks proposed in Lee et al. (2019): walking forward, walking backward, balancing, and crawling. These tasks require different gaits without an obviously identifiable shared behavior in the optimal policies but can still benefit from intermediate behaviors like balancing.

Kitchen: We use the challenging manipulation environment proposed by Gupta et al. (2019) where a 9 DoF Franka robot performs tasks in a kitchen. We create a task set out of 10 manipulation tasks: turning on or off different burners and light switches, and opening or closing different cabinets.

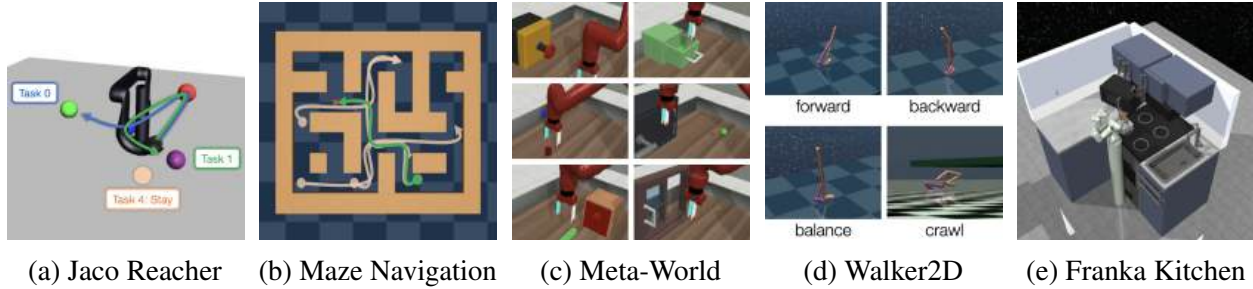


Figure 5.6: **Environments & Tasks:** (a) Multistage Jaco Reacher. The agent must reach different subgoals or stay still (Task 4). (b) Maze Navigation. The agent (green circle) must navigate to the goal (red circle). 4 other tasks are shown in orange. (c) Meta-World: 10 table-top manipulation tasks. (e) Franka Kitchen: 10 tasks, interacting with one appliance or cabinet.

5.6.2 Baselines

We first select popular and representative MTRL methods that share other forms of information to evaluate how behavior-sharing with QMP improves their performance:

- **No-Sharing** consists of N (number of tasks) independent RL architectures where each agent is assigned one task and trained to solve it without any information sharing with other agents.
- **Data-Sharing (UDS)** proposed in [Yu et al. \(2022\)](#) shares data between tasks, relabelling with minimum task reward. We modified this offline RL algorithm to online.
- **Parameter-Sharing** multi-head SAC policy sharing parameters but not behaviors over tasks.

We validate QMP’s approach to share behaviors via *off-policy data collection* with baselines:

- **No-Shared-Behaviors** consists of N RL agents where each agent is assigned one task and trained to solve it without any behavior sharing with other agents: no bias and no sharing.
- **Fully-Shared-Behaviors:** single SAC agent learning one shared policy, outputting the same action for a given state regardless of task (full parameter sharing): fully biased sharing.
- **Divide-and-Conquer RL (DnC)** ([Ghosh et al., 2018](#)) N policies sharing behaviors through policy distillation and mean regularization (adapted for MTRL): biased objective for sharing.
- **DnC (Regularization Only)** is a no policy distillation variant of DnC we propose as baseline.
- **QMP (Ours)** N policies that share behaviors in off-policy data collection: unbiased sharing.

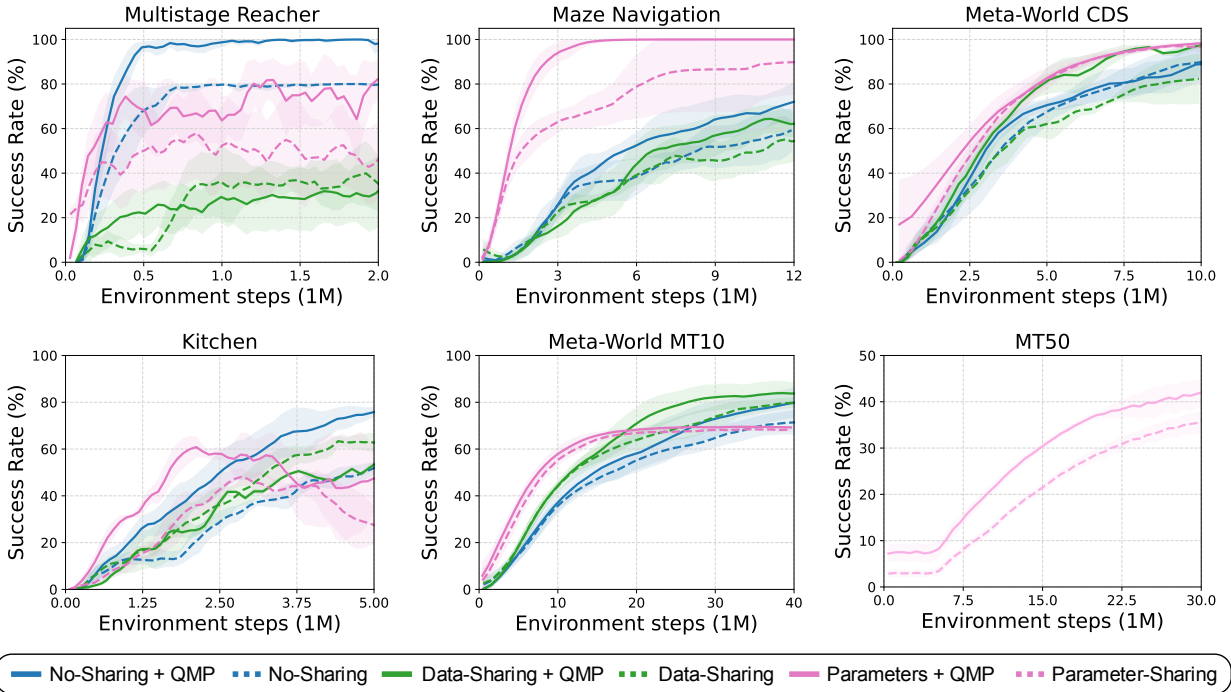


Figure 5.7: **Behavior sharing is complementary.** QMP (solid lines) shows improvement over MTRL frameworks (same-colored dashed lines): no-shared architecture (blue), shared parameters (pink), and shared data (green). Methods without parameter-sharing on MT50 converge very slowly. Success rate means and std (shaded) are over N tasks, 10 episodes per task, and 5 seeds.

We used SAC Haarnoja et al. (2018) for all environments and methods. All the non-parameter sharing baselines use the same SAC hyperparameters. Please refer to Appendix H for details.

5.7 Results

Our experiments address: (1) Does QMP provide complementary gains to other forms of MTRL? (2) How does sharing behavioral policies compare with alternate forms of behavior sharing? (3) Can QMP effectively identify shareable behaviors? (4) Ablating key components of QMP.

5.7.1 Is Behavior Sharing Complementary to other MTRL frameworks?

We demonstrate that our method is compatible with and provides complementary performance gains with other forms of MTRL that share different kinds of information, including parameter

sharing and data sharing. We compare the performance between 3 base MTRL algorithms, No-Sharing, Parameter-Sharing, and Data-Sharing, with the addition of QMP in Figure 5.7. The No-Sharing baseline provides a baseline comparison of QMP’s effectiveness on its own. For the Parameter-Sharing and Data-Sharing baselines we chose the base algorithms for their popularity and simplicity. In each case, we add QMP by simply replacing the data collection policy with π_i^{mix} . We find that **QMP is complementary to all three baseline frameworks**, mostly with additive performance gains in sample efficiency and final performance, while not hurting the performance of the base method in all but one case (Data-Sharing in Kitchen). We additionally see that QMP improves PCGrad’s (Yu et al., 2020) performance significantly in 3 out of 4 environments tested in Appendix E.3, showing that QMP is a simple and complementary addition to other forms of MTRL.

QMP significantly improves upon the No-Sharing baseline in all environments except Meta-World CDS where it performs comparatively. This demonstrates that sharing behavioral policies is a promising avenue for efficient and performant MTRL. In the data-sharing comparison, we see that the addition of QMP improves or performs comparatively to the base algorithm. In Multistage Reacher and Maze Navigation, we see that both Data-Sharing and Data + QMP perform worse than the other MTRL methods, highlighting the fact that sharing data directly between tasks can be ineffective without access to a re-labeled task rewards like in our setting. In environments where data-sharing does well, like Meta-World CDS, adding QMP does improve sample efficiency.

We find that Parameters + QMP generally outperforms Parameter-Sharing, while inheriting its sample efficiency gains. In many cases, the parameter-sharing methods converge sub-optimally, highlighting that shared parameter MTRL has its own challenges. However, in Maze Navigation, we find that sharing **Parameters + Behaviors greatly improves the performance over both the Parameter-Sharing baseline and No-Sharing + QMP variant of QMP**. This demonstrates the additive effect of these two forms of information sharing in MTRL. The agent initially benefits from the sample efficiency gains of the multi-head parameter-sharing architecture, while behavior sharing accelerates learning by selectively using other policies to keep learning even after the parameter-sharing effect plateaus. demonstrating the compatibility between QMP and parameter

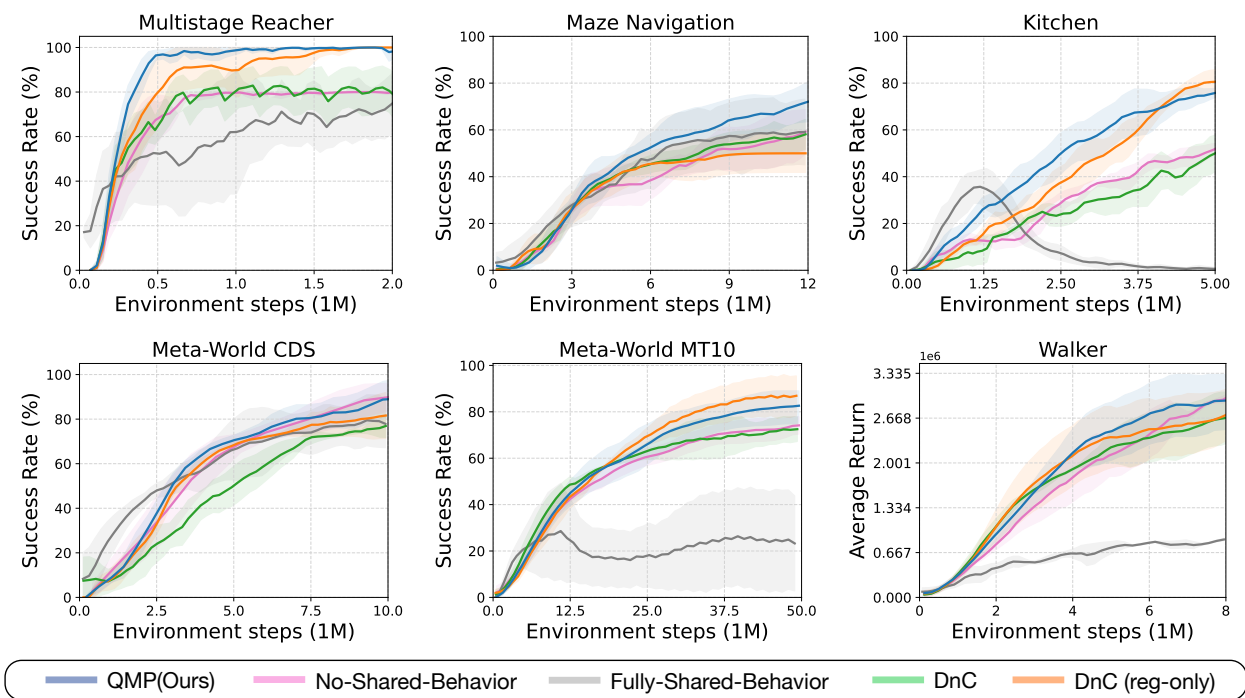


Figure 5.8: **QMP reliably shares behaviors.** In task sets exhibiting conflicting behaviors, QMP consistently matches or exceeds baselines in rate of convergence and final performance.

sharing as key ingredients to sample efficient MTRL. We further highlight that this **benefit of QMP increases with the number of tasks** increasing from 10 to 50 in Meta-World, where we see that QMP is actually more effective when combined with parameter sharing in MT50 than in MT10. QMP scales well with the number of tasks and can actually perform better likely due to *more shared behaviors* in the larger task set.

5.7.2 Baselines: Comparing Different Approaches to Share Behaviors

To verify QMP’s efficacy as a behavior-sharing mechanism, we evaluate baselines that share behaviors in different ways on 6 environments in Figure 5.8. QMP reliably matches or exceeds other methods, especially in tasks that require conflicting behaviors, while baselines are ineffective.

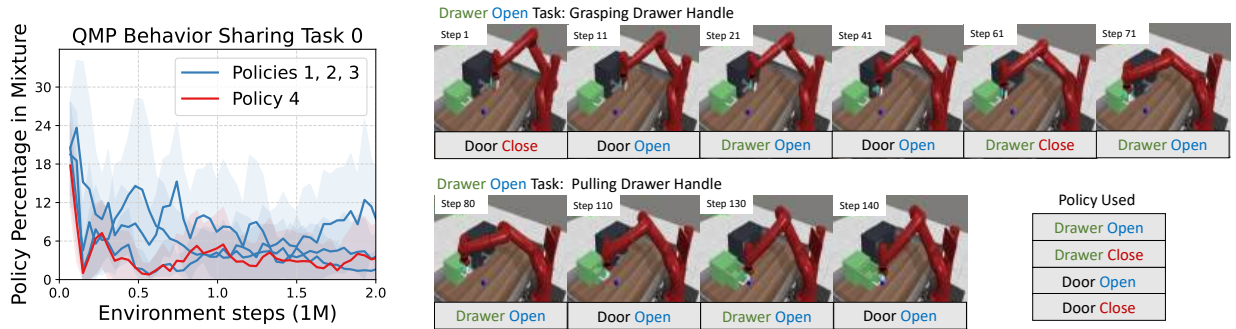
In the task sets with the most directly conflicting behaviors, Multistage Reacher and Maze Navigation, our method clearly outperforms other behavior-sharing and data-sharing baselines. In

Multistage Reacher, our method reaches $> 90\%$ success rate at 0.5 million environment steps, while DnC (reg.), the next best method, takes 3 times the number of steps to fully converge. The rest of the methods fail to attain the maximum success rate. The UDS baseline performs particularly poorly, illustrating that data sharing can be ineffective without ground truth rewards. We also see that QMP scales better from 3 to 10 tasks in Maze compared to other behavior sharing methods in Appendix Section E.4.

In the remaining task sets with no directly conflicting behaviors, we see that QMP is competitive with the best-performing baseline for more complex manipulation and locomotion tasks. Particularly, in Walker2D and Meta-World CDS, we see that QMP is the most sample-efficient method and converges to a better final performance than any other behavior sharing method. In Meta-World MT10 and Kitchen, DnC (regularization only) also performed very well, showing that well-tuned uniform behavior sharing can be very effective in tasks without conflicting behavior. However, QMP also performs competitively and more sample efficiently, showing that QMP is effective under the same assumptions as uniform behavior-sharing methods but can do so *adaptively* and across more *general task families*. The Fully-Shared-Behaviors baseline often performs poorly because it totally biases the policies, while the No-Shared-Behavior is a surprisingly strong baseline as it introduces no bias.

5.7.3 Can QMP effectively identify shareable behaviors?

Figure 5.9a shows the average proportion of sharing from other tasks for Multistage Reacher Task 0 over the course of training. We see that QMP learns to generally share less behavior from Policy 4 than from Policies 1-3 (Appendix Figure D.15). Conversely, QMP in Task 4 also shares the least total cross-task behavior (Appendix Figure D.14). We see this same trend across all 5 Multistage Reacher tasks, showing that the Q-switch successfully **identifies conflicting behaviors that should not be shared**. Further, Figure 5.9a also shows that **total behavior-sharing from other tasks goes down over training**. Thus, Q-switch learns to prefer its own task-specific policy as it becomes more proficient.



(a) Behavior-sharing over training

(b) Behavior-sharing in a single training episode.

Figure 5.9: (a) Mixture probabilities of other policies for Task 0 in Multistage Reacher with the conflicting task Policy 4 shown in red. (b) Policies chosen by the QMP behavioral policy every 10 timesteps for the Drawer Open task throughout one training episode. The policy approaches and grasps the handle (top row), then pulls the drawer open (bottom row).

We qualitatively analyze how behavior sharing varies within a single episode by visualizing a QMP rollout during training for the Drawer Open task in Meta-World CDS (Figure 5.9b). We see that it makes reasonable policy choices by (i) switching between all 4 task policies as it approaches the drawer (top row), (ii) using drawer-specific policies as it grasps the drawer-handle, and (iii) using Drawer Open and Door Open policies as it pulls the drawer open (bottom row). In conjunction with the overall results, this supports our claim that QMP can effectively identify shareable behaviors between tasks. For details on this visualization and the full analysis results see Appendix Section F.

Inspired by hierarchical RL (Dabney et al., 2021) and multi-task exploration (Xu et al., 2024), we briefly investigate *temporally extended behavior sharing* in Appendix E.6. Recently, Xu et al. (2024) showed that if one assumes a high overlap between optimal policies of different tasks, other task policies can aid exploration. So, we simply roll out each policy QMP selects for a fixed number of steps. While QMP theory no longer holds as it requires selecting a policy at every step, this naive temporal extension yields improvements in **some** environments like Maze with strong overlap.

5.7.4 Ablations

We show the importance of Q-switch in QMP (Def. 5.4.2) against alternate forms of policy mixtures (Def. 5.4.1). **QMP-Uniform** is a uniform distribution over policies, $f_i = \mathbb{U}(\{1, \dots, N\})$

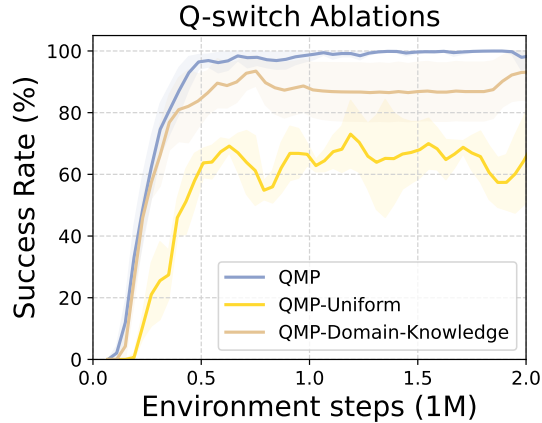


Figure 5.10: QMP outperforms alternate policy mixtures in Multistage Reacher.

and achieves only 60% success rate (Figure 5.10), showing the importance of selectivity. **QMP-Domain-Knowledge** is a hand-crafted, fixed policy distribution based on an estimate of similarity between tasks. Multistage Reacher measures this similarity by the shared sub-goal sequences between tasks (Appendix D). QMP-Domain performs well initially but plateaus early, showing that which behaviors are shareable depends on the state and current policy. We further ablate the $\arg \max$ in Q-switch against a softmax variation resulting in a *probabilistic mixture policy* in Appendix G.1, and evaluating on the *mean policy actions* (Appendix G.2) to validate our design.

5.8 Conclusion

In this chapter, we propose an unbiased approach to sharing behaviors in MTRL: Q-switch Mixture of Policies. QMP shares policies between tasks for off-policy data collection. We demonstrate empirically that QMP effectively improves the rate of convergence and task performance in manipulation, locomotion, and navigation tasks, and is guaranteed to be as good as the underlying RL algorithm and complementary to alternate MTRL. QMP does not assume that optimal task behaviors always coincide. Thus, its improvement magnitude is limited by the degree of shareable behaviors and the suboptimality gap that exists. At the same time, this lets the QMP algorithm be unbiased and find optimal policies with convergence guarantees while being equally or more sample-efficient.

Chapter 6

Conclusion

In this thesis, we presented methods for decision-making in complex action spaces where traditional reinforcement learning approaches fall short. Specifically, we addressed decision-making challenges involving unseen, varying, non-convex, and shareable action spaces by leveraging action representations and multiple policy candidates. Through developing new environments like CREATE, adapting existing tools like PPO, TD3, and SAC, and proposing new algorithms like SAVO and QMP, we demonstrated that intelligent agents could achieve effective generalization, optimal action selection, and efficient multi-task performance in action space settings reminiscent of human decision-making.

In Part I, we showed how action representations could enable an agent to generalize effectively to unseen and varying action spaces. By leveraging action metadata and learning representations through deep autoencoders, we were able to equip agents with the ability to reason about novel actions and adapt dynamically to their availability, much like humans do when cooking in a new kitchen. We found that relational reasoning between actions, modeled using graph neural networks, further enhanced this ability by accounting for interdependencies within a changing action set.

In Part II, we tackled the challenge of finding optimal actions within non-convex action-value landscapes. We developed methods for iteratively refining these landscapes, allowing agents to escape local optima by combining multiple policies and utilizing refined value landscapes inspired by tabu search. Extending this approach to multi-task reinforcement learning, we proposed a

novel paradigm of action sharing between tasks, showing significant gains in sample efficiency and performance.

The contributions of this thesis have applications in diverse domains such as robotics, recommendation systems, and physical reasoning. More importantly, the concepts we introduced help bridge the gap between human-like decision-making and current RL systems by enabling effective generalization, adaptation, optimization, and sharing in complex action spaces.

6.1 Open Challenges and Future Directions

While addressing the fundamental challenges in decision-making over complex action spaces, we discover new research questions.

Conception of action spaces. How can agents identify their action space for various tasks? All reinforcement learning works, including the problems considered in this work, assume access to a predefined auxiliary system that can enumerate or define the available actions for task solving. For instance, generalization to unseen tool choices is well defined over a list of tools that an auxiliary system provides to the agent. However, any AI agent, whether a virtual agent browsing the Internet or an embodied robot controlling its joint torques, has a fixed embodied action space where it can act. Yet, this agent must solve tasks at a higher abstraction, such as decision-making over different internet links to click on and different kitchen tools or ingredients to use while cooking.

Humans are embodied agents that can make innumerable decisions from self-conceived action spaces in the mental models of various tasks they encounter. Imagine planning and reasoning about potential future career paths. Each path represents an action in a decision-making process, yet we seamlessly formulate and decide between such actions by estimating their long-term impact. Similarly, in the physical world, what tools are viable in a particular task is ambiguous. Yet, we can truly improvise by reasoning about surrounding objects as a potential tool or even create new tools that can be used to solve our tasks. Identifying and creating an abstracted action space to subsequently use for achieving goals is a crucial hallmark of human intelligence, which remains farfetched for current RL agents as a fundamental open challenge of action space.

Reinforcement learning with token action space in LLMs. Reinforcement learning in token-level decision-making within large language models (LLMs) (Ouyang et al., 2022) introduces unique challenges due to the fixed yet vast discrete action space. This token space encompasses every possible token that the model can generate, making exploration inherently complex. Current approaches explore with temperature-based sampling from the current model’s probability distribution. This is a valid heuristic mechanism for exploration but limits the experiences to be close to the initial policy without necessarily covering the entire token space, especially for low-probability tokens that might be critical in niche contexts. This lack of coverage limits the capability of fine-tuning the model with a reinforcement learning signal because all algorithms require at least a minimal coverage over all the tokens.

Future research should explore exploration strategies with better state-action coverage. However, doing so sample-efficiently requires domain-specific research in how to heuristically limit the space of exploration. One way is to leverage hierarchical action representations where groups of tokens or subword units form higher-level “meta-actions” for exploration and data collection. Additionally, exploration methods in RL could be incorporated, such as intrinsic motivation (Schmidhuber, 2010), curiosity (Pathak et al., 2017), and uncertainty-based exploration (Osband et al., 2016; Burda et al., 2018) could enable LLMs to autonomously discover novel yet meaningful token sequences. Similarly, specialized hard exploration techniques that can exploit simulation and resetting, like GoExplore (Ecoffet et al., 2019) are suitable for LLM token space exploration. These advances could substantially enhance the fine-tunability of LLMs for tasks that may be vastly different from the domain of the pre-training data.

Actor-free Q-learning algorithms. One key reason for the introduction of actors in reinforcement learning in continuous action spaces is the need to find the optimal action. The state-of-the-art off-policy RL methods like DDPG and SAC utilize deterministic or stochastic actors for this exact purpose. However, this introduction of a gradient-based actor introduces challenges of local optima as we discovered in Part II. One natural question arises about whether such actors are even needed and what alternatives exist to find the most optimal action given a Q-function. One way to achieve

this could be to embed the maximization in the Q-function itself, which was explored in normalized advantage functions [Gu et al. \(2016\)](#). However, this modeling constraint limits the expressivity of the Q-function, and more expressive Q-function models could help alleviate the need for an actor completely.

References

- Monireh Abdoos, Nasser Mozayani, and Ana LC Bazzan. Traffic light control in non-stationary environments based on multi agent q-learning. In *2011 14th International IEEE conference on intelligent transportation systems (ITSC)*, pages 1580–1585. IEEE, 2011.
- Johannes Ackermann, Oliver Richter, and Roger Wattenhofer. Unsupervised task clustering for multi-task reinforcement learning. In Nuria Oliver, Fernando Pérez-Cruz, Stefan Kramer, Jesse Read, and Jose A. Lozano, editors, *Machine Learning and Knowledge Discovery in Databases. Research Track*, pages 222–237, Cham, 2021. Springer International Publishing. ISBN 978-3-030-86486-6.
- Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.
- Kelsey R Allen, Kevin A Smith, and Joshua B Tenenbaum. The tools challenge: Rapid trial-and-error learning in physical problem solving. *arXiv preprint arXiv:1907.09620*, 2019.
- Brandon Amos, Lei Xu, and J Zico Kolter. Input convex neural networks. In *International conference on machine learning*, pages 146–155. PMLR, 2017.
- Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *International Conference on Machine Learning*, pages 166–175, 2017.
- Leemon C Baird and A Harry Klopf. Reinforcement learning with high-dimensional continuous actions. *Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147*, 15, 1993.
- Anton Bakhtin, Laurens van der Maaten, Justin Johnson, Laura Gustafson, and Ross Girshick. Phyre: A new benchmark for physical reasoning. *arXiv:1908.05656*, 2019.
- Sai Praveen Bangaru, JS Suhas, and Balaraman Ravindran. Exploration for multi-task reinforcement learning with deep generative models. *arXiv preprint arXiv:1611.09894*, 2016.
- Yotam Barnoy, Molly O’Brien, Will Wang, and Gregory Hager. Robotic surgery with lean reinforcement learning. *arXiv preprint arXiv:2105.01006*, 2021.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8): 1798–1828, 2013.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Lukas Biewald. Experiment tracking with weights and biases. *Software available from wandb.com*, 2:233, 2020a.

Lukas Biewald. Experiment tracking with weights and biases, 2020b. URL <https://www.wandb.com/>. Software available from wandb.com.

Victor Blomqvist. Pymunk, n.d. URL <http://www.pymunk.org/>. Accessed 2020-02-18.

Olivier Bousquet, Stéphane Boucheron, and Gábor Lugosi. Introduction to statistical learning theory. In *Summer School on Machine Learning*, pages 169–207. Springer, 2003.

Craig Boutilier, Alon Cohen, Amit Daniely, Avinatan Hassidim, Yishay Mansour, Ofer Meshi, Martin Mladenov, and Dale Schuurmans. Planning and learning with stochastic action sets. *arXiv preprint arXiv:1805.02363*, 2018.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

Yash Chandak, Georgios Theocharous, James Kostas, Scott Jordan, and Philip Thomas. Learning action representations for reinforcement learning. In *International Conference on Machine Learning*, pages 941–950. PMLR, 2019.

Yash Chandak, Georgios Theocharous, Blossom Metevier, and Philip Thomas. Reinforcement learning when all actions are not always available. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3381–3388, 2020a.

Yash Chandak, Georgios Theocharous, Chris Nota, and Philip Thomas. Lifelong learning with a changing action set. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3373–3380, Apr 2020b. ISSN 2159-5399. doi: 10.1609/aaai.v34i04.5739. URL <http://dx.doi.org/10.1609/aaai.v34i04.5739>.

Yash Chandak, Georgios Theodorou, Chris Nota, and Philip Thomas. Lifelong learning with a changing action set. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34 (04):3373–3380, 2020c.

Gang Chen and Yiming Peng. Off-policy actor-critic in an ensemble: Achieving maximum general entropy and effective environment exploration in deep reinforcement learning. *arXiv preprint arXiv:1902.05551*, 2019.

Xinshi Chen, Shuang Li, Hui Li, Shaohua Jiang, Yuan Qi, and Le Song. Generative adversarial user model for reinforcement learning based recommendation system. In *International Conference on Machine Learning*, pages 1052–1061. PMLR, 2019a.

Xinyue Chen, Che Wang, Zijian Zhou, and Keith W Ross. Randomized ensembled double q-learning: Learning fast without a model. In *International Conference on Learning Representations*, 2020.

Yu Chen, Yingfeng Chen, Yu Yang, Ying Li, Jianwei Yin, and Changjie Fan. Learning action-transferable policy with action embedding. *arXiv preprint arXiv:1909.02291*, 2019b.

Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. GradNorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, 2018.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishikesh Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

Yuan Cheng, Songtao Feng, Jing Yang, Hong Zhang, and Yingbin Liang. Provable benefit of multitask representation learning in reinforcement learning. In *Neural Information Processing Systems*, 2023.

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. <https://github.com/maximecb/gym-minigrid>, 2018.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Anna Choromanska, Mikael Henaff, Michael Mathieu, Gerard Ben Arous, and Yann LeCun. The Loss Surfaces of Multilayer Networks. In Guy Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *Proceedings of Machine Learning Research*, pages 192–204, San Diego, California, USA, 09–12 May 2015. PMLR. URL <https://proceedings.mlr.press/v38/choromanska15.html>.

Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *International Conference on Machine Learning*, pages 834–843, 2017.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

John Co-Reyes, YuXuan Liu, Abhishek Gupta, Benjamin Eysenbach, Pieter Abbeel, and Sergey Levine. Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings. In *International Conference on Machine Learning*, pages 1008–1017, 2018.

Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Will Dabney, Georg Ostrovski, and Andre Barreto. Temporally-extended ε -greedy exploration. In *International Conference on Learning Representations*, 2021.

Christian Daniel, Gerhard Neumann, Oliver Kroemer, and Jan Peters. Hierarchical relative entropy policy search. *Journal of Machine Learning Research*, 2016.

Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

Emily L Denton and vighnesh Birodkar. Unsupervised learning of disentangled representations from video. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4414–4423. Curran Associates, Inc., 2017. URL <http://papers.nips.cc/paper/7028-unsupervised-learning-of-disentangled-representations-from-video.pdf>.

Carlo D’Eramo, Davide Tateo, Andrea Bonarini, Marcello Restelli, Jan Peters, et al. Sharing knowledge in multi-task deep reinforcement learning. In *International Conference on Learning Representations*, 2020.

Coline Devin, Abhishek Gupta, Trevor Darrell, Pieter Abbeel, and Sergey Levine. Learning modular neural network policies for multi-task and multi-robot transfer. In *IEEE International Conference on Robotics and Automation*, 2017.

Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Frederik Ebert, Chelsea Finn, Alex X Lee, and Sergey Levine. Self-supervised visual planning with temporal skip connections. In *Conference on Robot Learning*, pages 344–356, 2017.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

Harrison Edwards and Amos Storkey. Towards a neural statistician. In *International Conference on Learning Representations*, 2017. URL <https://openreview.net/forum?id=HJDBUF51e>.

Felix End, Riad Akrouf, Jan Peters, and Gerhard Neumann. Layered direct policy search for learning hierarchical skills. In *IEEE International Conference on Robotics and Automation*, 2017.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

Kuan Fang, Yuke Zhu, Animesh Garg, Andrey Kurenkov, Viraj Mehta, Li Fei-Fei, and Silvio Savarese. Learning task-oriented grasping for tool manipulation from simulated self-supervision. *arXiv preprint arXiv:1806.09266*, 2018.

Chris Fifty, Ehsan Amid, Zhe Zhao, Tianhe Yu, Rohan Anil, and Chelsea Finn. Efficiently identifying task groupings for multi-task learning. In *Neural Information Processing Systems*, 2021.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

Pete Florence, Corey Lynch, Andy Zeng, Oscar A Ramirez, Ayzaan Wahid, Laura Downs, Adrian Wong, Johnny Lee, Igor Mordatch, and Jonathan Tompson. Implicit behavioral cloning. In *Conference on Robot Learning*, pages 158–168. PMLR, 2022.

Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. <https://github.com/rlworkgroup/garage>, 2019.

Samuel J Gershman and Yael Niv. Novelty and inductive generalization in human reinforcement learning. *Topics in cognitive science*, 7(3):391–415, 2015.

Dibya Ghosh, Avi Singh, Aravind Rajeswaran, Vikash Kumar, and Sergey Levine. Divide-and-conquer reinforcement learning. In *International Conference on Learning Representations*, 2018.

Ruben Glatt, Felipe Leno Da Silva, Reinaldo Augusto da Costa Bianchi, and Anna Helena Reali Costa. Decaf: Deep case-based policy inference for knowledge transfer in reinforcement learning. *Expert Systems with Applications*, 2020.

Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

Yu Gong, Yu Zhu, Lu Duan, Qingwen Liu, Ziyu Guan, Fei Sun, Wenwu Ou, and Kenny Q Zhu. Exact-k recommendation via maximal clique optimization. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 617–626, 2019.

Ian Goodfellow. Deep learning, 2016.

Anirudh Goyal, Shagun Sodhani, Jonathan Binas, Xue Bin Peng, Sergey Levine, and Yoshua Bengio. Reinforcement learning with competitive ensembles of information-constrained primitives. *arXiv*, abs/1906.10667, 2019.

Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews)*, 42(6):1291–1307, 2012.

Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*, pages 2829–2838. PMLR, 2016.

Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long horizon tasks via imitation and reinforcement learning. In *Conference on Robot Learning*, 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

Jessica B Hamrick, Kelsey R Allen, Victor Bapst, Tina Zhu, Kevin R McKee, Joshua B Tenenbaum, and Peter W Battaglia. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018.

Steven Hansen, Will Dabney, Andre Barreto, Tom Van de Wiele, David Warde-Farley, and Volodymyr Mnih. Fast task inference with variational intrinsic successor features. *arXiv preprint arXiv:1906.05030*, 2019.

Junheng Hao, Tong Zhao, Jin Li, Xin Luna Dong, Christos Faloutsos, Yizhou Sun, and Wei Wang. P-companion: A principled framework for diversified complementary product recommendation. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pages 2517–2524, 2020.

F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

- Matthew Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. *arXiv preprint arXiv:1511.04143*, 2015.
- Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.
- Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. Deep reinforcement learning with a natural language action space. *arXiv preprint arXiv:1511.04636*, 2015.
- Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI conference on artificial intelligence*, 32(1), 2018.
- Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. In *AAAI Conference on Artificial Intelligence*, 2019.
- Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. *ICLR*, 2(5):6, 2017a.
- Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1480–1490. JMLR. org, 2017b.
- Sunghoon Hong, Deunsol Yoon, and Kee-Eung Kim. Structure-aware transformer policy for inhomogeneous multi-task reinforcement learning. In *International Conference on Learning Representations*, 2022.
- Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.
- Jiaqiao Hu, Michael C Fu, and Steven I Marcus. A model reference adaptive search method for global optimization. *Operations research*, 55(3):549–568, 2007.
- Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *arXiv preprint arXiv:2006.14171*, 2020.
- Zhewei Huang, Shuchang Zhou, BoEr Zhuang, and Xinyu Zhou. Learning to run with actor-critic ensemble. *arXiv preprint arXiv:1712.08987*, 2017.
- Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging, 2015.
- Eugene Ie, Chih-wei Hsu, Martin Mladenov, Vihan Jain, Sanmit Narvekar, Jing Wang, Rui Wu, and Craig Boutilier. Recsim: A configurable simulation platform for recommender systems. *arXiv preprint arXiv:1909.04847*, 2019a.

Eugene Ie, Vihan Jain, Jing Wang, Sanmit Narvekar, Ritesh Agarwal, Rui Wu, Heng-Tze Cheng, Tushar Chandra, and Craig Boutilier. Slateq: A tractable decomposition for reinforcement learning with recommendation sets. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2019b.

Masaaki Imaizumi and Kenji Fukumizu. Deep neural networks learn non-smooth functions effectively. In *The 22nd international conference on artificial intelligence and statistics*, pages 869–878. PMLR, 2019.

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

Ayush Jain, Andrew Szot, and Joseph Lim. Generalization to new actions in reinforcement learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4661–4672. PMLR, 13–18 Jul 2020. URL <http://proceedings.mlr.press/v119/jain20b.html>.

Ayush Jain, Norio Kosaka, Kyung-Min Kim, and Joseph J Lim. Know your action set: Learning action relations for reinforcement learning. In *International Conference on Learning Representations*, 2021.

Ayush Jain, Norio Kosaka, Xinhui Li, Kyung-Min Kim, Erdem Bıyık, and Joseph J Lim. Mitigating suboptimality of deterministic policy gradients in complex q-functions. *arXiv preprint arXiv:2410.11833*, 2024.

Ray Jiang, Sven Gowal, Timothy A Mann, and Danilo J Rezende. Beyond greedy ranking: Slate optimization via list-cvae. *arXiv preprint arXiv:1803.01682*, 2018.

Yuu Jinnai, Jee Won Park, David Abel, and George Konidaris. Discovering options for exploration by minimizing cover time. In *International Conference on Machine Learning*, 2019a.

Yuu Jinnai, Jee Won Park, Marlos C Machado, and George Konidaris. Exploration in reinforcement learning with deep covering options. In *International Conference on Learning Representations*, 2019b.

Leslie Pack Kaelbling. Learning to achieve goals. In *International Joint Conference on Artificial Intelligence*, 1993.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

Sham Machandranath Kakade. *On the sample complexity of reinforcement learning*. University of London, University College London (United Kingdom), 2003.

Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, et al. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning*, pages 651–673. PMLR, 2018.

Dmitry Kalashnikov, Jacob Varley, Yevgen Chebotar, Benjamin Swanson, Rico Jonschkowski, Chelsea Finn, Sergey Levine, and Karol Hausman. Mt-opt: Continuous multi-task robotic reinforcement learning at scale. *arXiv preprint arXiv:2104.08212*, 2021a.

Dmitry Kalashnikov, Jake Varley, Yevgen Chebotar, Benjamin Swanson, Rico Jonschkowski, Chelsea Finn, Sergey Levine, and Karol Hausman. Scaling up multi-task robotic reinforcement learning. In *Conference on Robot Learning*, 2021b.

Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*, pages 479–486. IEEE, 2020.

Hanjoo Kim, Minkyu Kim, Dongjoo Seo, Jinwoong Kim, Heungseok Park, Soeun Park, Hyunwoo Jo, KyungHyun Kim, Youngil Yang, Youngkwan Kim, et al. Nsmi: Meet the mlaas platform with a real-world case study. *arXiv preprint arXiv:1810.09957*, 2018.

Hyoungseok Kim, Jaekyeom Kim, Yeonwoo Jeong, Sergey Levine, and Hyun Oh Song. EMI: Exploration with mutual information. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3360–3369, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL <http://proceedings.mlr.press/v97/kim19a.html>.

Woojun Kim, Yongjae Shin, Jongeui Park, and Youngchul Sung. Sample-efficient and safe deep reinforcement learning via reset deep ensemble agents. *Advances in Neural Information Processing Systems*, 36, 2024.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. <https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail>, 2018.

Vitaly Kurin, Alessandro De Palma, Ilya Kostrikov, Shimon Whiteson, and M. Pawan Kumar. In defense of the unitary scalarization for deep multi-task learning. *arXiv preprint arXiv:2201.04122*, 2022.

- Adrien Laversanne-Finot, Alexandre Pere, and Pierre-Yves Oudeyer. Curiosity driven exploration of learned disentangled goal spaces. In Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto, editors, *Proceedings of The 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research*, pages 487–504. PMLR, 29–31 Oct 2018. URL <http://proceedings.mlr.press/v87/laversanne-finot18a.html>.
- Kuang-Huei Lee, Ted Xiao, Adrian Li, Paul Wohlhart, Ian Fischer, and Yao Lu. Pi-qt-opt: Predictive information improves multi-task robotic reinforcement learning at scale. In *Conference on Robot Learning*, pages 1696–1707. PMLR, 2023.
- Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward S. Hu, and Joseph J. Lim. Composing complex skills by learning transition policies. In *International Conference on Learning Representations*, 2019.
- Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
- Michael Lederman Littman. *Algorithms for sequential decision-making*. Brown University, 1996.
- Bo Liu, Xingchao Liu, Xiaojie Jin, Peter Stone, and Qiang Liu. Conflict-averse gradient descent for multi-task learning. In *Neural Information Processing Systems*, 2021a.
- Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.
- Shikun Liu, Stephen James, Andrew J Davison, and Edward Johns. Auto-lambda: Disentangling dynamic task relationships. *Transactions on Machine Learning Research*, 2022.
- Shuchang Liu, Fei Sun, Yingqiang Ge, Changhua Pei, and Yongfeng Zhang. Variation control and evaluation for generative slate recommendations. *Proceedings of the Web Conference 2021*, Apr 2021b. doi: 10.1145/3442381.3449864. URL <http://dx.doi.org/10.1145/3442381.3449864>.
- Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Citeseer, 2013.
- Marlos C Machado, Clemens Rosenbaum, Xiaoxiao Guo, Miao Liu, Gerald Tesauro, and Murray Campbell. Eigenoption discovery through the deep successor representation. *arXiv preprint arXiv:1710.11089*, 2017.
- Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Siddharth Mysore, George Cheng, Yunqi Zhao, Kate Saenko, and Meng Wu. Multi-critic actor learning: Teaching rl policies to act with style. In *International Conference on Learning Representations*, 2022.

Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations. In *IEEE International Conference on Robotics and Automation*, 2018a.

Ashvin V Nair, Vitchyr Pong, Murtaza Dalal, Shikhar Bahl, Steven Lin, and Sergey Levine. Visual reinforcement learning with imagined goals. In *Advances in Neural Information Processing Systems*, pages 9191–9200, 2018b.

Taewook Nam, Shao-Hua Sun, Karl Pertsch, Sung Ju Hwang, and Joseph J. Lim. Skill-based meta-reinforcement learning. In *International Conference on Learning Representations*, 2022.

Samuel Neumann, Sungsu Lim, Ajin Joseph, Yangchen Pan, Adam White, and Martha White. Greedy actor-critic: A new conditional cross-entropy method for policy improvement. *arXiv preprint arXiv:1810.09103*, 2018.

Behnam Neyshabur, Srinadh Bhojanapalli, David McAllester, and Nati Srebro. Exploring generalization in deep learning. *Advances in neural information processing systems*, 30, 2017.

Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*, 2018.

Evgenii Nikishin, Max Schwarzer, Pierluca D’Oro, Pierre-Luc Bacon, and Aaron Courville. The primacy bias in deep reinforcement learning. In *International conference on machine learning*, pages 16828–16847. PMLR, 2022.

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. In *International Conference on Machine Learning*, pages 2661–2670, 2017.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems*, 29, 2016.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *arXiv preprint arXiv:1802.07569*, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, 2017.

Deepak Pathak, Chris Lu, Trevor Darrell, Phillip Isola, and Alexei A. Efros. Learning to control self-assembling morphologies: A study of generalization via modularity. In *arXiv preprint arXiv:1902.05546*, 2019a.

Deepak Pathak, Chris Lu, Trevor Darrell, Phillip Isola, and Alexei A. Efros. Learning to control self-assembling morphologies: A study of generalization via modularity, 2019b.

Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. In *AAAI Conference on Artificial Intelligence*, 2018a.

Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. *Proceedings of the AAAI conference on artificial intelligence*, 32(1), 2018b.

Karl Pertsch, Youngwoon Lee, and Joseph Lim. Accelerating reinforcement learning with learned skill priors. In *Conference on Robot Learning*, 2021.

Aloïs Pourchot and Olivier Sigaud. Cem-rl: Combining evolutionary and gradient-based methods for policy search. *arXiv preprint arXiv:1810.01222*, 2018.

Zhiwei Qin, Xiaocheng Tang, Yan Jiao, Fan Zhang, Zhe Xu, Hongtu Zhu, and Jieping Ye. Ride-hailing order dispatching at didi via reinforcement learning. *INFORMS Journal on Applied Analytics*, 50(5):272–286, 2020.

Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017.

Matthew Riemer, Miao Liu, and Gerald Tesauro. Learning abstract options. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS' 18*, page 10445–10455, Red Hook, NY, USA, 2018. Curran Associates Inc.

David Rohde, Stephen Bonner, Travis Dunlop, Flavian Vasile, and Alexandros Karatzoglou. Recogym: A reinforcement learning environment for the problem of product recommendation in online advertising. *arXiv preprint arXiv:1808.00720*, 2018.

Clemens Rosenbaum, Ignacio Cases, Matthew Riemer, and Tim Klinger. Routing networks and the challenges of modular and compositional computation. *arXiv preprint arXiv:1904.12774*, 2019.

Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, 2018a.

Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pages 4470–4479. PMLR, 2018b.

Fumihiko Sasaki and Ryota Yamashina. Behavioral cloning from noisy demonstrations. In *International Conference on Learning Representations*, 2020.

Tom Schaul, Diana Borsa, Joseph Modayil, and Razvan Pascanu. Ray interference: a source of plateaus in deep reinforcement learning. *arXiv preprint arXiv:1904.11455*, 2019.

Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE transactions on autonomous mental development*, 2(3):230–247, 2010.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017a.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017b.

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. In *Neural Information Processing Systems*, 2018.

Lin Shao, Yifan You, Mengyuan Yan, Shenli Yuan, Qingyun Sun, and Jeannette Bohg. Grac: Self-guided and self-regularized actor-critic. In *Conference on Robot Learning*, pages 267–276. PMLR, 2022.

Pete Shinnars. Pygame, n.d. URL <http://pygame.org/>. Accessed 2020-02-18.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Riley Simmons-Edler, Ben Eisner, Eric Mitchell, Sebastian Seung, and Daniel Lee. Q-learning for continuous actions with cross-entropy guided policies. *arXiv preprint arXiv:1903.10605*, 2019.

Shagun Sodhani, Amy Zhang, and Joelle Pineau. Multi-task reinforcement learning with context-based representations. In *International Conference on Machine Learning*, 2021.

Edward Jay Sondik. *The optimal control of partially observable Markov processes*. Stanford University, 1971.

Junshuai Song, Zhao Li, Chang Zhou, Jinze Bai, Zhenpeng Li, Jian Li, and Jun Gao. Co-displayed items aware list recommendation. *IEEE Access*, 8:64591–64602, 2020. doi: 10.1109/ACCESS.2020.2984543.

Yanjie Song, Ponnuthurai Nagaratnam Suganthan, Witold Pedrycz, Junwei Ou, Yongming He, Yingwu Chen, and Yutong Wu. Ensemble reinforcement learning: A survey. *Applied Soft Computing*, page 110975, 2023.

Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *computer*, 27(6): 17–26, 1994.

Trevor Standley, Amir R. Zamir, Dawn Chen, Leonidas Guibas, Jitendra Malik, and Silvio Savarese. Which tasks should be learned together in multi-task learning? In *International Conference on Machine Learning*, 2020.

Xander Steenbrugge, Sam Leroux, Tim Verbelen, and Bart Dhoedt. Improving generalization for abstract reasoning tasks using disentangled feature representations. *arXiv preprint arXiv:1811.04784*, 2018.

Peter Sunehag, Richard Evans, Gabriel Dulac-Arnold, Yori Zwols, Daniel Visentin, and Ben Coppin. Deep reinforcement learning with attention for slate markov decision processes with high-dimensional states and actions. *arXiv preprint arXiv:1512.01124*, 2015.

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

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

Andrea Tacchetti, H. Francis Song, Pedro A. M. Mediano, Vinicius Zambaldi, János Kramár, Neil C. Rabinowitz, Thore Graepel, Matthew Botvinick, and Peter W. Battaglia. Relational forward models for multi-agent learning. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=rJlEojAqFm>.

Yee Teh, Victor Bapst, Wojciech M Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In *Neural Information Processing Systems*, 2017.

Guy Tennenholtz and Shie Mannor. The natural language of actions. *arXiv preprint arXiv:1902.01119*, 2019.

Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

Momchil S Tomov, Eric Schulz, and Samuel J Gershman. Multi-task reinforcement learning in humans. *Nature Human Behaviour*, 2021.

Hado Van Hasselt and Marco A Wiering. Using continuous action spaces to solve discrete problems. In *2009 International Joint Conference on Neural Networks*, pages 1149–1156. IEEE, 2009.

Vladimir Vapnik. *Statistical learning theory*, 1998.

Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782): 350–354, 2019.

Nelson Vithayathil Varghese and Qusay H Mahmoud. A survey of multi-task deep reinforcement learning. *Electronics*, 2020.

Tung-Long Vuong, Do-Van Nguyen, Tai-Long Nguyen, Cong-Minh Bui, Hai-Dang Kieu, Viet-Cuong Ta, Quoc-Long Tran, and Thanh-Ha Le. Sharing experience in multitask reinforcement learning. In *International Joint Conference on Artificial Intelligence*, 2019.

Risto Vuorio, Shao-Hua Sun, Hexiang Hu, and Joseph J Lim. Multimodal model-agnostic meta-learning via task-aware modulation. In *Neural Information Processing Systems*, 2019.

Pin Wang, Hanhan Li, and Ching-Yao Chan. Quadratic q-network for learning continuous control for autonomous vehicles. *arXiv preprint arXiv:1912.00074*, 2019.

Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *International Conference on Learning Representations*, 2018.

Ziyu Wang, Josh S Merel, Scott E Reed, Nando de Freitas, Gregory Wayne, and Nicolas Heess. Robust imitation of diverse behaviors. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5320–5329. Curran Associates, Inc., 2017. URL <http://papers.nips.cc/paper/7116-robust-imitation-of-diverse-behaviors.pdf>.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. Multi-task reinforcement learning: a hierarchical bayesian approach. In *International Conference on Machine Learning*, 2007.

Qingyun Wu, Hongning Wang, Liangjie Hong, and Yue Shi. Returning is believing: Optimizing long-term user engagement in recommender systems. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1927–1936, 2017.

Annie Xie, Frederik Ebert, Sergey Levine, and Chelsea Finn. Improvisation through physical understanding: Using novel objects as tools with visual foresight, 2019.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *International Conference on Robotics and Automation*, 2017.

Zhiyuan Xu, Kun Wu, Zhengping Che, Jian Tang, and Jieping Ye. Knowledge transfer in multi-task deep reinforcement learning for continuous control. In *Neural Information Processing Systems*, 2020.

Ziping Xu, Zifan Xu, Runxuan Jiang, Peter Stone, and Ambuj Tewari. Sample efficient myopic exploration through multitask reinforcement learning with diverse tasks. In *International Conference on Learning Representations*, 2024.

Mengyuan Yan, Adrian Li, Mrinal Kalakrishnan, and Peter Pastor. Learning probabilistic multi-modal actor models for vision-based robotic grasping. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 4804–4810. IEEE, 2019.

Ruihan Yang, Huazhe Xu, Yi Wu, and Xiaolong Wang. Multi-task reinforcement learning with soft modularization. In *Neural Information Processing Systems*, 2020.

Deheng Ye, Zhao Liu, Mingfei Sun, Bei Shi, Peilin Zhao, Hao Wu, Hongsheng Yu, Shaojie Yang, Xipeng Wu, Qingwei Guo, et al. Mastering complex control in moba games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6672–6679, 2020.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*, 2019.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. In *Neural Information Processing Systems*, 2020.

Tianhe Yu, Aviral Kumar, Yevgen Chebotar, Karol Hausman, Sergey Levine, and Chelsea Finn. Conservative data sharing for multi-task offline reinforcement learning. In *Neural Information Processing Systems*, 2021.

Tianhe Yu, Aviral Kumar, Yevgen Chebotar, Karol Hausman, Chelsea Finn, and Sergey Levine. How to leverage unlabeled data in offline reinforcement learning. In *International Conference on Machine Learning*, 2022.

Yuanqiang Yu, Tianpei Yang, Yongliang Lv, Yan Zheng, and Jianye Hao. T3s: Improving multi-task reinforcement learning with task-specific feature selector and scheduler. In *International Joint Conference on Neural Networks*, 2023.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017a. URL <https://proceedings.neurips.cc/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf>.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017b.

Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2018.

Chicheng Zhang and Zhi Wang. Provably efficient multi-task reinforcement learning with model transfer. In *Neural Information Processing Systems*, 2021.

Grace Zhang, Ayush Jain, Injune Hwang, Shao-Hua Sun, and Joseph J Lim. Qmp: Q-switch mixture of policies for multi-task behavior sharing. *arXiv preprint arXiv:2302.00671*, 2023.

Jesse Zhang, Haonan Yu, and Wei Xu. Hierarchical reinforcement learning by discovering intrinsic options. In *International Conference on Learning Representations*, 2020.

Jin Zhang, Siyuan Li, and Chongjie Zhang. CUP: Critic-guided policy reuse. In *Neural Information Processing Systems*, 2022.

Xiangyu Zhao, Liang Zhang, Long Xia, Zhuoye Ding, Dawei Yin, and Jiliang Tang. Deep reinforcement learning for list-wise recommendations. *arXiv preprint arXiv:1801.00209*, 2017.

Xiangyu Zhao, Long Xia, Liang Zhang, Zhuoye Ding, Dawei Yin, and Jiliang Tang. Deep reinforcement learning for page-wise recommendations. *Proceedings of the 12th ACM Conference on Recommender Systems*, Sep 2018. doi: 10.1145/3240323.3240374. URL <http://dx.doi.org/10.1145/3240323.3240374>.

Zhuobin Zheng¹², Chun Yuan, Zhihui Lin¹², and Yangyang Cheng¹². Self-adaptive double bootstrapped ddpg. In *International Joint Conference on Artificial Intelligence*, 2018.

Tao Zhou, Zoltán Kuscsik, Jian-Guo Liu, Matúš Medo, Joseph Rushton Wakeling, and Yi-Cheng Zhang. Solving the apparent diversity-accuracy dilemma of recommender systems. *Proceedings of the National Academy of Sciences*, 107(10):4511–4515, 2010.

Brian D Ziebart, Andrew Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd national conference on Artificial intelligence-Volume 3*, pages 1433–1438. AAAI Press, 2008.

Lixin Zou, Long Xia, Zhuoye Ding, Jiaying Song, Weidong Liu, and Dawei Yin. Reinforcement learning to optimize long-term user engagement in recommender systems. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2810–2818, 2019.

Onur Çelik, Dongzhuoran Zhou, Gen Li, Philipp Becker, and Gerhard Neumann. Specializing versatile skill libraries using local mixture of experts. In *Conference on Robot Learning*, 2021.

Appendices

Appendix A

Generalization to New Actions via Action Representations

A Environment Details

A.1 Grid World

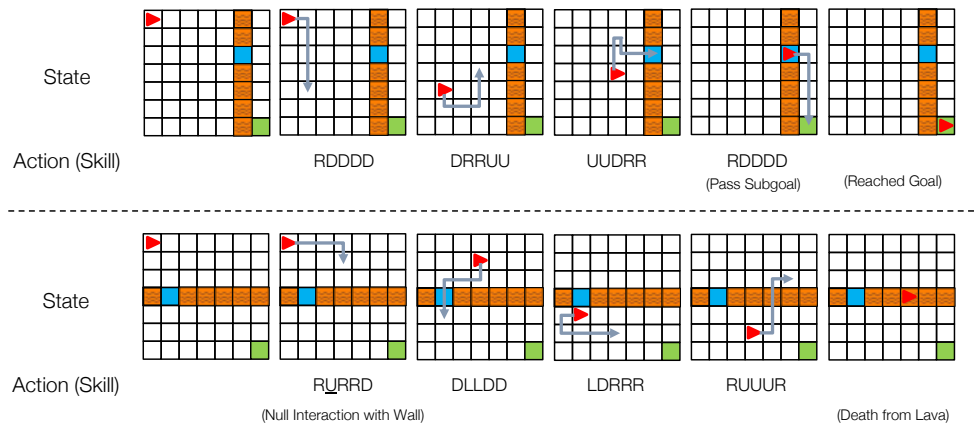


Figure A.1: Grid World Environment: 9x9 grid navigation task. The agent is the red triangle, and the goal is the green cell. The environment contains one row or column of lava wall with a single opening acting as a subgoal (blue). Each action consists of a sequence of 5 consecutive moves in one of the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

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 A.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,

$$R(s) = \lambda_{Subgoal} \cdot \mathbf{1}_{Subgoal} + \left(1 - \lambda_{Goal} \frac{N_{total}}{N_{max}}\right) \cdot \mathbf{1}_{Goal} \quad (\text{A.1})$$

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 A.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 $\mathbf{v}_{user} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. After each step in the episode, the user transitions between organic and bandit sessions, where the user vector is perturbed by resampling $\mathbf{v}_{user} \sim \mathcal{N}(\mathbf{v}_{user}, \sigma_1 \sigma_2 \mathbf{I})$, 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, $\mathbf{c} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. 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 \mathbf{c}_i is given by:

$$p_{click}(\mathbf{v}_{user}, \mathbf{c}_i) = f(\mathbf{c}_i \cdot \mathbf{v}_{user} + \mu_i), \text{ where} \tag{A.2}$$

$$f(x) = \sigma(a * \sigma(b * \sigma(c * x) - d) - e),$$

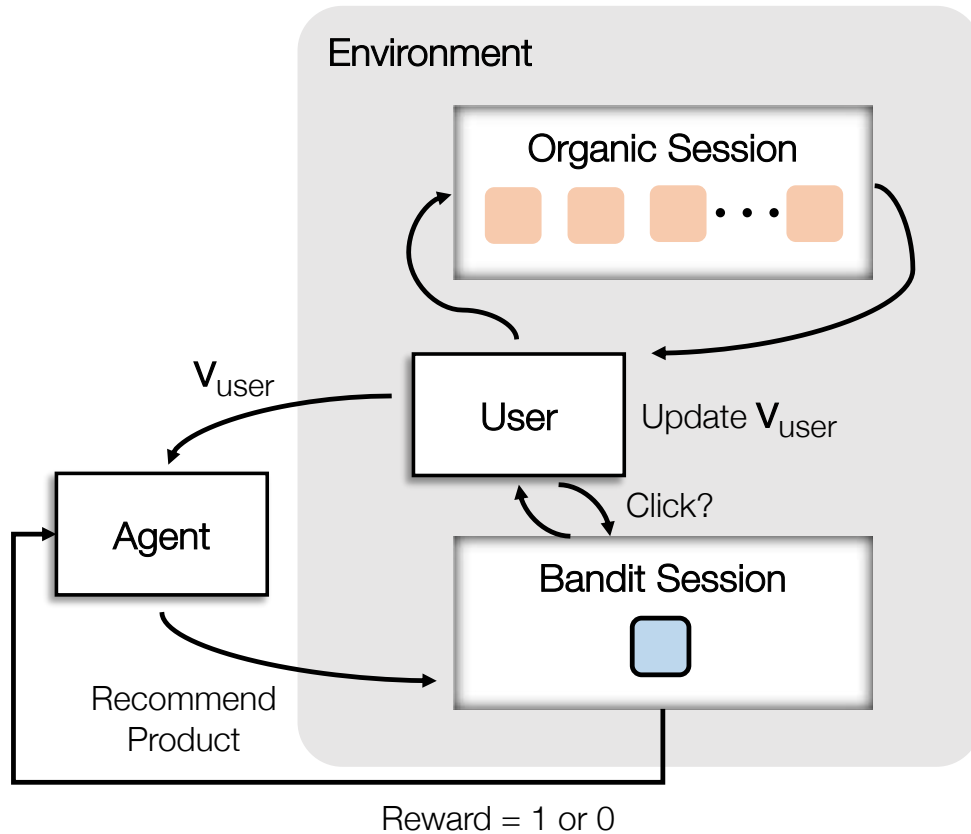


Figure A.2: Recommender System schematic: The user transitions stochastically between two sessions: organic and bandit. Each transition updates the user vector. Organic sessions simulate the user independently browsing other products. Bandit sessions simulate the agent recommending products to the current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

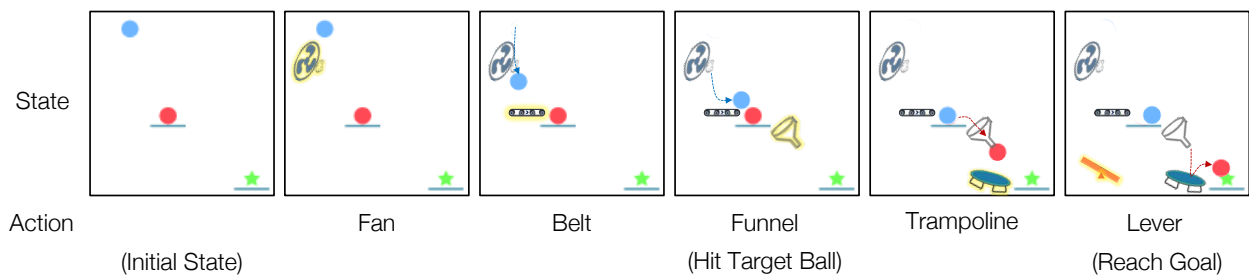


Figure A.3: CREATE Push Environment: The blue ball falls into the scene and is directed towards the target ball (red), which is pushed towards the goal location (green star). This is achieved with the use of various physical tools that manipulate the path of moving objects in peculiar ways. At every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

where $a = 14, b = 2, c = 0.3, d = 2, e = 6, \sigma$ is the 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 A.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 A.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 A.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, n.d.](#)) and Pygame physics engine ([Shinners, n.d.](#)).

CREATE environment features 12 tasks, as shown in Figure A.7. Results for 3 main tasks are shown in Figure 2.5, 2.6 and 9 others in Figure A.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 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:

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

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 A.1.

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

Table A.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 A.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:

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

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 A.2.

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

Table A.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

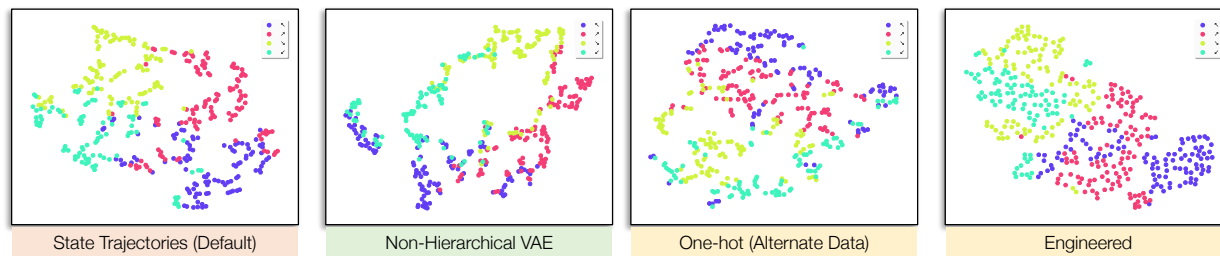


Figure A.4: t-SNE Visualization of learned skill representation space for Grid World environment. Colored by the quadrant that the skill translates the agent to.

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

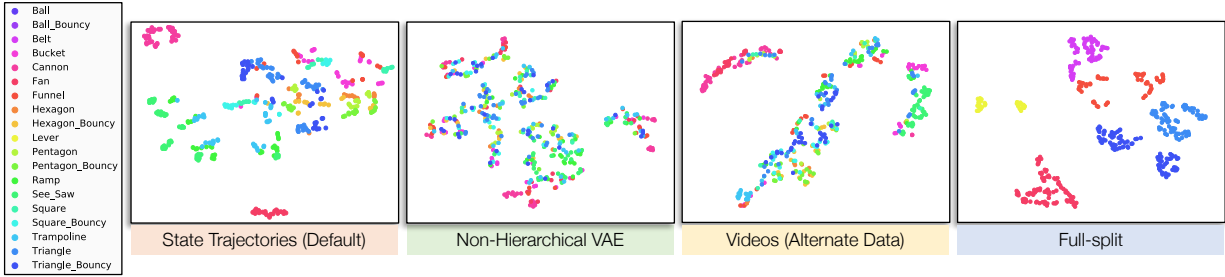


Figure A.5: t-SNE Visualization of learned tool representation space for CREATE environment. Colored by the tool class.

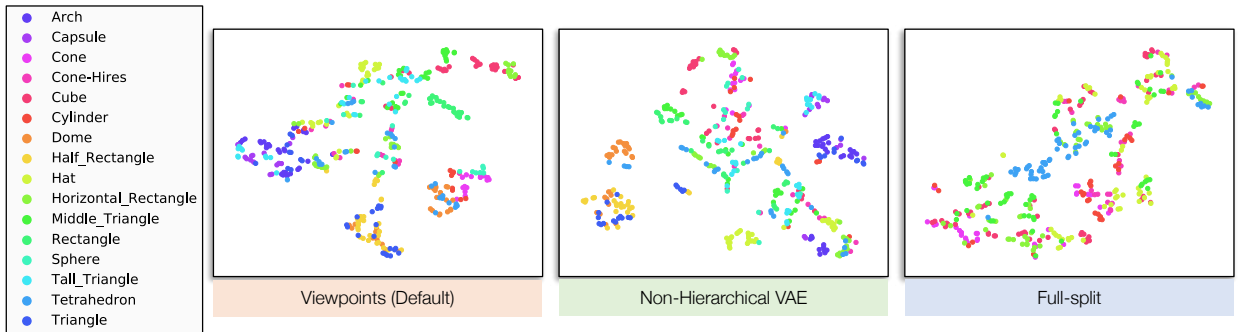


Figure A.6: t-SNE Visualization of learned action representation space for the Shape-stacking environment. Colored by the shape class.

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 A.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.

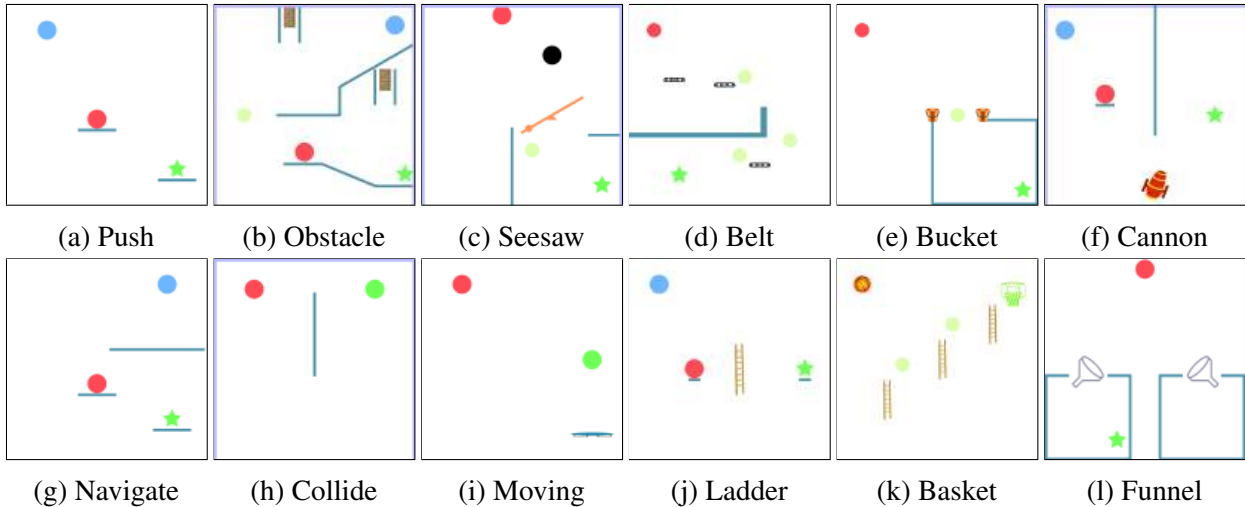


Figure A.7: 12 CREATE tasks. Complete results on (a) - (c) are in Figure 2.5, 2.6, while (d) - (l) are in Figure A.8.

- 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 comparable to these oracle representations.

CREATE: Figure A.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.

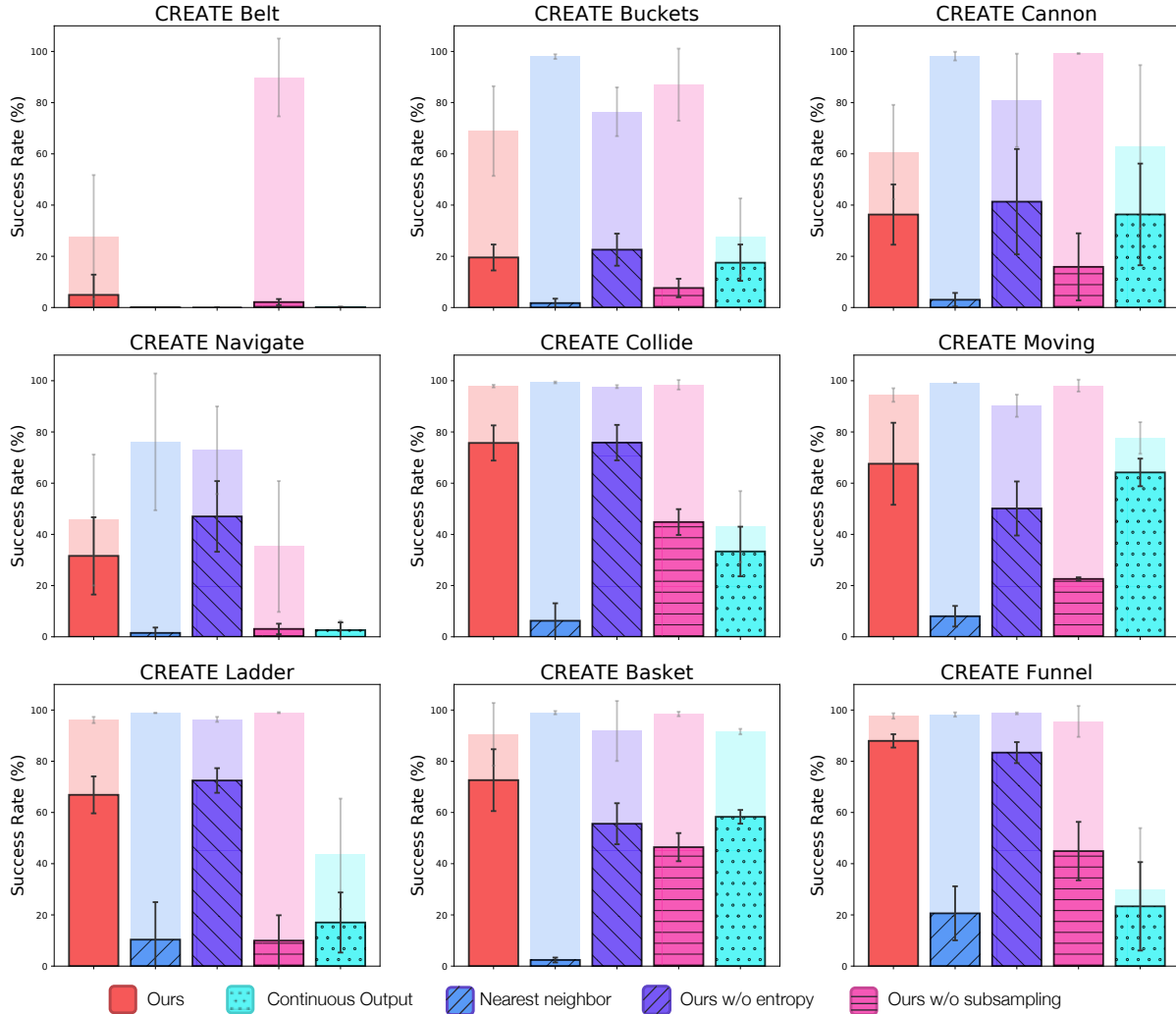


Figure A.8: Results on the remaining 9 CREATE tasks with the same evaluation details as the main paper (Figure 2.5). We compare our method against all the baselines (Section 2.5.3) and ablations (Section 2.5.4).

- **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 A.6 shows the inferred action representations in Shape Stacking. The shapes are colored according to shape class. All action representations are 128-dimensional.

- **Viewpoints (default):** HVAE encodes action observations in the form of viewpoints of the shape from different camera angles and positions.

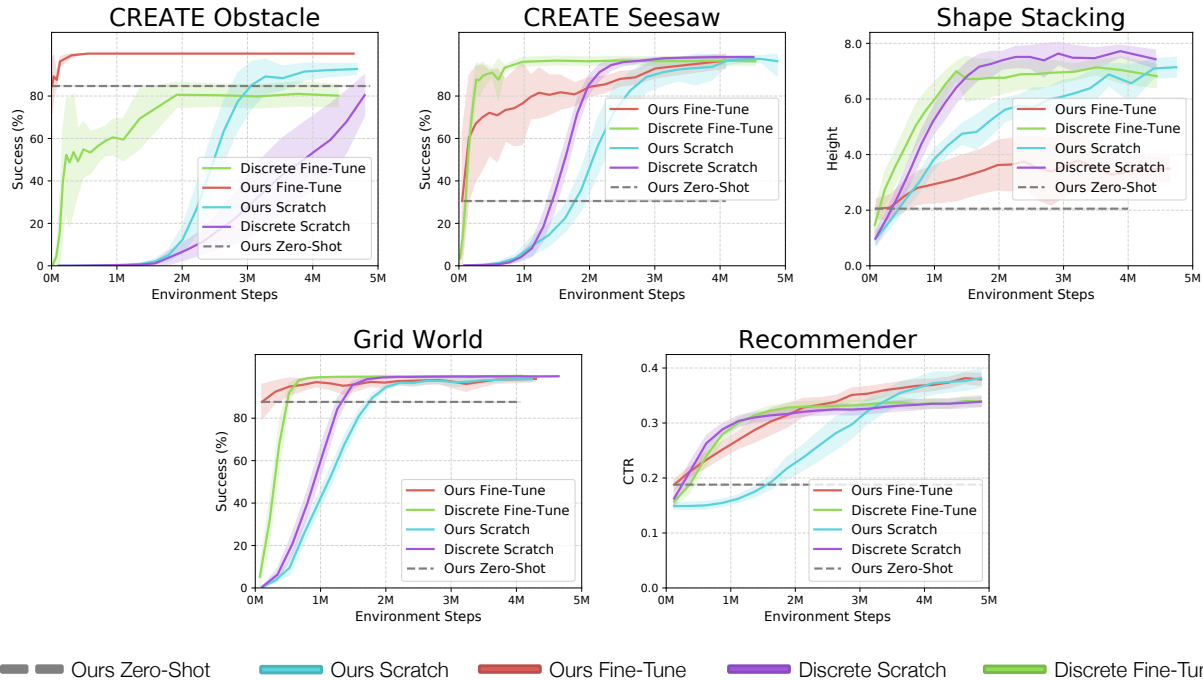


Figure A.9: Finetuning or training the policy from scratch on the new action space across the remaining 5 tasks (Figure 2.10 only shows results on CREATE Push). The evaluation settings are the same as described in Section 2.6.4.

- 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 A.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 A.8 demonstrates our method’s results on the remaining nine CREATE tasks (the initial three tasks are in Figure 2.5, 2.6). 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 A.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 A.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 A.11. There are minimal differences in the average success rates of the two design choices.

C.5 Fully Observable Recommender System

Figure A.12 demonstrates our method in a fully observable recommender environment where the product constant μ_i from Eq. A.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

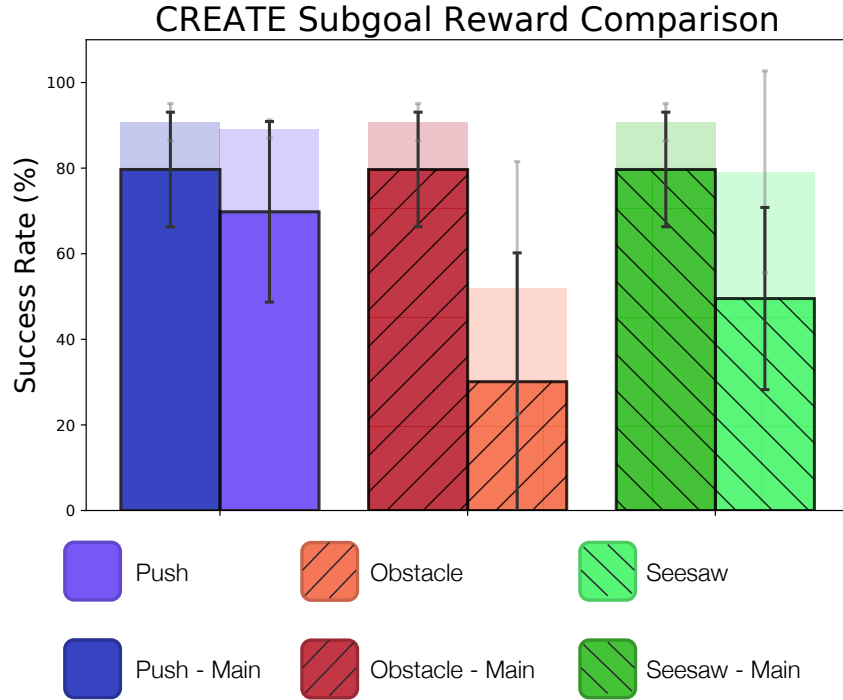


Figure A.10: Comparison of a version of CREATE that does not use subgoal rewards. The “Main” methods are from the main paper using subgoal rewards (Figure 2.5).

recommender systems. Therefore, we focus on the partially observed environment in the main results.

C.6 Additional Shape Stacking Results

Figure A.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 dropping 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>.

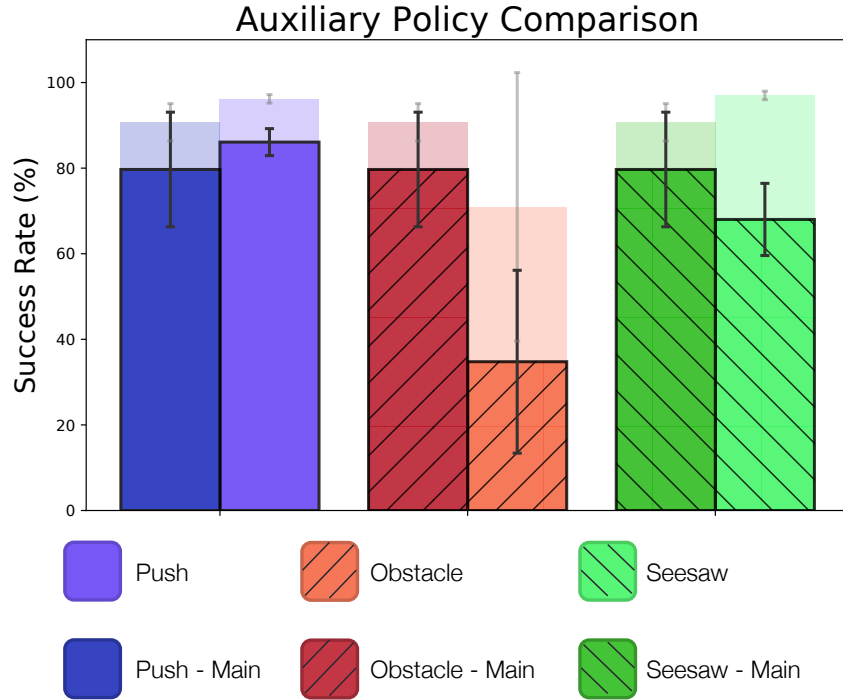


Figure A.11: Comparison of an alternative auxiliary network architecture that is conditioned on the selected discrete action. The “Main” results are the default results that do not condition the auxiliary policy on the selected action (Figure 2.5).

Figure A.13 also shows the results of our method trained and evaluated on *Full Split* which was introduced in Table A.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).

C.7 Learning Curves

Figure A.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

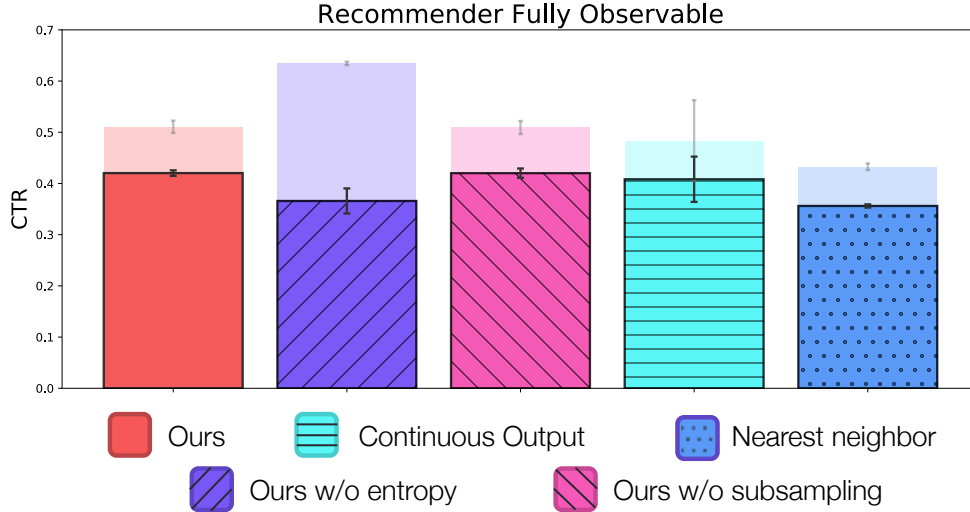


Figure A.12: Training and testing results on the fully observable version of Recommender System with standard evaluation settings.

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

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 A.3: Environment-specific hyperparameters for Grid World, Recommender, CREATE, and Shape Stacking tasks relevant to HVAE and policy.

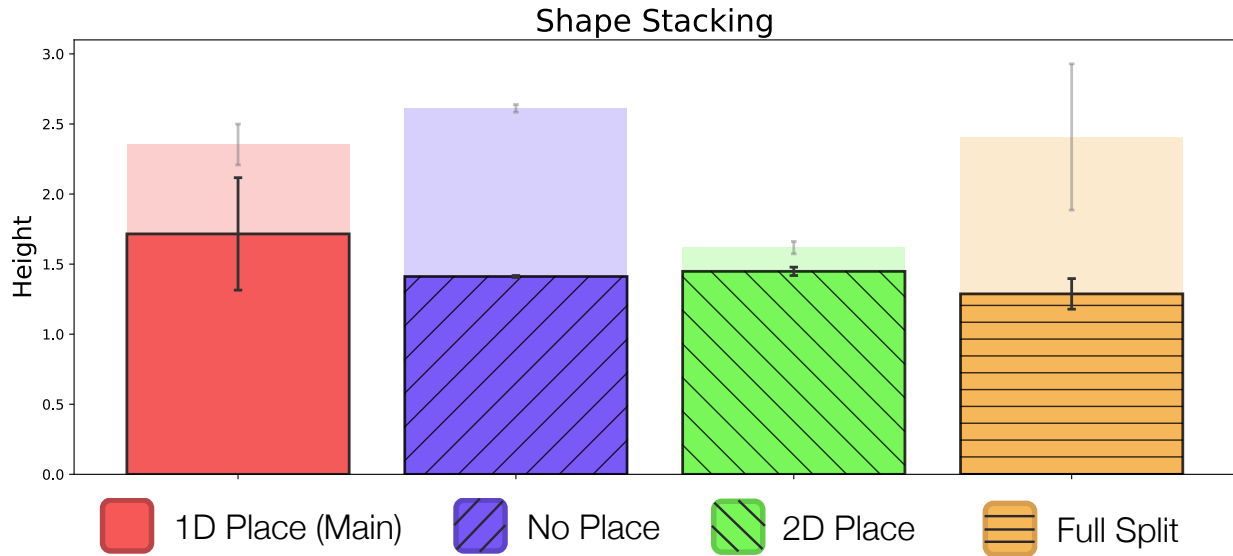


Figure A.13: Comparing different placement strategies in shape stacking and showing performance on the *Full Split* action split. Results are using our method with the standard evaluation details.

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, 2020b). All the environments were developed using the OpenAI Gym interface (Brockman et al., 2016). The HVAE implementation is based on the PyTorch implementation of Neural Statistician (Edwards and Storkey, 2017), and we use RAdam optimizer (Liu et al., 2019). For training the policy network, we use PPO (Schulman et al., 2017a; Kostrikov, 2018) with the Adam optimizer (Kingma and Ba, 2014). Further details can be found in the supplementary code.¹

¹Code available at <https://github.com/clvrai/new-actions-rl>

D.2 Hyperparameters

The default hyperparameters shared across all environments are shown in Table A.4 and environment-specific hyperparameters are given in Table A.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 A.4: General Hyperparameters for HVAE and policy shared across environments

D.2.1 Hyperparameter Search

Initial HVAE hyperparameters were inherited from the implementation of [Edwards and 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 and 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

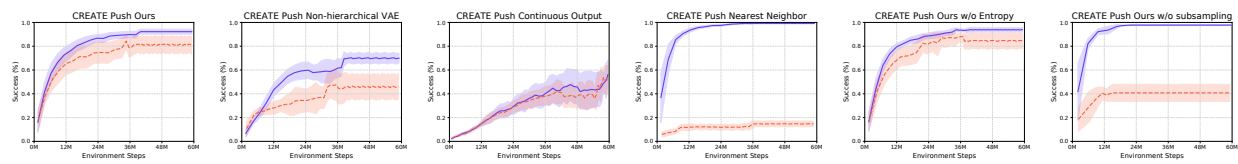
State Encoder f_ω : When the input state is in image-form (channel-wise stacked frames in CREATE and Stacking), f_ω is implemented with a 5-layer convolutional network, followed by a

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.

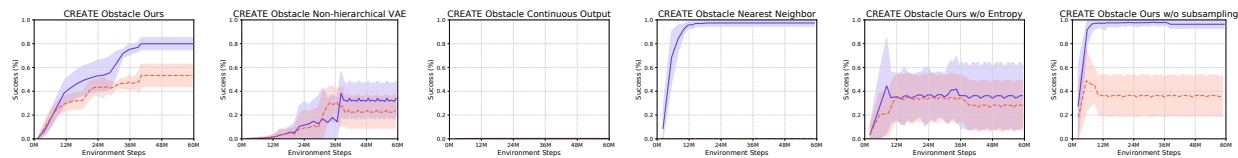
Critic Network V : For image-based states, the output of the state encoder f_ω is passed through a linear layer to result in the value function of the state. This is done to share the convolutional layers between the actor and critic. For non-image states, we use 2-layer MLP with tanh activation, followed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

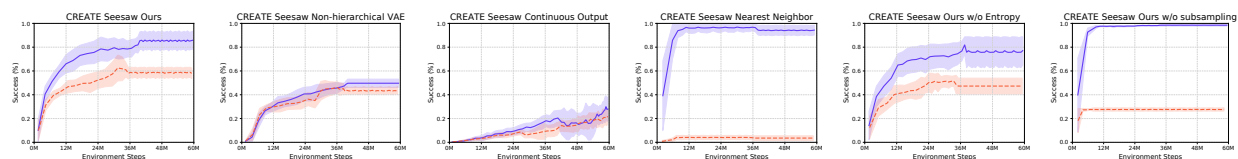
Auxiliary Policy f_χ : The output of the state encoder is also separately used to compute auxiliary action outputs. For CREATE and Shape Stacking, we have a 2D position action in $[-1, 1]$. For such constrained action space, we use a Beta distribution whose α and β are computed using linear layers over 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 $[1, \infty]$. 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.



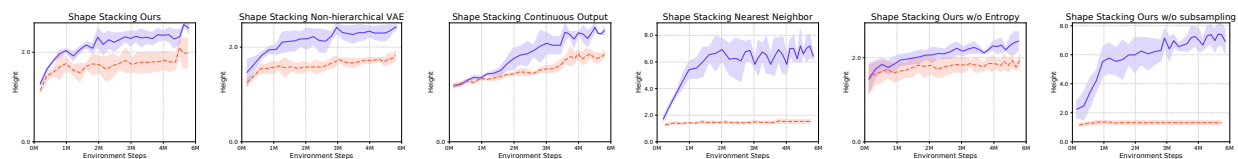
(a) CREATE Push



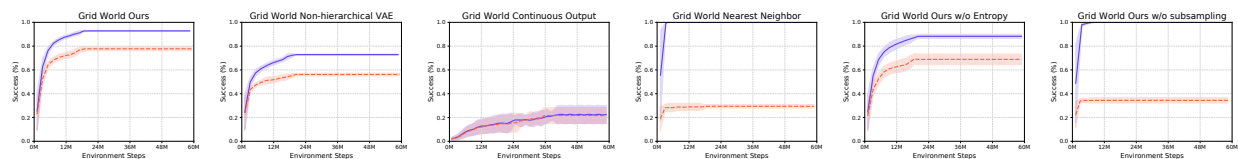
(b) CREATE Obstacle



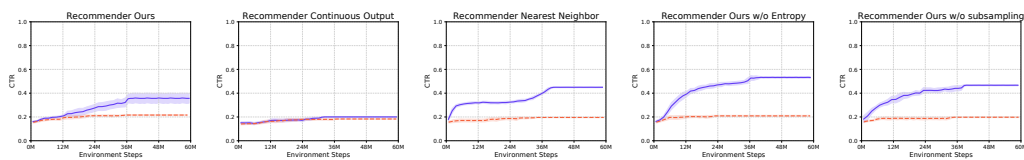
(c) CREATE Seesaw



(d) Shape Stacking



(e) Grid World



(f) Recommender System



Figure A.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.

Appendix B

Know Your Action Set in Varying Action Spaces via Action Relations

A Environment Details

A.1 Dig Lava Grid Navigation

The grid world environment, introduced in Sec. 3.5.1, requires an agent to reach a goal by navigating a 2D maze with two lava rivers using a variable set of skills.

State: The state space is a concatenation of two flattened 9x9 grids:

1. Environment: each grid element is an integer ID representing whether the cell is empty, orange-lava, pink-lava, goal, or sub-goal
2. Agent: an empty grid except for 1 at agent-coordinates.

Termination: An episode is terminated in success when the agent reaches the goal and in failure when it stays in lava for two consecutive timesteps or after a total of 50 timesteps. Note that the agent can leave a lava cell only using the dig lava skill of the corresponding color.

Actions: The base action set is a fixed set of 9 skills and in each episode, a set of 7 skills is given to the agent. 5 skills are always available: *move-right*, *move-down*, *move-left*, *move-up*, and *turn-left*. The other 2 skills are randomly sampled from a set of 4 skills: *turn-right*, *step-forward*, *dig-orange-lava*, and *dig-pink-lava*.

Reward: The agent receives a large goal reward on reaching the goal. There are two subgoal rewards for a successful crossing of each lava column (i.e. column 4 and column 8) for the first time. The goal and subgoal rewards are discounted based on the number of action steps taken to reach that location, thus rewarding shorter paths. To further encourage shorter paths, successful lava digging is rewarded. A small exploration reward is added whenever the agent visits a new cell in the episode. The exploration reward is accumulated and subtracted when the agent reaches a subgoal or a goal to ensure that the exploration reward does not hinder learning short paths. Thus,

$$\begin{aligned}
R(s, a) = & \mathbb{1}_{Goal} \cdot \left[R_{Goal} \left(1 - \lambda_{Goal} \frac{N_{\text{current steps}}}{N_{\text{max steps}}} \right) - R_{\text{Exploration}} N_{\text{steps from prev subgoal}} \right] + \\
& \mathbb{1}_{Subgoal} \cdot \left[R_{Subgoal} \left(1 - \lambda_{Subgoal} \frac{N_{\text{current steps}}}{N_{\text{max steps}}} \right) - R_{\text{Exploration}} N_{\text{steps from prev subgoal}} \right] + \\
& \mathbb{1}_{\text{Successful Dig}} \cdot R_{\text{Dig}} + \mathbb{1}_{\text{New State}} \cdot R_{\text{Exploration}}
\end{aligned} \tag{A.1}$$

where $R_{\text{Goal}} = 100$, $R_{\text{Subgoal}} = 0.5$, $R_{\text{Exploration}} = 0.01$, $R_{\text{Dig}} = 0.01$,
 $\lambda_{\text{Goal}} = 0.99$, $\lambda_{\text{Subgoal}} = 0.9$, $N_{\text{max steps}} = 50$

Action Representations: The action representations are 11-dimensional vectors manually defined using a mix of one-hot vectors, as shown in Table B.1. Dimensions 1-5 identify the category of skills (movement, elemental, dig-orange, dig-pink), 6-7 distinguish movement skills (right, down, left, up), 8-9 are always 0 (originally meant for diagonal skills), 10-11 are used to distinguish elemental skills (turn-left, turn-right, move-forward).

A.2 Chain REAction Tool Environment (CREATE)

The CREATE environment (Sec. 3.5.2) requires an agent to place tools in a physics-based puzzle to make the target ball reach the goal position. We follow all the base settings of the CREATE Push Environment from [Jain et al. \(2020\)](#). We add the functionality of new activator tools and reduce the number of available actions per episode from 50 to 25.

Category	Skill	Action Representation										
Movement	move-right	0	0	0	0	1	-1	-1	0	0	0	0
	move-down	0	0	0	0	1	-1	1	0	0	0	0
	move-left	0	0	0	0	1	1	-1	0	0	0	0
	move-up	0	0	0	0	1	1	1	0	0	0	0
Elemental	turn-left	0	0	1	0	0	0	0	0	0	-1	1
	turn-right	0	0	1	0	0	0	0	0	0	1	-1
	move-forward	0	0	1	0	0	0	0	0	0	1	1
Digging	dig-orange	0	1	0	0	0	0	0	0	0	0	0
	dig-pink	1	0	0	0	0	0	0	0	0	0	0

Table B.1: Action representations for the skills used in Dig Lava Grid Navigation environment.

State: The agent receives the past three gray-scale frames, which are stacked channel-wise to make a $84 \times 84 \times 3$ input.

Termination: The episode ends in success when the agent accomplishes the goal and in failure when there is no remaining movement in the game or after 30 timesteps.

Actions: Each original tool from CREATE is now associated with newly added *activator tools* as shown in Figure B.1. Thus, we add 5 activator tools to the 2110 general tools, which are variations in angle, size, friction, or elasticity of the tools shown in Figure B.1. Unlike Jain et al. (2020), we remove the No-Operation action. Activator tools are pass-through tools and only serve the function of activating their corresponding general tools when placed in contact.

We split the general tool space into 1098 tools for training, 507 tools for validation, and 507 tools for testing. All 5 activator tools are available for sampling during training, validation, and testing. In each episode, 23 general tools and 2 activator tools are randomly sampled and made available to the agent. The agent outputs in a hybrid action space consisting of (1) the discrete tool selection and (2) (x, y) coordinates of tool position.

Reward: We adopt the reward structure from Jain et al. (2020). CREATE Push is a sparse reward environment with rewards for reaching the goal and making the target ball move. There is an additional *alive* reward to continue the episode. Placements outside the scene are penalized. The original CREATE environment penalizes overlapping tools. However, since overlapping activators and general tools is required for solving tasks, we modify the overlap penalty to apply only when






















 Spring	 Trampoline (TRP)	 Bouncy Ball (BBL)	 Bouncy Box (BBX)	 Bouncy Polygons
 Magnet	 Ramp (RMP)	 Fixed Ball (FBL)	 Fixed Box (FBX)	 Fixed Polygons
 Fire	 Cannon (CNN)	 Hinge or Lever (HNG)	 Hinge Constrained (HGC)	 See-Saw (SS)
 Water	 Funnel (FNL)	 Bucket (BKT)		
 Electric	 Fan (FAN)	 Belt (BLT)		

Figure B.1: **CREATE Activator Mapping**: Original CREATE tools (right) are activated by the respective newly introduced activator tools (left). E.g., a *Cannon* tool (abbreviated as CNN in attention maps) placed on the environment will only be functional when a *Fire* tool is placed in contact with it. Other objects are not affected by non-functional tools - they simply pass through.

either both are general tools or both are activator tools. Thus, the agent receives the following reward:

$$R(s, a) = R_{\text{alive}} + \mathbb{1}_{\text{Goal}} \cdot R_{\text{Goal}} + \mathbb{1}_{\text{target hit}} \cdot R_{\text{target hit}} + \mathbb{1}_{\text{invalid}} \cdot R_{\text{invalid}} + \mathbb{1}_{\text{overlap}} \cdot R_{\text{overlap}} \quad (\text{A.2})$$

where $R_{\text{alive}} = 0.01$, $R_{\text{Goal}} = 10.0$, $R_{\text{target hit}} = 1$, and $R_{\text{invalid}} = R_{\text{overlap}} = -0.01$.

Action Representations: Each action representation is a 134-dimensional vector, a concatenation of two 128-D and 6-D vectors, as shown in Table B.2. The 128-D vector corresponds to the learned characteristics of the tool, and the 6-D vector is a binary vector denoting whether the tool is a general tool, an activator tool, or a no-op tool (no-op is disabled for experiments).

For general tools, [Jain et al. \(2020\)](#) obtain action representations using a Hierarchical VAE. They encode a set of tool observations into a latent representation. Each tool is made to interact with a probing ball launched from various angles, positions, and speeds. The tool characteristics can be inferred from this collection of tool interactions. We utilize these 128-D learned tool representations from [Jain et al. \(2020\)](#) for the general tools and pad them with a 6-D zero-vector. Generalization to unseen tools is possible because a trained agent can utilize the 128-D tool embedding to extract the relevant characteristics of any given tool. The agent must infer which activator a general tool is associated with using its tool embedding.

For activator tools, the first 128 dimensions are always zero, and the final 6 dimensions correspond to a one-hot vector, each for {no-op, Fire, Water, Electric, Magnet, Spring}.

Tool	Action Representation						
General Tools	Learned 128-D tool characteristics	0	0	0	0	0	0
No-Op	128-D Zero Vector	1	0	0	0	0	0
Fire	128-D Zero Vector	0	1	0	0	0	0
Water	128-D Zero Vector	0	0	1	0	0	0
Electric	128-D Zero Vector	0	0	0	1	0	0
Magnet	128-D Zero Vector	0	0	0	0	1	0
Spring	128-D Zero Vector	0	0	0	0	0	1

Table B.2: Action representations for the tools in CREATE environment.

A.3 RecSim

The simulated RecSys environment (RecSim in Sec. 3.5.3), requires an agent to select a list of items that match the user’s interest out of a variety of recommendable items. We simulate users that have a preference over high-CPR lists (Sec 3.5). The agent’s task is to infer this preference from user clicks and recommend a list of items to optimize both user interest and CPR.

State: The state is represented by the user interest embedding ($e_u \in \mathbb{R}^n$ where n denotes the number of categories of items) in categories that transitions over time as the user consumes different

items upon click. So, when the user clicks an item with the corresponding item embedding($e_i \in \mathbb{R}^n$) then the user interest embedding(e_u) will be updated as follows,

$$\begin{aligned}\Delta(e_u) &= (-y|e_u| + y) \cdot (1 - e_u), \text{ for } y \in [0, 1] \\ e_i &\leftarrow e_u + \Delta(e_u) \text{ with probability } [e_u^T e_i + 1]/2 \\ e_u &\leftarrow e_u - \Delta(e_u) \text{ with probability } [1 - e_u^T e_i]/2\end{aligned}$$

This essentially pulls the user’s preference towards the item that was clicked.

Action: The base action set is a set of 500 items (250 for each train and test action set), and in each episode, a sampled subset of size 20 is given to the agent. To simulate the varying action space environment, we implemented the **most common category sampling** method to form the candidate-set. Here, the majority of items are sampled from one common category, and the remaining items are sampled from other categories. In this way, identifying the most common category is crucial to recommend a coherent list of items. Note that if we just sampled items uniformly across all categories, then CPR maximization would be less interesting as no single category has the potential to fill the entire list of items (i.e., achieve maximum CPR).

Reward: The base reward is a simulated response (e.g., clicks) from users (the user model (Ie et al., 2019a) stochastically skips or clicks an item in the list based on the user interest embedding). To better simulate the realistic scenario of user preference being affected by CPR of a presented list, we implement additional features in the user model:

$$\begin{aligned}\text{score}_{item} &= \alpha_{user} * \langle e_u, e_i \rangle + \alpha_{metric} * m \\ p_{item} &= \frac{e^{\text{score}_{item}}}{\sum e^{\text{score}_{item}}} \\ R &= f_{\text{click_or_skip}}(p_{item})\end{aligned}$$

where, $e_u, e_i \in \mathbb{R}^n$ are the user and item embedding, respectively, $\langle \cdot, \cdot \rangle$ is the dot product notation and $\alpha_{user}, \alpha_{metric}, m \in \mathbb{R}$ where m denotes the list-metric (e.g., CPR-score). So, given the score score_{item} of an item, the user model computes the click likelihood through a softmax function over

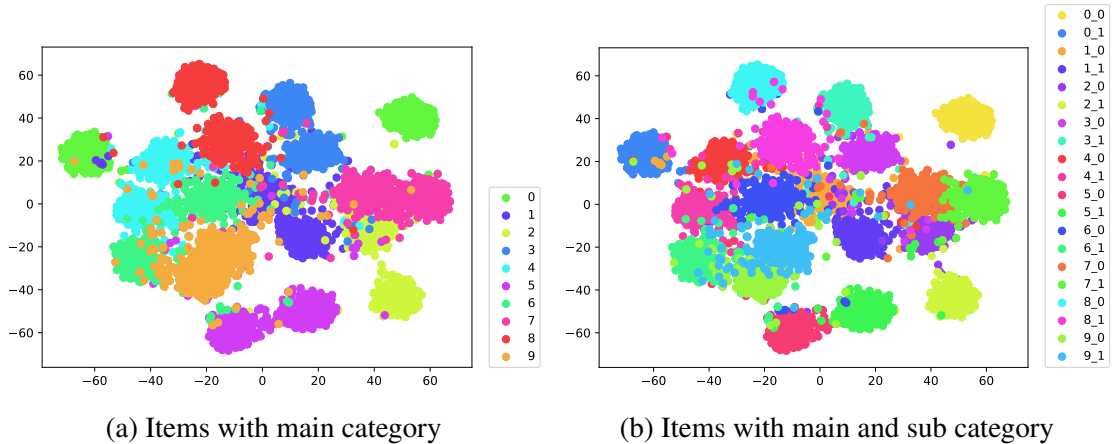


Figure B.2: t-SNE visualization of synthetically generated items in RecSim.

all items and a predefined skip-item followed by a categorical distribution ($f_{\text{click_or_skip}}$). It takes the computed likelihood and outputs either click (reward=1) or skip (reward=0) as user feedback.

Action Representations: Originally, [Ie et al. \(2019a\)](#) use the discrete representation of items based on the one-hot encoding of item-category. However, this does not support generalization over items (actions). Therefore, we implement continuous item representations sampled from a Gaussian Mixture Model (GMM) with centers around each item category. Each item category has two sub-categories for items, which are also clustered. This ensures that the action representation contains information about the primary and sub-categories. Fig. B.2 shows the t-SNE visualization of action representations for the 500 items, based on a GMMs. There are 10 main categories that each have two sub-categories. Figure B.2a shows item representations labeled according to the primary category. Figure B.2b shows item representations clustered according to sub-category.

A.4 Real-data recommender system

The real-data RecSys environment (Sec. 3.5.3) requires an agent to select a list of items that match the user’s interest out of a variety of recommendable items. We experiment with the scenario in which domain engineers want an RL agent’s list-actions to conform to user preference while optimizing a listwise metric. Since having a high CPR is correlated with better user response ([Hao et al., 2020](#)), we use CPR as the additional listwise metric. Thus, we reward the agent based on (i)

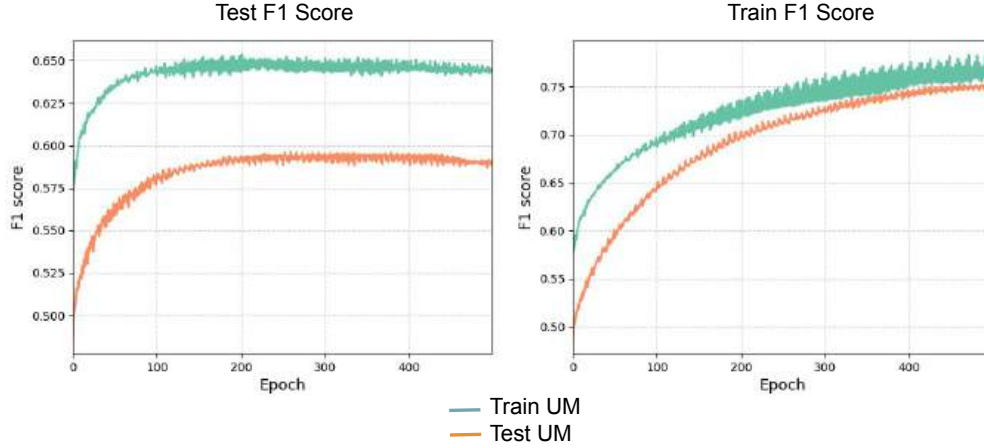


Figure B.3: Real-Data Recommender System: F1 Score for Training/test user models. (Left) The online user model is trained on online-training data and evaluated on both online-training (green) and online-held-out (orange) data. (Right) The offline user model is trained on offline-training data and evaluated on both offline-training (green) and offline-held-out data (orange). Thus, the disparity between online data training and evaluation curves (left) shows that it is hard to train the online user model. In contrast, the offline user model generalizes reasonably well (right).

user models trained from real data and (ii) CPR of the recommended list of items. We collected the dataset from two different periods, two weeks in late August in 2021 as the offline period (used for training) and the following two weeks from early September in 2021 as the online period (used for evaluation).

State: The state is a concatenation of a sequence of historical user interactions. Concretely, a set of item representations (32 dimensional real vectors) of the three most recent clicked items (i.e., 32×3 matrix) are appended individually to the user attributes (137 dimensional real vectors) such as age, occupation, and localities. Therefore, the state is in the form of $507 = 169 \times 3 = (32 + 137) \times 3$. To act optimally, the agent must extract the useful representation of the user preference through this historical observation.

Action: The base action set is a set of 85 items, and in each episode, a sampled subset of size 20 is given to the agent. We employed the same sampling methods as in Sec. A.3. So, the agent needs to select a list of 6 items given the sampled candidate set of 20 items.

Reward: The base reward is a simulated response (click or skip) from the user model trained from real-world data. We add the CPR metric (between 0 to 1) to this click reward (1 or 0). The agent’s objective is to optimize user preference and the CPR of the list of items it recommends.

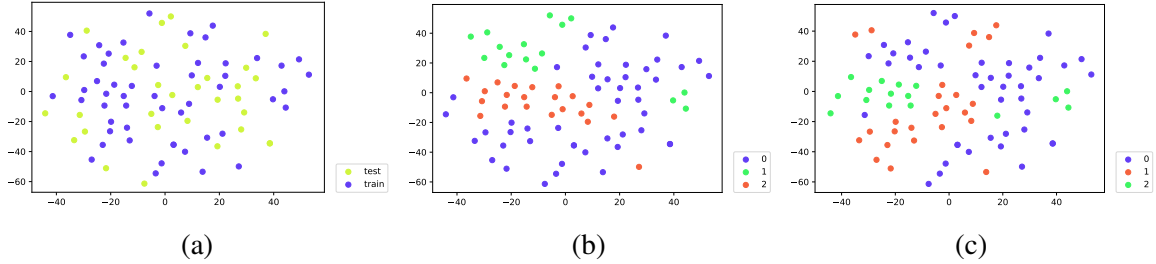


Figure B.4: (a) t-SNE visualization of split of train and test item representations. This shows that the train and test items are within the same distribution (b) Visualizations of item representations clustered according to the main category. This shows that item representations contain information about the main category, which is necessary to maximize CPR. (c) Item distribution labeled according to sub category, which is another information necessary for CPR.

Training of User models: We follow the two different periods (offline and online) in the data extraction procedure. Thus, we trained two different user models ($f_{user} : \mathcal{S} \times \mathcal{A}^{\text{list-size}} \rightarrow \mathbb{R}$) to (i) train the agents offline and (ii) evaluate them with online users. The result of training those user models can be found in Figure B.3 (a). Thus, the reward is computed as follows; $R = f_{user}(e_{user}, e_{item}) + m$ where f_{user} is either the offline user model or the online user model that provides us with the simulated response of users (e.g., click or skip). The user model architectures are described in Sec. C.3.5.

Action Representation: In the previous work CDQN (Chen et al., 2019a), the authors found it useful to characterize the items by wide and deep features. For example, their movie recommendation task considered the text description as the wide feature and the movie category as the deep feature. And they utilized the Wide and Deep Network (Cheng et al., 2016) to get useful representations of items. Following their work, we used the wide and deep network to pretrain the item embedding given the rich item features in the real-world data. However, we empirically observed that employing VAEs for each wide and deep feature of items leads to the better-segregated representation of items. Therefore, given an item instance, we separate the raw item attributes into the deep attributes (e.g., reward points of campaigns) and the wide attributes (e.g., text description), which are then fed into the VAE based wide and deep network to get the compressed representation by combining the wide and the deep features together. See Sec C.3.4 for more details about the network architecture. Thus, each action representation is a 32-dimensional vector encoded by a

VAE based Wide and Deep network. The learned representations of items are visualized based on the train-test split, distribution of the main category, and distribution of sub-category in Figure B.4.

B Further Experimental Results

B.1 Effect of using domain knowledge in action graph edges

To show how to incorporate domain knowledge about action relations into the action graph (discussed in Section 3.4), we use the Dig Lava Grid World environment. Specifically, we use the knowledge that directional actions are always available. Thus, the only relevant action relations are the ones with the variable actions, i.e., 2 out of 4 actions (including both dig-lava skills). So, while building the action graph, we only keep connections to and from the 2 variable actions instead of having a fully-connected structure. The rest of the AGILE architecture and algorithm stays the same. This reduces the number of bidirectional edges from $7^2 = 49$ to 29 by removing 20 edges between the always available actions. As shown in Figure B.5, the learning speed of AGILE is slightly accelerated, and the seed variance has reduced. We expect domain knowledge to help efficiency even more, when the action space is large since the edges scale quadratically.

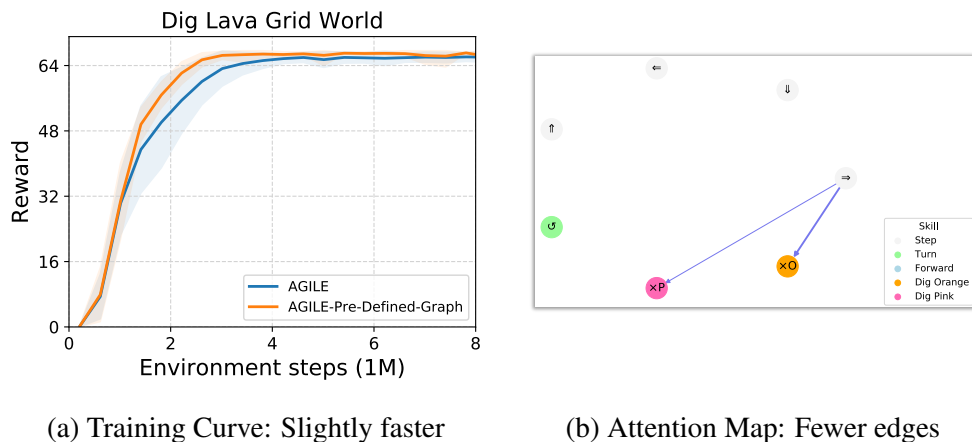
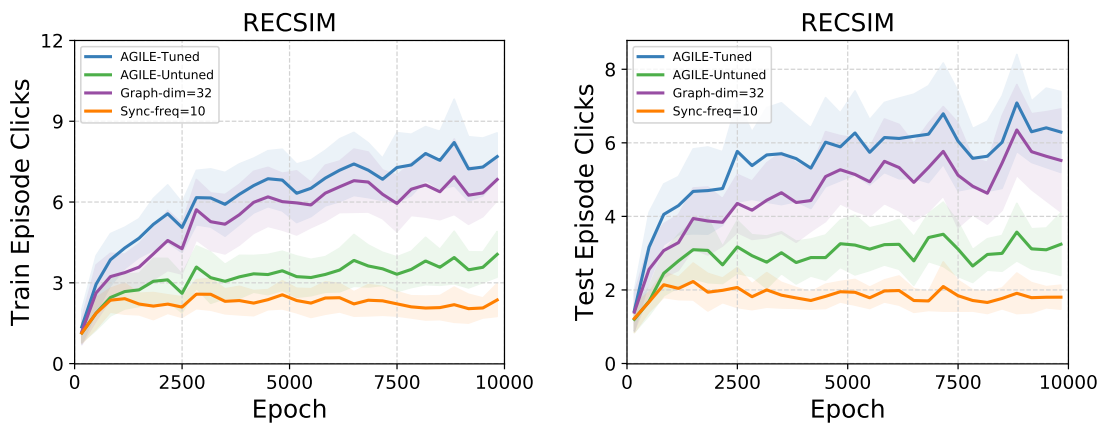
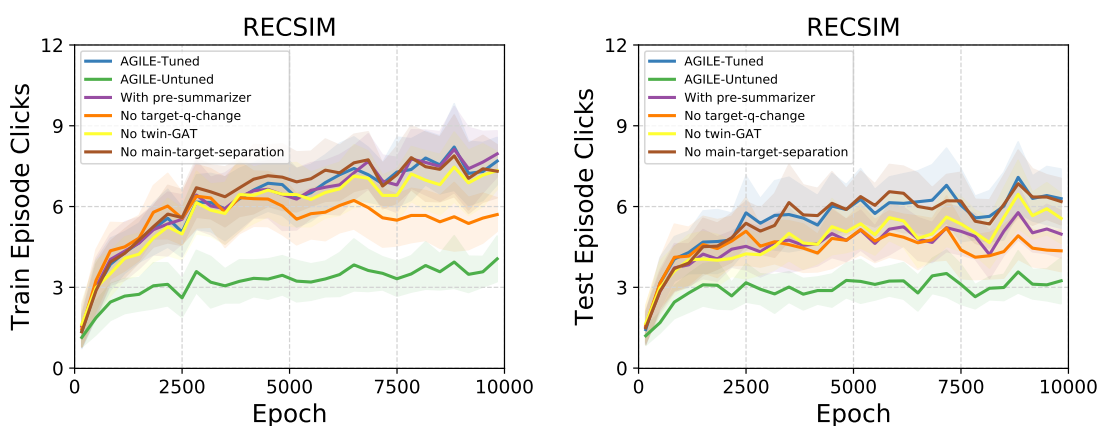


Figure B.5: Effect of using domain knowledge to predefine the possible relations via edges in the action graph of AGILE. We know that in the Grid Navigation task, only the action relations with respect to variable actions are important. Thus, we remove all the other edges. This makes the learning slightly faster and more stable as shown by a reduction in seed variance. (5 seeds)



(a) Hyperparameter search



(b) Design choices

Figure B.6: Results of value-based AGILE on RecSim CPR task (a) hyperparameter testing and (b) validating architecture design choices, on train actions (left) and test actions (right).

B.2 Validating design choices for value-based AGILE

We conducted an exhaustive search on the architecture of value-based AGILE from two different perspectives; (a) Hyper-parameters and (b) Architectures. Note that the same hyper-parameter search procedure of this section was applied to all other methods, and the same trend of the improvement in AGILE was found for other methods. In this section, AGILE used in the main results (Fig. 3.4, 3.5, 3.7) is called *AGILE-Tuned*. The barebone version of *AGILE-Tuned* is called *AGILE-Untuned*. We illustrate how each change contributes to an improvement in performance.

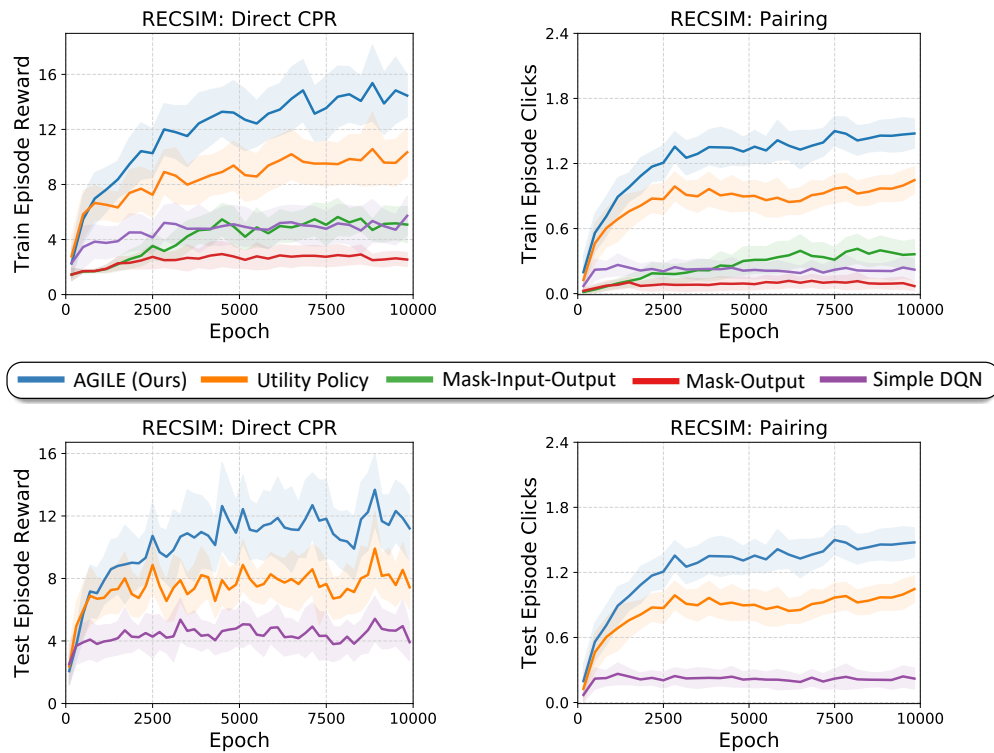


Figure B.7: Comparison against the baselines on train (top) and test (bottom) actions on the Direct CPR (left) and Pairing (right) RecSim environments. Along with Figure 3.4, these results exhibit the same trend that AGILE consistently outperforms all the baselines on both train and test actions.

B.2.1 Hyper-parameter Search in AGILE

- **AGILE-Tuned without sync-freq-change:** In Mnih et al. (2015), the authors used the periodic syncing between the target and the main networks to alleviate the issue of frequently moving Q-value targets. In this work, we compare two extreme cases of the sync frequency: 10 depicted by $Sync-freq=10$ in Fig. B.6 (a) and 500 depicted by *AGILE-Tuned*.
- **AGILE-Tuned without graph-dim-change:** To understand the difficulty in expressing the action relations through a compact representation, we compare two hidden dimension sizes. The node-features are encoded in 32 ($Graph-dim=32$) or 64(*AGILE-Tuned*) dimensions.

Figure B.6(a) shows the result on both train and test actions. *AGILE-Tuned* outperformed all the methods. $Graph-dim=32$ is slightly worse than *AGILE-Tuned* and $Sync-freq=10$ fails to learn anything meaningful. Thus, frequently moving target network harms the agent’s performance while sufficient expressiveness in the action graph improves the performance.

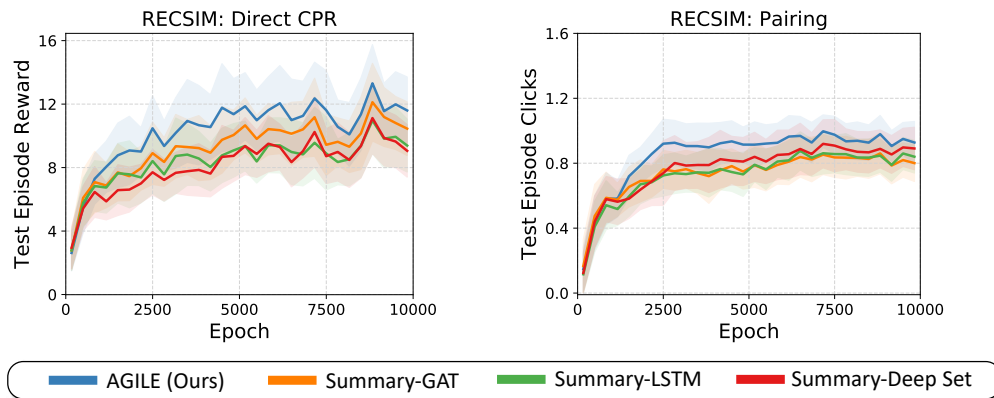


Figure B.8: Comparison against ablations on the Direct CPR (left) and Pairing (right) RecSim environments. Along with Figure 3.5, these results exhibit the same trend that AGILE slightly outperforms the various summary ablations, which do not explicitly utilize relational action features and rely only on the summary vector to encode all the necessary action relations.

B.2.2 Design Choices of AGILE

- **AGILE-Tuned with pre-summarizer:** In *AGILE-Tuned*, the concatenation of an action representation and the state is a node feature input to the GAT (Sec.C.3.1)). Here, we experiment and observe that adding a 2-layer MLP with ReLU over the node features does not help performance.
- **AGILE-Tuned without target-q-change:** [Chen et al. \(2019a\)](#) compute the target q-values for training CDQN using the list-action at the next time-step. But, there is another potential target q-value from the next item in the current list-action. Here, we compare these two methods to get the target q-value: (a) *intra-list(AGILE-Tuned)*: the target q-value is from the next list index. (b) *across-list (No target-q change)*: the target q-value is from the next timestep.
- **AGILE-Tuned without twinGAT-change:** In AGILE, the GAT output provides the utility network with the relational action representation and the action summary. Here, we compared two options: (a) *Sharing GAT (No twin-GAT)*: there is a single GAT working to provide both of them. (b) *Non-sharing GAT (AGILE-Tuned)*: this employs two different GATs for each.
- **AGILE-Tuned without main/target-encoder-separation-change:** In the implementation of AGILE, we compared two different architectural decisions. (a) *No main-target separation*: Separate the list-action encoder in CDQN for main and target Q-networks. (b) *AGILE-Tuned*: share the same list-action encoder for main and target Q-networks.

Figure B.6(b) shows the result on both train and test actions. *AGILE-Tuned, With pre-summrizer*, and *No twin-GAT* showed the similar performance which is better than *No target-q-change*. The difference between *AGILE-Tuned* and *No target-q-change* is that that the cascaded network in *AGILE-Tuned* uses the target q-value from the intermediate list constructed. This is a more accurate target q-value as compared to the target q-value from another list from a future time-step.

B.3 Effect of direct v/s indirect reward in RecSim

In Fig.3.4, we studied the learning capability of AGILE under the action interdependence on RecSim in which the CPR itself was not directly visible to the agents. Therefore, agents need to indirectly understand how well the list-action is through user feedback (i.e., clicks). In this section, we strengthen our results by examining the consistency of results. The CPR is made directly visible to all the agents in this setting. So, we implemented an additional environment in RecSim where the reward is a sum of the click reward and the CPR metric. We call this environment as *Direct CPR*. Figure.B.7 shows the comparison of AGILE against the same baseline agents as in Fig.3.4 on the Direct CPR RecSim. And in Fig.B.8 and Fig.B.9, we got the consistent result that AGILE slightly outperforms the ablations. AGILE outperforming the baselines shows that the knowledge of dependence on other available actions is crucial for optimal policy. This result is consistent with the indirect CPR optimization setting in Fig.3.4. So AGILE can perform well in the direct maximization of CPR.

B.4 Recsim-Pairing Environment

In the experiment of Fig. 3.5, we found that in RecSim, the relation of items is easy to model such that AGILE could not outperform the ablations. In contrast, AGILE outperformed the ablations in CREATE and Grid World by correctly utilizing the action relation in decision-making. We hypothesize that these environments require complex relations between actions (e.g., tools and activators in CREATE). To this end, we implement the pre-defined pairings among items in RecSim such that clicks can only happen when the correct pairs of items are recommended. Since action

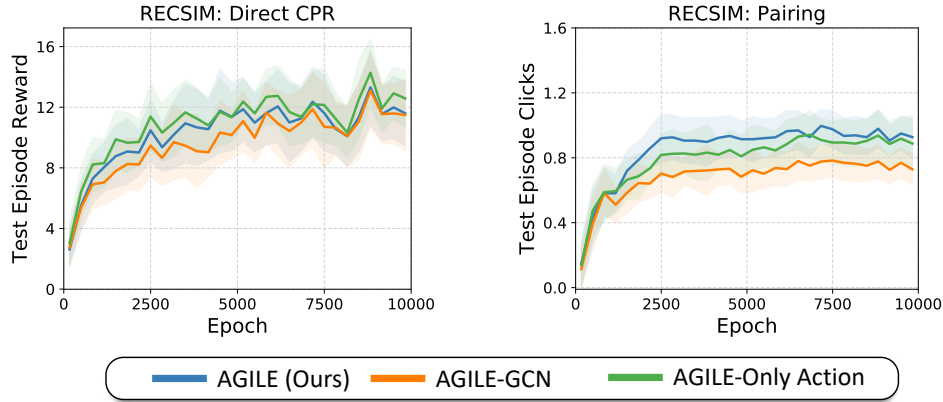


Figure B.9: Analyses of (i) GAT v/s GCN and (ii) state-action graph v/s action-only graphs on (left) RecSim: Direct CPR environment and (right) RecSim: Pairing environment.

relations are complex, AGILE is expected to outperform the ablations. Figure B.7 shows that AGILE beats the baselines and in Fig.B.8 AGILE slightly but consistently outperforms the ablations. In Fig.B.9, AGILE outperforming AGILE-GCN shows that a GAT is capable of modeling the action relations correctly. AGILE converges faster than AGILE Only-Action. This shows that the state and the partially constructed list are crucial to learning to attend the other half in pairing items efficiently.

C Approach and Baseline Details

C.1 Details of Baselines and Ablations

Like AGILE, all baselines and ablations receive the state and action representations as input and output a Q-value or a probability distribution over available actions. Figure B.10 describes the architectures of all the methods. Here, we discuss how each baseline and ablation is different from AGILE and why AGILE is expected to outperform them:

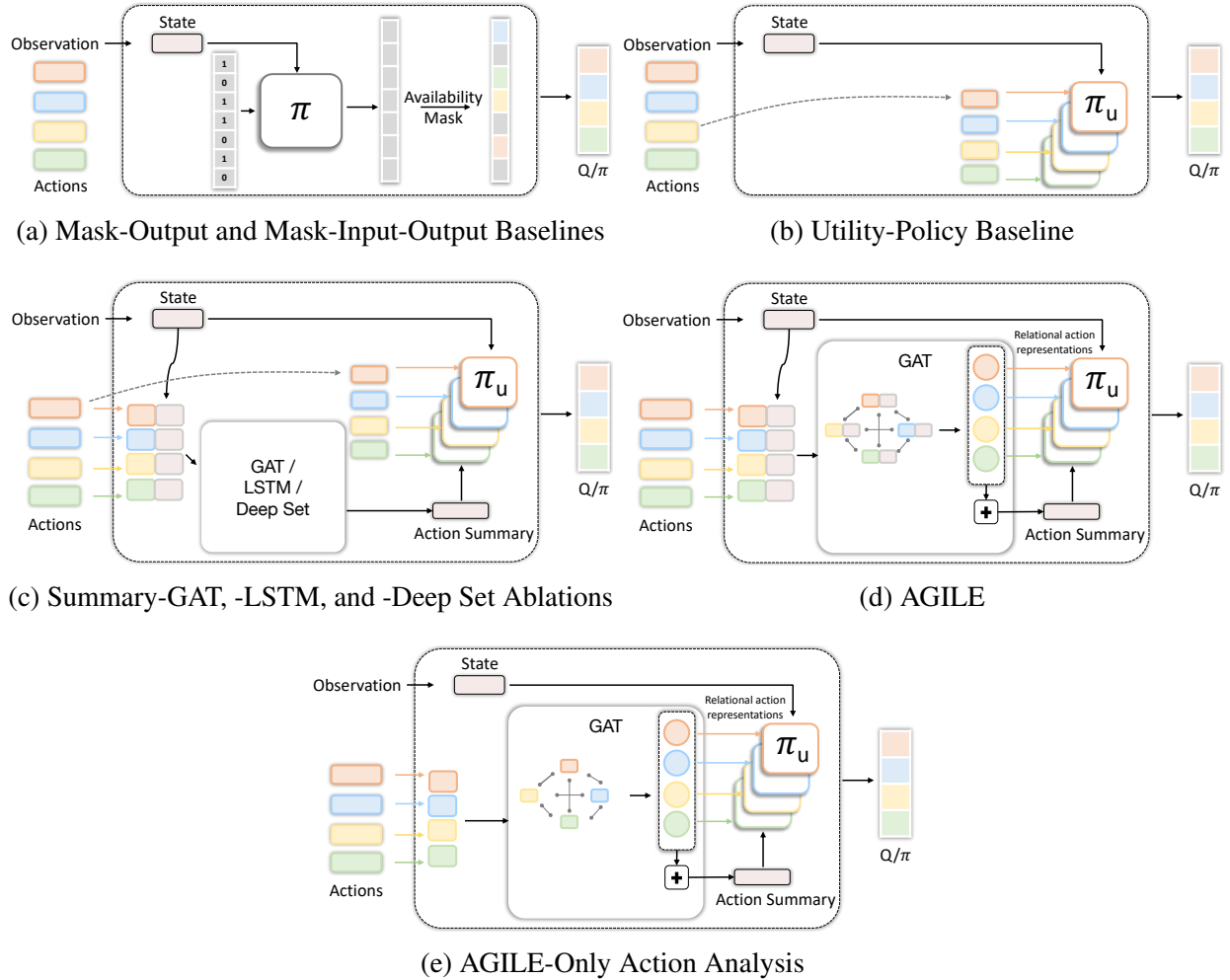


Figure B.10: Architectures of all methods used as baselines (Sec. 3.6.1), ablations (Sec. 3.6.1), and analyses (Sec. 3.6.3). (a) Following prior work in SAS-MDPs (Boutillier et al., 2018; Chandak et al., 2020a) and invalid action masking (Huang and Ontaño, 2020; Ye et al., 2020; Kanervisto et al., 2020), the **mask-output** baseline masks out the unavailable actions assuming a known action set. The **mask-input-output** baseline additionally augments the state with the action availability mask. (b) Utility-policy (Jain et al., 2020) can generalize over actions by using action representations. However, it computes each action utility independent of other available actions. (c) **Summary Ablations** augment the utility-policy with an extra action-summary input, which is a compressed version of the list of available action representations. This compression can be done by mean-pooling over a *Bi-LSTM* output layer, or a *deep set*, or a graph network (*GAT*) processed node features. (d) **AGILE (Ours)** uses a *GAT*’s node features both to compute an action set summary and as relational action features replacing the original action representations. While the action set summary is a compact representation of the available actions, it does not sufficiently scale when there are many actions or the task requires many action relations for each action decision. (e) **AGILE-Only Action** is the version of AGILE where the state is not used to compute action relations in *GAT*. This is a simpler architecture to learn but is not expected to work for certain tasks where action relations change depending on the state.

- **Mask-Output** (No representations, No input action set): This baseline assumes a fixed-action space that is known in advance. Since it does not use action representations, it cannot generalize to unseen actions or exploit the structure in action space. Moreover, it does not take the available action set as input and thus cannot solve tasks where action decisions require knowledge of other actions.
- **Mask-Input-Output** (No representations): By augmenting a binary availability mask of given actions to the state input of *Mask-Output*, this method can utilize the information about the available action set as a set. However, the input availability-mask is fed into an MLP, which lacks the inductive bias of order invariance of action set and cannot learn relations explicitly like graph networks.
- **Utility-Policy** (No input action set): By using action representations, this method can use the structure of action space for efficient training and also generalize to unseen actions. However, like *Mask-Output*, it does not utilize the available action set as part of the state and makes each action decision independent of other actions. Thus, it is expected to be suboptimal in all the tasks we consider.
- **Summary-LSTM**: This is an ablation where we do not utilize the per-action relational features computed by a GAT in AGILE. Instead, we use raw action features as input to the utility network. However, we utilize the available action set information by summarizing the set of input action representations into a vector. The summarization based on Bi-LSTM does not use the order invariance property of the action set, which makes it less efficient to learn than other summarizers. AGILE is expected to outperform all summary-only ablations in environments where several different action relations need to be modeled for computing different action utilities. This is true for complex environments like CREATE, where the agent needs to consider various tools and activators to make an optimal decision about its action choice. In such environments, learning a shared summary-vector for all action utility computations is inefficient and possibly prohibitive for learning. We note that the summary-ablations are sufficient in simple environments such as

Grid World navigation. The agent just needs to summarize which of the two dig-lava skills are available. Such a summary is enough to compute the utility of each of the 7 available actions.

- **Summary-Deep Set:** This ablation utilizes the order invariance of action sets using a deep-set. While it still suffers from the limitations of lack of per-action relational action features, it is a low-parameter network and thus easy to train.
- **Summary-GAT:** This is a variation of AGILE, where raw action features replace the relational action features. Using a GAT to extract the action set summary exploits order invariance. However, being a fully-connected graph network, it is can be slow to train while not offering much more than the deep-set summarizer. This can be useful when certain action relations are predefined using domain knowledge (Section B.1).
- **Agile-Only-Action:** This variation of AGILE does not utilize state input in the GAT. Thus, the summary vector and relational action features are computed independently of state-context. While this is a simpler architecture to train, it can be insufficient in environments where the action relations vary depending on the state.

C.2 Details on Listwise AGILE

AGILE is implemented based on the listwise RL architecture, CDQN (Chen et al., 2019a). As shown in Algorithm 5, CDQN builds a list-action incrementally by selecting N actions in sequence. For each list-index, a Q-network is used to select the best action. In addition to the usual state input, the partially built list-action is also an input to the Q-network. Concretely, the state and current list are encoded into latent embeddings for any list index. These are concatenated with each action representation in the remaining action set to build a list of nodes to be input into the action graph of AGILE. Treating it as a fully-connected graph, AGILE outputs a set of relational action features for all inputs actions. These are mean-pooled into a summary vector representing the entire action set. The representations of state, current-list, summary, and relational action features are used as input by a utility network (MLP) which outputs the Q-value of that particular action for the current

list-index. The action with the maximum Q-value is added to the list, and the algorithm moves to the next list-index.

For training AGILE-based CDQN, we maintain a target Q-network that is synchronized periodically with the main Q-network. Suppose a tuple of $(s, a_{\text{list}}, r, \tilde{s})$ is sampled from the replay buffer. For list index n between 1 and $N - 1$, we compute the target Q-value using the current state s with partial list-action $a_{1:n}$ as input. However, for the last list index $n = N$, we compute the target Q-value using the next state \tilde{s} for its first list index. A mean-squared error loss is used over all the list indices to train the main Q-network.

Algorithm 5 Cascaded DQN: Listwise Action RL

```

1: def listwise_action():
2:   Parameters: Q-network  $\phi$  - Encoders, GAT and Utility network
3:   Inputs: State  $s$ , actions  $\mathcal{A}$ , representations  $\mathcal{C} = \{c_{a_0}, \dots, c_{a_k}\}$ , list-action length  $N$ 
4:   Initialize: List Action  $a_{\text{list}} = []$ . Candidate Actions  $\mathcal{A}' = \mathcal{A}$ 
5:   for  $n = 1, \dots, N$  do:
6:     Encode State:  $e_s = \phi_s(s)$ 
7:     Encode Current List:  $e_{\text{list}} = \phi_{\text{list}}(a_{\text{list}})$ 
8:     Build Graph Nodes:  $\mathcal{V} = \{[e_s, e_{\text{list}}, c_a] : a \in \mathcal{A}'\}$ 
9:     Build fully-connected Adjacency Matrix of size  $|\mathcal{V}|$ :  $\mathcal{E}$ 
10:    Relational Action Features:  $e_{\text{relational}}^a = \phi_{\text{GAT}}(\mathcal{V}, \mathcal{E}) \forall a \in \mathcal{A}'$ 
11:    Action Set Summary:  $e_{\text{summary}} = \frac{1}{|\mathcal{V}|} \sum_a e_{\text{relational}}^a$ 
12:    Q-values:  $Q_n(s, a, \mathcal{A}') = \phi_{\text{utility}}(e_s, e_{\text{list}}, e_{\text{summary}}, e_{\text{relational}}^a)$  ▷ Ablations:  $e_{\text{relational}}^a = c_a$ 
13:    Select action:  $a_n = \arg \max_{a \in \mathcal{A}'} Q_n(s, a, \mathcal{A}')$ 
14:    Update:  $a_{\text{list}} = a_{\text{list}} \cup a_n$ .  $\mathcal{A}' = \mathcal{A}' \setminus a_n$ 
15: def listwise_update():
16:   Parameters: Q-network  $\phi$ , Target Q-network  $\phi_T$ , discount factor  $\gamma$ 
17:   Inputs: state  $s$ , action  $a_{\text{list}} = a_1, \dots, a_N$ , reward  $r$ , next state  $\tilde{s}$ 
18:   for  $n = 1, \dots, N$  do:
19:     Use listwise_action( $\phi$ ) to get Q-function:  $q_n = Q_n(s, a_n, \mathcal{A} \setminus a_{1:n-1})$ 
20:     Use listwise_action( $\phi_T$ ) to get target Q-value :

```

$$y_n = \begin{cases} r + \gamma \max_a Q_{n+1}(s, a, \mathcal{A} \setminus a_{1:n}) & n < N \\ r + \gamma \max_a Q_1(\tilde{s}, a, \mathcal{A}) & n = N \end{cases}$$

```

21:   Optimize Loss  $\mathcal{L} = \sum_{n=1}^N (q_n - y_n)^2$ 

```

C.3 Network Architectures

C.3.1 Action Graph

The action graph takes as input the action representations and the state-information (we also include the list-embedding for Listwise AGILE; See Sec. C.2). Given the concatenation of the input components above, an optional 2-layer MLP with ReLU, called *pre-summarizer-mlp*, transforms it into the node features for the action graph. Then, the resultant node features are passed on to two GAT layers followed by a residual connection. The same architecture of GAT is duplicated with different weights to provide separate pathways to compute the summary vector and relational action features. Therefore, we have two different sets of node features. The first set of node features is mean-pooled to produce the action-summary vector, which is put through a 2-layer MLP with ReLU to post-process before being passed on to the utility network. The other set of node features (i.e., relational action features) is directly fed into the utility network.

AGILE with GCN: When the GCN is used in the action graph, the same input as above goes into a linear layer to compress the size. Subsequently, the resultant node-features are used in GCN message-passing based on the normalized adjacency matrix, with all the diagonal elements made 0. Otherwise, the GCN reduces to learning the same edge coefficients for all the nodes. The rest of the following architecture is the same as GAT above.

Summary-GAT: In this ablation, instead of the relational action features being fed into the utility network, the raw action representations are used. Rest of the architecture remains the same.

C.3.2 Summarizers: Bi-LSTM and Deep Set

Bi-LSTM: The raw action representations of candidate actions are passed on to the 2-layer MLP followed by ReLU. Then, the output of the MLP is processed by a 2-layer bidirectional LSTM (Huang et al., 2015). Another 2-layer MLP follows this to create the action set summary to be used in the following utility network.

DeepSet: Similar to the Bi-LSTM variant of the summarizer, we employed the 2-layer MLP with ReLU followed by the mean pooling over all the candidate actions to compress the information. Finally, the 2-layer MLP with ReLU provides the resultant action summary with the following utility network described in the next subsection.

C.3.3 Utility Network

We implemented two types of utility networks to show the potential of AGILE architecture working with different kinds of RL algorithms and environments. Both the utility networks take as input the same components: the state-information, the relational or raw action representations, and the action summary of the action graph (See Sec C.3.1). The utility network is a 2-layer MLP applied parallelly on all these inputs corresponding to each action. It computes a scalar value for each action.

Policy Gradient(PPO): The action utility values are used as logits of a Categorical distribution, which is then used to train the AGILE architecture with policy gradient algorithm, PPO.

Value-based(CDQN): *List Encoder:* An intermediate list that contains the currently selected list items at each intra-list time-step is passed on to a single layer gated recurrent network (GRU (Cho et al., 2014)) followed by a 2-layer MLP to extract the compact representation of the intermediate list. See Algorithm 5 for details on this.

Q-network: As in Sec C.3.1, the input components for the utility network are the raw action representation, the state-information, the list-embedding, the node-features, and the action set summary from the action graph — $(s_t, e_{\text{list}}, e_{\text{node}}, e_{\text{summary}}, a_k)$ in Algorithm 5. These are concatenated into a single vector and passed on to a 2-layer MLP with ReLU to compute the Q-value of an item.

C.3.4 Action Representation Network

Hierarchical VAE (CREATE): The CREATE action representations are borrowed from Jain et al. (2020) directly. The Hierarchical VAE network takes as input a list of behavioral trajectories

representing each action (tool) and reconstructs it with a two-layer hierarchy of VAEs. We refer the reader to Appendix D.3.1 of [Jain et al. \(2020\)](#) for complete details of the Hierarchical VAE network.

VAE in Deep and Wide architecture (Real World recommender system): In a VAE, the encoder is implemented with a 5-layer MLP with a Batch Normalization layer ([Maas et al., 2013](#)) and LeakyReLU ([Maas et al., 2013](#)). On the other hand, the decoder is implemented with a single MLP to process the input, followed by the same architecture of MLPs used in the encoder. Finally, a 2-layer MLP with Batch Norm and LeakyReLU and the hyperbolic tangent activation function is used to reconstruct the input instance.

C.3.5 Reward Inference Network (User Model in Real World RecSys)

The user model takes as input the user information(i.e., a concatenation of user attributes and a sequence of the user interactions) and a set of item embeddings in the list. The user information is passed on to a single layer gated recurrent network(GRU ([Cho et al., 2014](#))) followed by a 2-layer MLP to extract the compact representation of the state. The same GRU network architecture (with different weights) processes the set of item embeddings into a list-embedding. Finally, a 2-layer MLP takes as input the concatenation of those two embeddings(i.e., state-embedding and list-embedding) and provides the scores of items in the list followed by the sigmoid function to transform to the individual click likelihood.

D Experiment Details

D.1 Implementation Details

We used PyTorch ([Paszke et al., 2019](#)) for our implementation, and the experiments were primarily conducted on workstations with either NVIDIA GeForce RTX 2080 Ti, P40, or V100 GPUs on Naver Smart Machine Learning platform (NSML) ([Kim et al., 2018](#)). Each experiment seed takes about 4 hours for Grid Navigation, 60 hours for CREATE, 8 hours for RecSim, and 15 hours for Real-Data Recommender Systems, to converge. We use the Weights & Biases tool ([Biewald, 2020a](#))

for logging and tracking experiments. All the environments were developed using the OpenAI Gym interface (Brockman et al., 2016). For training Grid Navigation and CREATE environments, we use the PPO (Schulman et al., 2017b) implementation based on Kostrikov (2018). For the recommender system environments, we use DQN (Mnih et al., 2015). We use the Adam optimizer (Kingma and Ba, 2014) throughout. We attach the code with details to reproduce all the experiments, except the real-data recommender system.

D.2 Hyperparameters

We build on hyperparameters used in prior work on CDQN (Chen et al., 2019a) and utility policy (Jain et al., 2020). The hyperparameters for the additional components introduced in AGILE, baselines, and ablations are shown in Table B.3. The environment-specific and RL algorithm hyperparameters are described in Table B.4.

Hyperparameter	Value
AGILE	
number of GAT layers	2
number of attention heads	1
number of message passing steps	1
leakyReLU alpha	0.2
graph hidden dimension	64
Residual Connection	True
Mask-Input-Output	
Availability mask MLP hidden size	64
Ablations	
Summary-LSTM hidden size	64
Summary-LSTM directions	2
Summary-LSTM number of layers	2
Summary-Deep-Set hidden size	64
Summary-GAT hidden size	64
Summarizer pooling operation	mean

Table B.3: AGILE, baseline and ablations: Hyperparameters for additional components

D.3 Hyperparameter Tuning

Initial hyperparameters are inherited from the prior works (PPO: [Jain et al. \(2020\)](#); CDQN: [Chen et al. \(2019a\)](#)). To ensure fairness across all baselines and our methods, we use the original hyperparameters for the RL algorithms, such as learning rate, entropy coefficient, etc. (Table B.4). We choose a sufficiently large number for total epochs (DQN) and total environment steps (PPO) to ensure all the methods converge.

Our main contribution is the AGILE architecture. Thus, to correctly validate against baseline and ablation architectures, we search over shared parameters together and generally observe the same trend across all methods. This was expected because of the shared underlying implementation of the RL algorithm. Specifically, we searched over the following shared parameters, usually over 5 seeds and sometimes 3 seeds:

- **Hidden dimension in DQN networks:** We searched over $\{32, 64, 128\}$ and found that $64 \approx 128 > 32$. We choose 64 as the base hidden dimension across all encoders, summarizers, and GATs.
- **Target network sync frequency:** This was a sensitive parameter for DQN training. We searched over $\{10, 20, 100, 500, 1000\}$ and found 500 to be the best performing across all methods.
- **Weight sharing in CDQN:** In [Chen et al. \(2019a\)](#), the authors did not share the weights of Q-nets in their Cascading architecture, but we empirically observed that sharing the weights of them improved the performance. This is likely due to the nature of RecSim tasks requiring similar items to be recommended across the slate to maximize CPR.
- **Training batch size (DQN):** We searched over $\{32, 64, 128, 256\}$ on RecSim and chose 128 as, beyond that, we observed diminishing returns. For Real RecSys, we used the same batch size.
- **Training batch size (PPO):** We searched over $\{4096, 8192\}$ for Grid World and $\{3072, 4608\}$ for CREATE and observed that larger batch sizes are better. We did not increase the batch size further due to limits on GPU memory size.

Below, we detail the hyperparameters and network variations searched specifically for the graph attention network (GAT) used in AGILE and its variants.

- **Skip Connection:** Adding skip connection was crucial to making AGILE learn well.
- **Number of graph attention layers:** We searched over $\{1, 2, 3\}$ layers and found 2 layers to be the best performing for both PPO and CDQN based policies.
- **Number of attention heads:** We compared having $\{1, 2\}$ attention heads and found no meaningful learning for 2 attention heads.
- **Number of message passing steps:** Adding more message-passing steps than 1 did not improve AGILE performance.

Other relevant design choices are described and validated in Section B.2.

Hyperparameter	Grid world	CREATE	RecSim	Real RecSys
Environment				
action representation size	11	134	20	32
observation space	162	$84 \times 84 \times 3$	10	507
candidate actions per episode	7	25	20	20
max. episode length	50	30	15	10
RL Training				
training batch size	8192	4608	128	128
parallel processes	64	48	16	16
discount factor	0.99	0.99	0.99	0.99
learning rate	1e-3	1e-3	1e-4	1e-4
hidden layer size	64	64	64	64
PPO				
entropy coefficient	0.05	0.005		
continuous action entropy coefficient	-	0.0005		
total environment steps	8×10^6	10^8		
value loss coefficient	0.5	0.5		
PPO epochs	4	4		
PPO clip parameter	0.1	0.1		
DQN				
epochs			10000	5000
target network sync frequency			500	500
Q-network hidden dimension			256×64	256×64
list encoder dimension			64	64
action encoder dimension			32	32
list size			6	6
replay buffer size			10^6	10^6
initialize replay buffer size			5000	100
epsilon decay			$1 \rightarrow 0.1$	$1 \rightarrow 0.1$
epsilon decay last epoch			500	250

Table B.4: Environment/Policy-specific Hyperparameters for RL training in AGILE

Appendix C

Optimizing Action in Non-Convex Q-functions via Successive Actors

A Proof of Convergence of Maximizer Actor in Tabular Settings

Theorem A.1 (Convergence of Policy Iteration with Maximizer Actor). *In a finite Markov Decision Process (MDP) with finite state space \mathcal{S} , consider a modified policy iteration algorithm where, at each iteration n , we have a set of $k + 1$ policies $\{\nu_0, \nu_1, \dots, \nu_k\}$, with $\nu_0 = \mu_n$ being the policy at the current iteration learned with DPG. We define the maximizer actor μ_M as:*

$$\mu_M(s) = \arg \max_{a \in \{\nu_0(s), \nu_1(s), \dots, \nu_k(s)\}} Q^{\mu_n}(s, a), \quad (\text{A.1})$$

where $Q^{\mu_n}(s, a)$ is the action-value function for policy μ_n . Then, the modified policy iteration algorithm using the maximizer actor is guaranteed to converge to a final policy μ_N .

Proof. To prove convergence, we will show that the sequence of policies μ_n yields monotonically non-decreasing value functions that converge to a stable value function V^N .

Policy Evaluation Converges

Thus, iteratively applying \mathcal{T}^π starting from any initial Q_0 converges to the unique fixed point Q^π .

Given the current policy μ_n , the policy evaluation computes the action-value function Q^{μ_n} , satisfying:

$$Q^{\mu_n}(s, a) = R(s, a) + \gamma \sum_{s'} P(s, a, s') V^{\mu_n}(s'),$$

where $V^{\mu_n}(s') = Q^{\mu_n}(s', \mu_n(s'))$.

In the tabular setting, the Bellman operator \mathcal{T}^{μ_n} defined by

$$[\mathcal{T}^{\mu_n} Q](s, a) = R(s, a) + \gamma \sum_{s'} P(s, a, s') Q(s', \mu_n(s'))$$

is a contraction mapping with respect to the max norm $\|\cdot\|_\infty$ with contraction factor γ ,

$$\|\mathcal{T}^{\mu_n} Q - \mathcal{T}^{\mu_n} Q'\|_\infty \leq \gamma \|Q - Q'\|_\infty.$$

Therefore, iteratively applying \mathcal{T}^{μ_n} converges to the unique fixed point Q^{μ_n} .

Policy Improvement with DPG and Maximizer Actor

Step 1: DPG Update

We define $\tilde{\mu}_n$ as the DPG policy that locally updates μ_n towards maximizing the expected return based on Q^{μ_n} . For each state s , we perform a gradient ascent step using the Deep Policy Gradient (DPG) method to obtain an improved policy $\tilde{\mu}_n$:

$$\tilde{\mu}_n(s) \leftarrow \mu_n(s) + \alpha \nabla_a Q^{\mu_n}(s, a) \Big|_{a=\mu_n(s)},$$

where $\alpha > 0$ is a suitable step size.

This DPG gradient step leads to local policy improvement following over μ_n (Silver et al., 2014):

$$V^{\tilde{\mu}_n}(s) \geq V^{\mu_n}(s), \quad \forall s \in \mathcal{S}.$$

(b) *Maximizer Actor*

Given additional policies ν_1, \dots, ν_k , define the maximizer actor μ_{n+1} as:

$$\mu_{n+1}(s) = \arg \max_{a \in \{\tilde{\mu}_n(s), \nu_1(s), \dots, \nu_k(s)\}} Q^{\mu_n}(s, a).$$

Since $\mu_{n+1}(s)$ selects the action maximizing $Q^{\mu_n}(s, a)$ among candidates, we have:

$$Q^{\mu_n}(s, \mu_{n+1}(s)) = \max_{a \in \{\tilde{\mu}_n(s), \nu_1(s), \dots, \nu_k(s)\}} Q^{\mu_n}(s, a) \geq Q^{\mu_n}(s, \tilde{\mu}_n(s)) \geq V^{\mu_n}(s).$$

By the Policy Improvement Theorem, since $Q^{\mu_n}(s, \mu_{n+1}(s)) \geq V^{\mu_n}(s)$ for all s , it follows that:

$$V^{\mu_{n+1}}(s) \geq V^{\mu_n}(s), \quad \forall s \in \mathcal{S}.$$

Thus, the sequence $\{V^{\mu_n}\}$ is monotonically non-decreasing.

Convergence of Policy Iteration

Since the sequence $\{V^{\mu_n}\}$ is monotonically non-decreasing and bounded above by V^* , it converges to some $V^\infty \leq V^*$. Given the finite number of possible policies, the sequence $\{\mu_n\}$ must eventually repeat a policy. Suppose that at some iteration N , the policy repeats, i.e., $\mu_{N+1} = \mu_N$.

At this point, since the policy hasn't changed, we have:

$$\mu_N(s) = \arg \max_{a \in \{\tilde{\mu}_N(s), \nu_1(s), \dots, \nu_k(s)\}} Q^{\mu_N}(s, a), \quad \forall s \in \mathcal{S}.$$

Since $\tilde{\mu}_N(s)$ is obtained by performing a DPG update on $\mu_N(s)$, and we have that $\mu_N(s)$ maximizes $Q^{\mu_N}(s, a)$ among $\{\tilde{\mu}_N(s), \nu_1(s), \dots, \nu_k(s)\}$, it must be that:

$$Q^{\mu_N}(s, \mu_N(s)) \geq Q^{\mu_N}(s, a), \quad \forall a \in \{\tilde{\mu}_N(s), \nu_1(s), \dots, \nu_k(s)\}.$$

Moreover, since $\tilde{\mu}_N(s)$ is obtained via gradient ascent from $\mu_N(s)$, and yet does not yield a higher Q -value, it implies that:

$$\nabla_a Q^{\mu_N}(s, a)|_{a=\mu_N(s)} = 0.$$

This suggests that $\mu_N(s)$ is a local maximum of $Q^{\mu_N}(s, a)$. This shows that this modification to the policy iteration algorithm of DPG is guaranteed to converge.

Since the set $\{\tilde{\mu}_N(s), \nu_1(s), \dots, \nu_k(s)\}$ includes more actions from \mathcal{A} , $\mu_N(s)$ is the action that better maximizes $Q^{\mu_N}(s, a)$ than $\tilde{\mu}_N$. Therefore, μ_N is a greedier policy with respect to Q^{μ_N} than $\tilde{\mu}_N$.

□

B Proof of Reducing Number of Local Optima in Successive Surrogates

Theorem B.1. *Consider a state $s \in \mathcal{S}$, the function Q as defined in Eq. 4.1, and the surrogate functions Ψ_i as defined in Eq. 4.7. Let $N_{opt}(f)$ denote the number of local optima (assumed countable) of a function $f : \mathcal{A} \rightarrow \mathbb{R}$, where \mathcal{A} is the action space. Then,*

$$N_{opt}(Q(s, a)) \geq N_{opt}(\Psi_0(s, a; \{a_0\})) \geq N_{opt}(\Psi_1(s, a; \{a_0, a_1\})) \geq \dots \geq N_{opt}(\Psi_k(s, a; \{a_0, \dots, a_k\})).$$

Proof. For each $i \geq 0$, define the surrogate function Ψ_i recursively:

$$\Psi_i(s, a; \{a_0, \dots, a_i\}) = \max \{Q(s, a), \tau_i\}, \tag{B.1}$$

where

$$\tau_i = \max_{0 \leq j \leq i} Q(s, a_j).$$

Note that τ_i is non-decreasing with respect to i , i.e., $\tau_{i+1} \geq \tau_i$.

Our goal is to show that for each $i \geq 0$,

$$N_{\text{opt}}(\Psi_i(s, a; \{a_0, \dots, a_i\})) \geq N_{\text{opt}}(\Psi_{i+1}(s, a; \{a_0, \dots, a_{i+1}\})).$$

We proceed by considering how the set of local optima changes from Ψ_i to Ψ_{i+1} .

Consider any local optimum a' of Ψ_i . There are two cases:

Case 1: $Q(s, a') > \tau_i$

In this case, $\Psi_i(s, a') = Q(s, a')$ and Ψ_i coincides with Q in a neighborhood of a' . Since a' is a local optimum of Ψ_i , it is also a local optimum of Q . Because $\tau_{i+1} \geq \tau_i$, there are two subcases:

Subcase 1a: $Q(s, a') > \tau_{i+1}$

Here, $\Psi_{i+1}(s, a') = Q(s, a')$ and, in a neighborhood of a' , Ψ_{i+1} coincides with Q . Thus, a' remains a local optimum of Ψ_{i+1} .

Subcase 1b: $Q(s, a') \leq \tau_{i+1}$

Since $Q(s, a') > \tau_i$ and $\tau_{i+1} \geq \tau_i$, this implies $\tau_{i+1} > \tau_i$ and $Q(s, a') = \tau_{i+1}$. Then,

$$\Psi_{i+1}(s, a') = \tau_{i+1},$$

and in a neighborhood around a' , $\Psi_{i+1}(s, a) \geq \tau_{i+1}$. Thus, a' is not a local optimum of Ψ_{i+1} because there is no neighborhood where $\Psi_{i+1}(s, a) < \Psi_{i+1}(s, a')$.

Case 2: $Q(s, a') \leq \tau_i$

In this case, $\Psi_i(s, a') = \tau_i$, and Ψ_i is constant at τ_i in a neighborhood of a' . Thus, a' may be considered a local optimum in Ψ_i if the function does not exceed τ_i nearby. When moving to Ψ_{i+1} , since $\tau_{i+1} \geq \tau_i$, we have:

$$\Psi_{i+1}(s, a') = \tau_{i+1} \geq \tau_i.$$

In the neighborhood of a' , Ψ_{i+1} remains at least τ_{i+1} , so a' is not a local optimum in Ψ_{i+1} unless $Q(s, a) < \tau_{i+1}$ in a neighborhood around a' .

However, since $\Psi_{i+1}(s, a) \geq \tau_{i+1}$ for all a , the function does not decrease below $\Psi_{i+1}(s, a')$ in any neighborhood of a' . Therefore, a' is not a local optimum of Ψ_{i+1} .

Conclusion: From the above cases, we observe that:

- Any local optimum a' of Ψ_i where $Q(s, a') > \tau_{i+1}$ remains a local optimum in Ψ_{i+1} .
- Any local optimum a' of Ψ_i where $Q(s, a') \leq \tau_{i+1}$ does not remain a local optimum in Ψ_{i+1} .

Since Ψ_{i+1} does not introduce new local optima (because $\Psi_{i+1}(s, a) \geq \Psi_i(s, a)$ for all a and coincides with Q only where $Q(s, a) > \tau_{i+1}$), the number of local optima does not increase from Ψ_i to Ψ_{i+1} .

Base Case: For $i = 0$, we have:

$$\Psi_0(s, a; \{a_0\}) = \max \{Q(s, a), Q(s, a_0)\}.$$

If we consider $Q(s, a_0)$ to be less than the minimum value of $Q(s, a)$ (which can be arranged by choosing a_0 appropriately or by defining τ_0 to be less than $\inf_a Q(s, a)$), then $\Psi_0(s, a) = Q(s, a)$, and the base case holds trivially.

Inductive Step: Assuming that

$$N_{\text{opt}}(\Psi_i(s, a; \{a_0, \dots, a_i\})) \leq N_{\text{opt}}(\Psi_{i+1}(s, a; \{a_0, \dots, a_{i+1}\})),$$

we have shown that

$$N_{\text{opt}}(\Psi_{i+1}(s, a; \{a_0, \dots, a_{i+1}\})) \leq N_{\text{opt}}(\Psi_i(s, a; \{a_0, \dots, a_i\})).$$

By induction, it follows that:

$$N_{\text{opt}}(Q(s, a)) \geq N_{\text{opt}}(\Psi_0(s, a; \{a_0\})) \geq N_{\text{opt}}(\Psi_1(s, a; \{a_0, a_1\})) \geq \dots \geq N_{\text{opt}}(\Psi_k(s, a; \{a_0, \dots, a_k\})).$$

□

C Environment Details

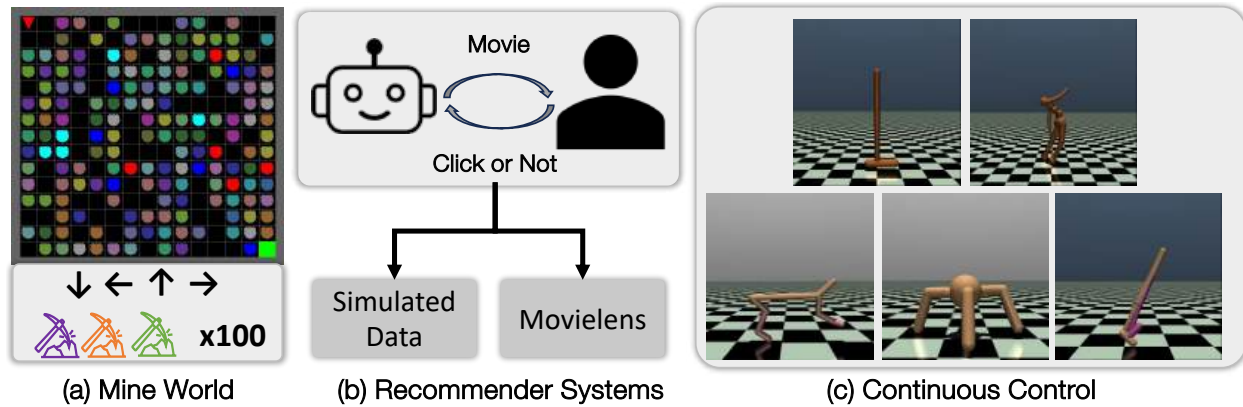


Figure C.1: **Benchmark Environments** involve discrete action space tasks like Mine World and recommender systems (simulated and MovieLens-Data) and restricted locomotion tasks.

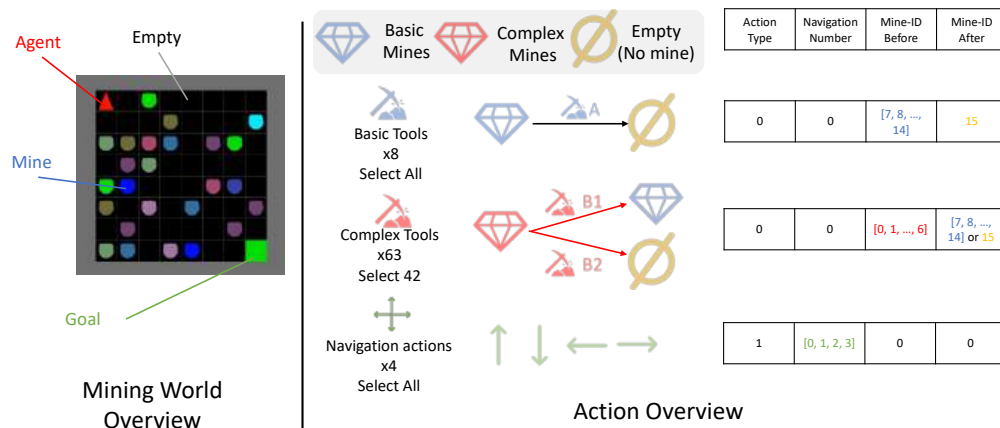


Figure C.2: **Mining Expedition**. The red agent must reach the green goal by navigating the grid and using one or more pick-axes to clear each mine blocking the path.

C.1 MiningEnv

The grid world environment, introduced in Section 4.5, requires an agent to reach a goal by navigating a 2D maze as soon as possible while breaking the mines blocking the way.

State: The state space is an $8+K$ dimensional vector, where K equals to *mine-category-size*. This vector consists of 4 independent pieces of information: Agent Position, Agent Direction, Surrounding Path, and Front Cell Type.

1. Agent Position: A 2D vector representing the agent’s x and y coordinates.
2. Agent Direction: One dimension representing directions (0: right, 1: down, 2: left, 3: up).
3. Surrounding Path: A 4-dimensional vector indicating if the adjacent cells are empty or a goal (1: empty/goal, 0: otherwise).
4. Front Cell Type: A $(K + 1)$ -dimensional one-hot vector with first K dimensions representing the type of mine and the last dimension representing if the cell is empty (zero) or goal (one).

Finally, we will normalize each dimension to $[0, 1]$ with each dimension’s minimum/maximum value.

Termination: An episode terminates when the agent reaches the goal or after 100 timesteps. Upon reset, the grid layout changes while keeping the agent’s start and goal positions fixed.

Actions: Actions include navigation (up, down, left, right) and pick-axe categories. Navigation changes the agent’s direction and attempts to move forward. The agent cannot step into a mine but will change direction when trying to step onto a mine or the border of the grid. The pick-axe tool actions (50 types) have a predefined one-to-one mapping of how they interact with the mines, which means they can be successfully applied to only one kind of mine, and either transform that kind of mine into another type of mine or directly break it.

Reward: The agent’s reward comprises a goal-reaching reward, a distance-based step reward, and rewards for successful tool use or mine-breaking. The goal reward is discounted by steps taken over the episode, encouraging shorter paths to reach the goal.

$$\begin{aligned}
R(s, a) = & \mathbb{1}_{Goal} \cdot R_{Goal} \left(1 - \lambda_{Goal} \frac{N_{\text{current steps}}}{N_{\text{max steps}}} \right) + \\
& R_{Step} (D_{\text{distance before}} - D_{\text{distance after}}) + \\
& \mathbb{1}_{\text{correct tool applied}} \cdot R_{Tool} + \\
& \mathbb{1}_{\text{successfully break mine}} \cdot R_{Bonus}
\end{aligned} \tag{C.1}$$

where $R_{Goal} = 10$, $R_{Step} = 0.1$, $R_{Tool} = 0.1$, $R_{Bonus} = 0.1$, $\lambda_{Goal} = 0.9$, $N_{\text{max steps}} = 100$

Action Representations Actions are represented as a 4D vector with normalized values $[0, 1]$ as described in Figure C.2. Dimensions represent skill category (navigation or pick-axe), movement direction (right, down, left, up), mine type where this action can be applied, and the outcome of applying the tool to the mine, respectively.

C.2 RecSim

In the simulated recommendation system (RecSys) environment, the agent selects an item from a large set that aligns with the user’s interests. Users are modeled with dynamically changing preferences that evolve based on their interactions (clicks). The agent’s objective is to infer these evolving preferences from user clicks and recommend the most relevant items to maximize the total number of clicks.

State: The user’s interest is represented by an embedding vector $e_u \in \mathbb{R}^n$, where n is the number of item categories. This embedding evolves over time as the user interacts with different items. When a user clicks on an item with embedding $e_i \in \mathbb{R}^n$, the user interest embedding e_u is updated as follows:

$$\begin{aligned} e_u &\leftarrow e_u + \Delta e_u, & \text{with probability } \frac{e_u^\top e_i + 1}{2} \\ e_u &\leftarrow e_u - \Delta e_u, & \text{with probability } \frac{1 - e_u^\top e_i}{2}, \end{aligned}$$

where Δe_u represents an adjustment that depends on the alignment between e_u and e_i . This update mechanism adjusts the user’s preference towards the clicked item, reinforcing the connection between the current action on future recommendations.

Action: The action set consists of all items that can be recommended, and the agent must select the item most relevant to the user’s long-term preferences over the episode.

Reward: The reward is based on user feedback: either a click (reward = 1) or skip (reward = 0). The user model computes a score for each item using the dot product of the user and item embeddings:

$$\text{score}_{item} = \langle e_u, e_i \rangle$$

The click probability is computed with a softmax over the item score and a predefined skip score:

$$p_{item} = \frac{e^{\text{score}_{item}}}{e^{\text{score}_{item}} + e^{\text{score}_{skip}}}, \quad p_{skip} = 1 - p_{item}$$

The user then stochastically chooses to click or skip based on this distribution.

Action Representations: Following [Jain et al. \(2021\)](#), items are represented as continuous vectors sampled from a Gaussian Mixture Model (GMM), with centers representing item categories.

C.3 Continuous Control

MuJoCo ([Todorov et al., 2012](#)) is a physics engine that provides a suite of standard reinforcement learning tasks with continuous action spaces, commonly used for benchmarking continuous control algorithms. We briefly describe some of these tasks below:

Hopper: The agent controls a one-legged robot that must learn to hop forward while maintaining balance. The objective is to maximize forward velocity without falling.

Walker2d: The agent controls a two-legged bipedal robot that must learn to walk forward efficiently while maintaining balance. The goal is to achieve stable locomotion at high speeds.

HalfCheetah: The agent controls a planar, cheetah-like robot with multiple joints in a 2D environment. The task requires learning a coordinated gait to propel the robot forward as quickly as possible.

Ant: The agent controls a four-legged, ant-like robot with multiple degrees of freedom. The challenge is to learn to walk and navigate efficiently while maximizing forward progress.

C.3.1 Restricted Locomotion in Mujoco

The restricted locomotion Mujoco tasks are introduced to demonstrate how common DPG-based approaches get stuck in local optima when the Q-landscape is complex and non-convex. This setting limits the range of actions the agent can perform in each dimension, simulating realistic scenarios such as wear and tear of hardware. For example, action space may be affected as visualized in Figure 4.6. A mixture-of-hypersphere action space is used to simulate such asymmetric restrictions, which affect the range of torques for the Hopper and Walker joints, as well as the forces applied to the inverted pendulum and double pendulum. The hyperspheres are sampled randomly, and their size and radius are carefully tuned to ensure that the action space has enough valid actions to solve the task.

Definition of restriction.

- **Restricted Hopper & Walker**

Invalid action vectors are replaced with 0 by changing the environment's step function code:

```
1 def step(action):
2     ...
3     if check_valid(action):
4         self.do_simulation(action)
5     else:
6         self.do_simulation(np.zeros_like(action))
7     ...
```

The Hopper action space is 3-dimensional, with torque applied to [thigh, leg, foot], while the Walker action space is 6-dimensional, with torque applied to [right thigh, right leg, right foot, left thigh, left leg, left foot]. The physical implication of restricted locomotion is that zero torques are exerted for the Δt duration between two actions, i.e., no torques are applied for 0.008 seconds. This effectively slows down the agent's current velocities and angular velocities due to friction whenever the agent selects an invalid action.

- **Inverted Pendulum & Inverted Double Pendulum**

Invalid action vectors are replaced with -1 by changing the environment’s step function code:

```
1 def step(action):
2     ...
3     if not check_valid(action):
4         action[:] = -1.
5     self.do_simulation(action)
6     ...
```

The action space is 1-dimensional, with force applied on the cart. The implication is that the cart is pushed in the left direction for 0.02 (default) seconds. Note that the action vectors are not zeroed because a 0-action is often the optimal action, particularly when the agent starts upright. This would make the optimal policy trivially be *learning to select invalid actions*.

D Additional Results

D.1 Experiments on Continuous Control (Unrestricted Mujoco)

In standard MuJoCo tasks, the Q-landscape is likely easier to optimize compared to MuJoCo-Restricted tasks. In Figure C.3, we find that baseline models consistently perform well in all standard tasks, unlike in MuJoCo-Restricted tasks. Thus, we can infer the following:

1. Baselines have sufficient capacity, are well-tuned, and can navigate simple Q-landscapes optimally.
2. SAVO performs on par with other methods in MuJoCo tasks where the Q-landscape is easier to optimize, showing that SAVO is a robust, widely applicable actor architecture.
3. Baselines performing well in unrestricted locomotion but suboptimally in restricted locomotion delineates the cause of suboptimality to be the complexity of the underlying Q-landscapes, such as those shown in Figure 4.2. SAVO is closer to optimal in both settings because it can navigate both simple and complex Q-functions better than alternate actor architectures.

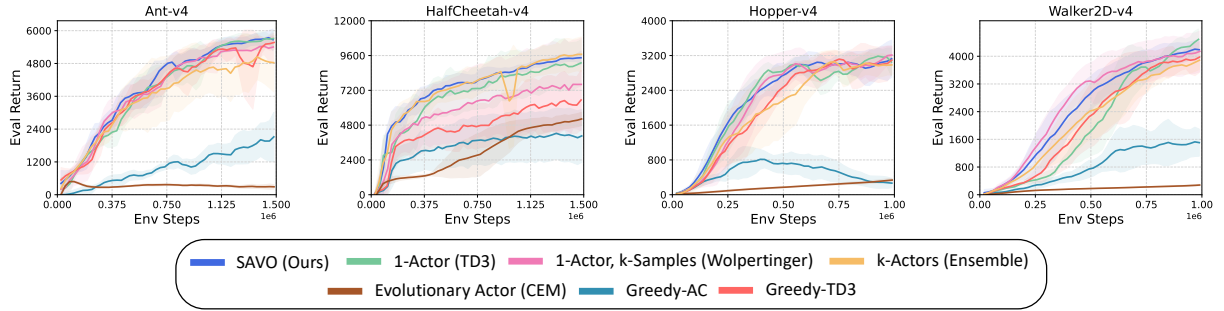


Figure C.3: **Unrestricted Locomotion** (Section D.1). SAVO and most baselines perform optimally in standard MuJoCo continuous control tasks, where the Q-landscape is easy to navigate.

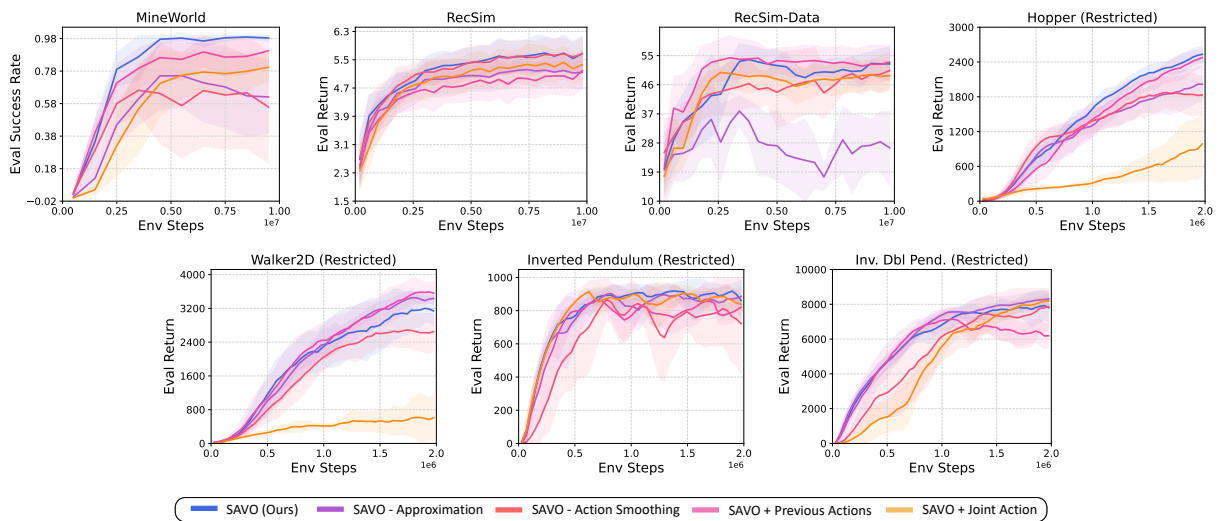


Figure C.4: **Ablations of SAVO variations** (Section D.2) shows the importance of (i) the approximation of surrogates, (ii) removing TD3’s action smoothing, (iii) conditioning on preceding actions in the successive actor and surrogate networks, and (iv) individual actors that separate the action candidate prediction instead of a joint high-dimensional learning task.

D.2 Per-Environment Ablation Results

Figure C.4 shows the per-environment performance of SAVO ablations, compiled into aggregate performance profiles in Figure 4.7b. The **SAVO - Approximation** variant underperforms significantly in discrete action space tasks, where traversing between local optima is complex due to nearby actions having diverse Q-values (see the right panel of Figure 4.2). Similarly, adding TD3’s target action smoothing to SAVO results in inaccurate learned Q-values when several differently valued actions exist near the target action, as in the complex landscapes of all tasks considered.

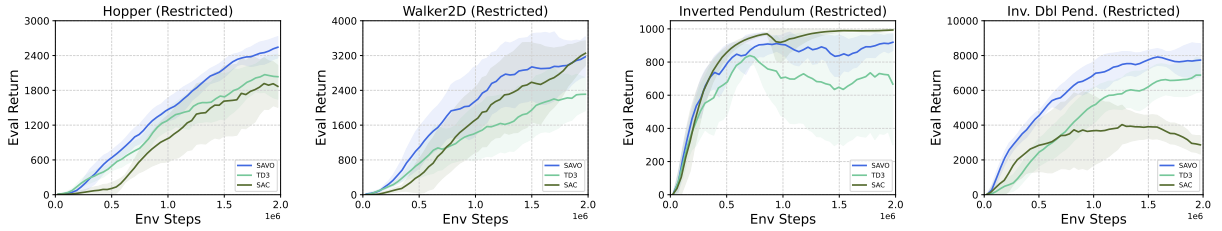


Figure C.5: **SAC is orthogonal to the effect of SAVO** (Section D.3). SAC is a different algorithm than TD3, whereas SAVO is a plug-in actor architecture for TD3. Thus, tasks where SAC outperforms TD3 differ from tasks where SAVO outperforms TD3. Also, TD3 outperforms SAC in Restricted Hopper and Inverted-Double-Pendulum. However, SAVO+TD3 guarantees improvement over TD3.

Removing information about preceding actions does not significantly degrade SAVO’s performance since preceding actions’ Q-values are indirectly incorporated into the surrogates’ training objective (see Eq. (4.9)), except for MineWorld where this information helps improve efficiency.

The **SAVO + Joint** ablation learns a single actor that outputs a joint action composed of k constituents, aiming to cover the action space so that multiple coordinated actions can better maximize the Q-function compared to a single action. However, this increases the complexity of the architecture and only works in low-dimensional tasks like Inverted-Pendulum and Inverted-Double-Pendulum. SAVO simplifies action candidate generation by using several successive actors with specialized objectives, enabling easier training without exploding the action space.

D.3 SAC is Orthogonal to SAVO

We compare Soft Actor-Critic (SAC), TD3, and TD3 + SAVO across various Mujoco-Restricted tasks. Figure C.5 shows that SAC sometimes outperforms and sometimes underperforms TD3. Therefore, SAC’s stochastic policy does not address the challenge of non-convexity in the Q-function. In contrast, SAVO+TD3 consistently outperforms TD3 and SAC, demonstrating the effectiveness of SAVO in complex Q-landscapes. While SAC can be better than TD3 in certain environments, its algorithmic modifications are orthogonal to the architectural improvements due to the SAVO actor.

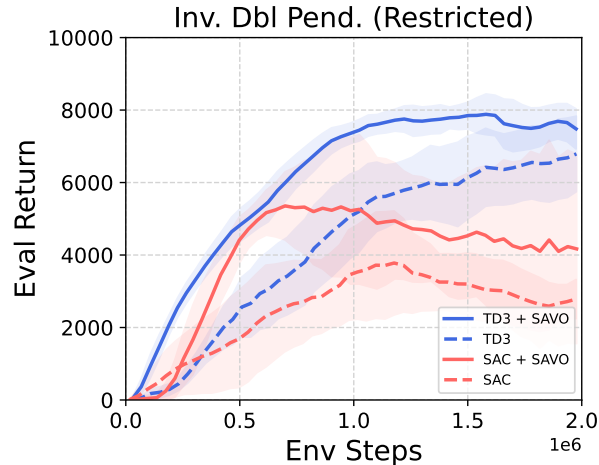


Figure C.6: **SAC is suboptimal in complex Q-landscape** (Section D.3) of Restricted Inverted Double Pendulum, but SAVO helps.

In the Restricted Inverted Double Pendulum task (Figure C.6), SAC underperforms even TD3. Analogous to TD3, the suboptimality is due to the non-convexity in the soft Q-function landscape, where small changes in nearby actions can lead to significantly different environment returns. We combine SAC with SAVO’s successive actors and surrogates to better maximize the soft Q-function, naively considering action candidates for μ_M as the mean action of the stochastic actors. We observe that this preliminary version of SAC + SAVO shows significant improvements over SAC in complex Q-landscapes. In future work, we aim to formalize a SAVO-like objective that effectively enables SAC’s stochastic actors to navigate the non-convexity of its soft Q-function.

D.4 Increasing Size of Discrete Action Space in RecSim

We test the robustness of our method to more challenging Q-value landscapes in Figure C.7, especially in discrete action space tasks with massive action spaces. In RecSim, we increase the number of actual discrete actions from 10,000 to 100,000 and 500,000. The experiments show that SAVO outperforms the best-performing baseline of TD3 + Sampling (Wolpertinger) and the best-performing ablation of SAVO + Joint Action. This shows that SAVO maintains robust performance even as the action space size increases and the Q-function landscape becomes more intricate. In contrast, the baselines experienced performance deterioration as action sizes grew larger.

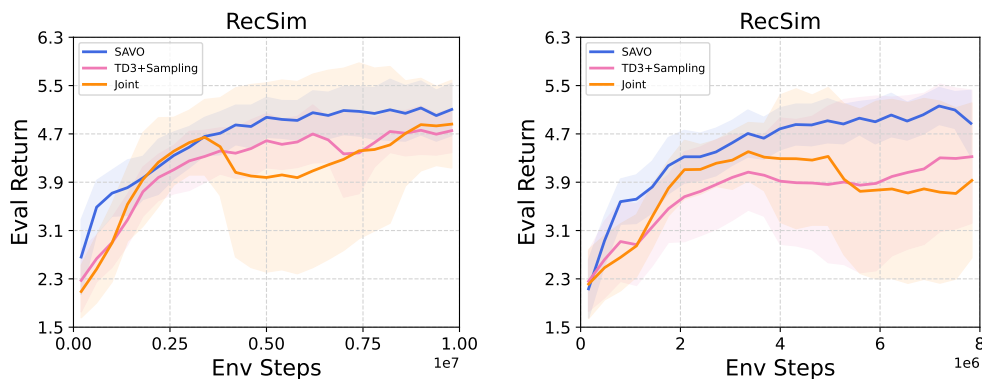


Figure C.7: **Increasing RecSim action set size** (Section D.4). (Left) 100,000 items, (Right) 500,000 items (6 seeds) maintains the performance trends of SAVO and the best-performing baseline (TD3 + Sampling) and the best-performing ablation (SAVO with Joint-Action).

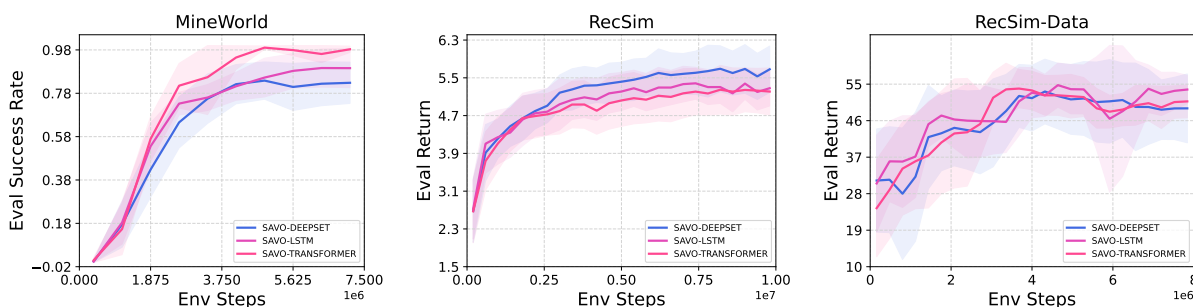


Figure C.8: **Action summarizer comparison** (Section E.1). The effect is not significant. The results are averaged over 5 random seeds, and the seed variance is shown with shading.

E Validating SAVO Design Choices

E.1 Design Choices: Action summarizers

Three key architectures were considered for the design of the action summarizer: DeepSets, LSTM, and Transformer models, represented by SAVO, SAVO-LSTM, and SAVO-Transformer in Figure C.8, respectively. In general, the effect of the action summarizer is not significant, and we choose DeepSet for its simplicity for most experiments.

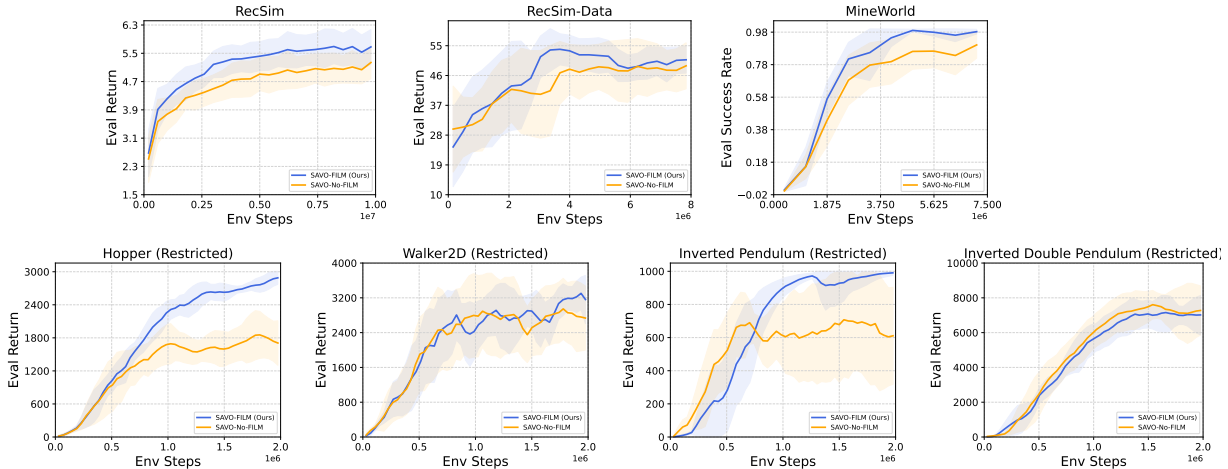


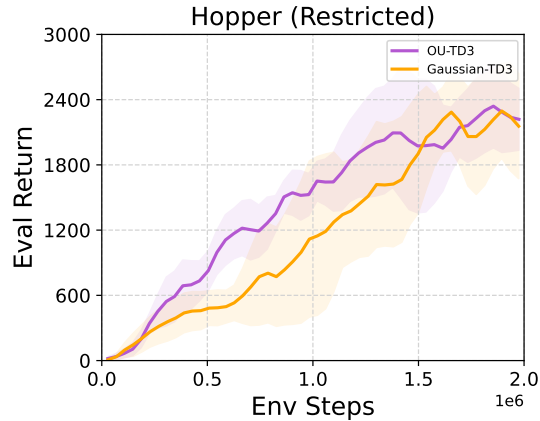
Figure C.9: **FiLM to condition on preceding actions** (Section E.2). FiLM ensures layerwise dependence on the preceding actions for acting in actors ν_i and for predicting value in surrogates $\hat{\Psi}_i$, which generally results in better performance across tasks.

E.2 Conditioning on Previous Actions: FiLM vs. MLP

We examined two approaches for conditioning on the previous action list summary: Feature-wise Linear Modulation (FiLM) and concatenation with input, represented by the FiLM and non-film variants in Figure C.9. Across tasks, FiLM outperformed the non-FiLM version, showing the effectiveness of layerwise conditioning in leveraging prior action information for action selection and surrogate value prediction. This shows that the successive actors are appropriately utilizing the actions from the preceding actors to tailor their search for optimal actions, and the successive surrogates can better evaluate Q-values, knowing where they can be thresholded by the loss function.

E.3 Exploration Noise comparison: OUNoise vs Gaussian

We compare Ornstein-Uhlenbeck (OU) noise with Gaussian noise across our environments and find that OU noise was generally better, with the difference being minimal. We chose to use OU for our experiments and a comparison on Hopper-Restricted is shown in Figure C.10a. We note that TD3 (Fujimoto et al., 2018) also suggests no significant difference between OU and Gaussian noise and favored Gaussian for simplicity. All our baselines use the same exploration backbone,



(a) Noise types

Figure C.10: **OU versus Gaussian Noise** (Section E.3). We do not see a significant difference due to this choice, and select OU noise due to better overall performance in experiments.

and we confirm that using OU noise is consistent with the available state-of-the-art results with TD3 + Gaussian noise on common environments like Ant, HalfCheetah, Hopper, and Walker2D.

F Network Architectures

F.1 Successive Actors

The entire actor is built as a successive architecture (see Figure 4.4), where each successive actor receives two pieces of information: the current state and the action list generated by preceding actors. Each action is concatenated with the state to contextualize it and then summarized using a list-summarizer, described in Section F.3. This list summary is concatenated with the state again and passed into an MLP with ReLU (3 layers for *MuJoCo* tasks and 4 layers for *MineWorld* and *RecSim*) as described in Table C.1. This MLP generates one action for each successive actor, which is subsequently used as an input action to the succeeding action lists. For discrete action space tasks, this generated action is processed with a 1-NN to find the nearest exact discrete action. Finally, the actions generated by each individual successive actor are accumulated, and the maximizer actor μ_M step from Eq. (4.5) selects the highest-valued action according to *Critic* Q-network in Section F.2.

F.2 Successive Surrogates

As Figure 4.4 illustrates, there is a surrogate network for each actor in the successive actor-architecture. Each successive critic receives three pieces of information: the current state, the action list generated by preceding actors, and the action generated by the actor corresponding to the current surrogate. Each action is concatenated with the state to contextualize it and then summarized using a list-summarizer, described in Section F.3. This list summary is concatenated with the state and the current action, and passed into a 2-layer MLP with ReLU (See Table C.1). This MLP generates the surrogate value $\hat{\Psi}_i(s, a; a_{<i})$ used as an objective to ascend over by its corresponding actor ν_i .

F.3 List Summarizers

To extract meaningful information from the list of candidate actions, we employed several list summarization methods following [Jain et al. \(2021\)](#). These methods are described below:

Bi-LSTM: The action representations of the preceding actors’ actions are first passed through a two-layer multilayer perceptron (MLP) with ReLU activation functions. The output of this MLP is then processed by a two-layer bidirectional LSTM network ([Huang et al., 2015](#)). The resulting output is fed into another two-layer MLP to create an action set summary, which serves as an input for the actor-network (Section F.1) and the surrogate network (Section F.2).

DeepSet: The action representations of the preceding actors’ actions are initially processed by a two-layer MLP with ReLU activations. The outputs are then aggregated using mean pooling over all candidate actions to compress the information into a fixed-size summary. This summary is passed through another two-layer MLP with ReLU activation to produce the action set summary, which serves as an input for the actor-network (Section F.1) and the surrogate network (Section F.2).

Transformer: Similar to Bi-LSTM, the action representations of the preceding actors’ actions are first processed by a two-layer MLP with ReLU activations. The outputs are then input into a Transformer network with self-attention and feed-forward layers to summarize the information. The resulting summary is used as part of the input to the actor-network (Section F.1) and the surrogate network (Section F.2).

F.4 Feature-wise Linear Modulation (FiLM)

Feature-wise Linear Modulation (Perez et al., 2018b) is a technique used in neural networks to condition intermediate feature representations based on external information, enhancing the network’s adaptability and performance across various tasks. FiLM modulates the features of a layer by applying learned, feature-wise affine transformations. Specifically, given a set of features \mathbf{F} , FiLM applies a scaling and shifting operation,

$$\text{FiLM}(\mathbf{F}) = \gamma \odot \mathbf{F} + \beta,$$

where γ and β are modulation parameters learned from another source (e.g., a separate network or input), and \odot denotes element-wise multiplication. This approach allows the network to selectively emphasize or de-emphasize aspects of the input data, effectively capturing complex and context-specific relationships. FiLM has been successfully applied in tasks such as visual question answering and image captioning, where conditioning visual features on textual input is essential. We apply FiLM while conditioning the actor and surrogate networks on the summary of preceding actions.

G Experiment and Evaluation Setup

G.1 Aggregated Results: Performance Profiles

To rigorously validate the aggregate efficacy of our approach, we adopt the robust evaluation methodology proposed by Agarwal et al. (2021). By incorporating their suggested performance profiles, we conduct a comprehensive comparison between our method and baseline approaches, providing a thorough understanding of the statistical uncertainties inherent in our results. Figure 4.7a shows the performance profiles across all tasks. The x-axis represents normalized scores, calculated using *min-max scaling* based on the initial performance of untrained agents aggregated across random seeds (i.e., *Min*) and the final performance from Figure 4.8 (i.e., *Max*). The results show that our method consistently outperforms the baselines across various random seeds and environments.

Our performance curve remains at the top as the x-axis progresses, while the baseline curves decline earlier. This highlights the reliability of SAVO over different environments and 10 seeds.

G.2 Implementation Details

We used PyTorch (Paszke et al., 2019) for our implementation, and the experiments were primarily conducted on workstations with either NVIDIA GeForce RTX 2080 Ti, P40, or V32 GPUs on. Each experiment seed takes about 4-6 hours for Mine World, 12-72 hours for Mujoco, and 6-72 hours for RecSim, to converge. We use the Weights & Biases tool (Biewald, 2020a) for plotting and logging experiments. All the environments were interfaced using OpenAI Gym wrappers (Brockman et al., 2016). We use the Adam optimizer (Kingma and Ba, 2014) throughout for training.

G.3 Common Hyperparameter Tuning

To ensure fairness across all baselines and our methods, we performed hyperparameter tuning over parameters that are common across methods:

- **Learning Rates of Actor and Critic:** (*Actor*) We searched over learning rates $\{0.01, 0.001, 0.0001, 0.0003\}$ and found that 0.0003 was the most stable for the actor’s learning across all tasks. (*Critic*) Similar to the actor, we searched over the same set of learning rates and found the same value of 0.0003 was the most stable for the critic’s learning across all tasks.
- **Network Sizes of Actor and Critic:** For each task, we searched over simple 3 or 4 MLP layers to determine the network size that performed best but did not observe major differences. (*Critic*) To ensure a fair comparison, we used the same network size for the critic (Q-network) and surrogates across all methods within each task. (*Actor*) Similar to the critic, we used the same network size for the various actors in all the baselines and successive actors in SAVO within a particular task.

G.4 Hyperparameters

The environment and RL algorithm hyperparameters are described in Table C.1.

Hyperparameter	Mine World	MuJoCo & Adroit	RecSim
Environment			
Total Timesteps	10^7	3×10^6	10^7
Number of epochs	5,000	8,000	10,000
# Envs in Parallel	20	10	16
Episode Horizon	100	1000	20
Number of Actions	104	N/A	10000
True Action Dim	4	5	30
Extra Action Dim	5	N/A	15
RL Training			
Batch size	256	256	256
Buffer size	5×10^5	5×10^5	10^6
Actor: LR	3×10^{-4}	3×10^{-4}	3×10^{-4}
Actor: ϵ_{start}	1	1	1
Actor: ϵ_{end}	0.01	0.01	0.01
Actor: ϵ decay steps	5×10^6	5×10^5	10^7
Actor: ϵ in Eval	0	0	0
Actor: MLP Layers	128_64_64_32	256_256	64_32_32_16
Critic: LR	3×10^{-4}	3×10^{-4}	3×10^{-4}
Critic: γ	0.99	0.99	0.99
Critic: MLP Layers	128_128	256_256	64_32
# updates per epoch	20	50	20
List Length	3	3	3
Type of List Encoder	DeepSet	DeepSet	DeepSet
List Encoder LR	3×10^{-4}	3×10^{-4}	3×10^{-4}

Table C.1: Environment/Policy-specific Hyperparameters for SAVO

H Q-Value Landscape Visualizations

H.1 1-Dimensional Action Space Environments

We analyzed the Q-value landscapes in Mujoco environments to show how successive critics help actors find better actions by reducing local optima. Figure C.11 illustrates a typically smooth

and easy-to-optimize Q-value landscape in unrestricted Inverted-Pendulum. Figure C.12 illustrates that in restricted locomotion, the Q-value landscape (leftmost and rightmost figures) is uneven with many local optima. However, the Q-value landscapes learned by successive surrogates $\hat{\Psi}_i$ become successively smoother by removing local peaks below the Q-values of previously selected actions. This helps actors find closer to optimal actions than with a single critic.

Finally, when we plot the actions a_0, a_1, a_2 selected by the learned successive actors on the original Q-landscape (rightmost figure), we see they often achieve higher Q-values than a_0 , the action a single actor has learned. Thus, the maximizer actor μ_M often finds closer to optimal actions than a single actor, resulting in better performance as shown in the return comparison between μ_M and single actor (Figure 4.11c) and the performance against baselines (Figure 4.8).

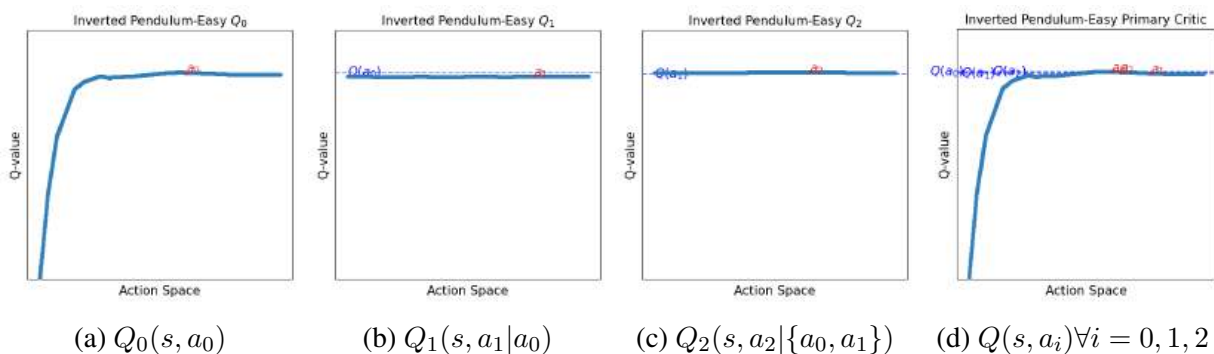


Figure C.11: Successive surrogate landscapes and the Q-landscape of Inverted Pendulum-v4.

H.2 High-Dimensional Action Space Environments

Figure C.13 visualize Q-value landscapes for a TD3 agent in Hopper-v4. We project actions from the 3D action space of Hopper-v4 onto a 2D plane using Uniform Manifold Approximation and Projection (UMAP) and sample 10,000 actions evenly to ensure thorough coverage. The Q-values are plotted using `trisurf`, which may introduce some artificial roughness but offers more reliable visuals than grid-surface plotting. Despite limitations of dimensionality reduction — such as distortion of distances — the Q-landscape for Hopper-v4 reveals a large globally optimal

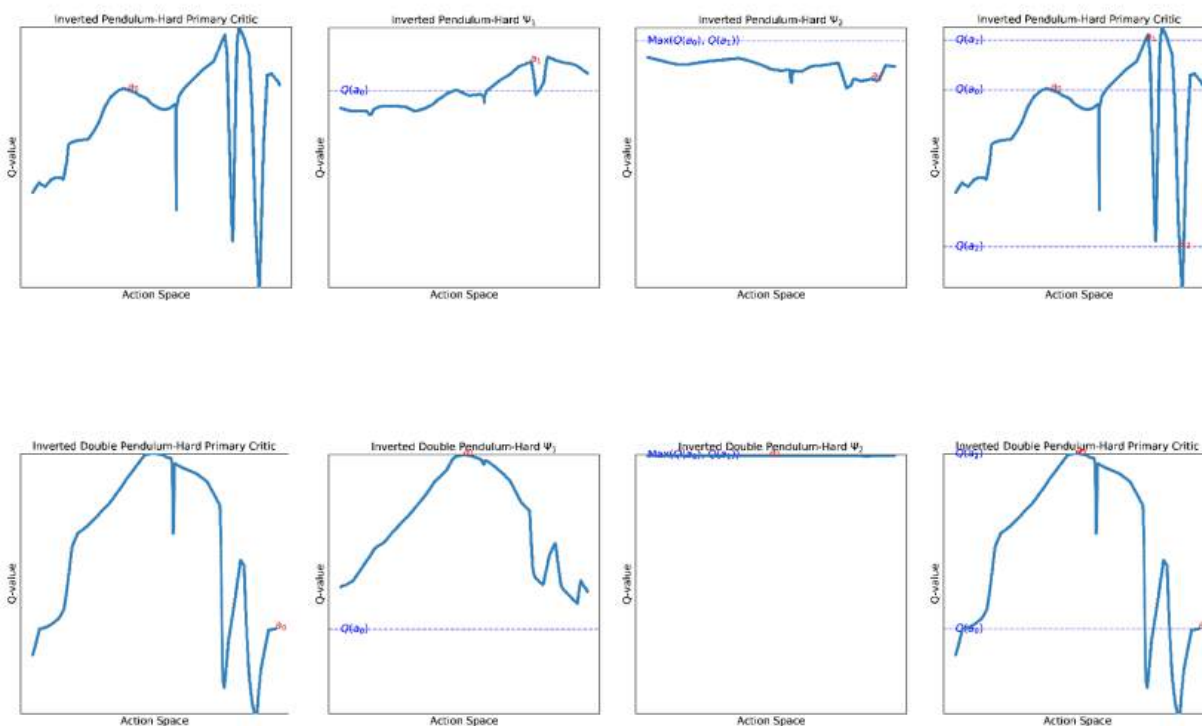


Figure C.12: Successive surrogate landscapes and Q landscape for Restricted Inverted-Pendulum and Restricted Inverted-Double-Pendulum environments.

region (shown in yellow), offering a clear gradient path that prevents the gradient-based actor from getting stuck in local optima.

In contrast, Hopper-Restricted (Figure C.14) has more complex Q-landscapes due to valid actions being restricted in one of the hyperspheres shown in Figure 4.6. Consequently, these Q-landscapes appear to have more locally optimal regions than Hopper-v4. This creates many peaks where gradient-based actors might get trapped, degrading the resultant agent performance.

The curse of dimensionality limits conclusive analyses on higher dimensional environments like Walker2D-v4 (6D) and Ant-v4 (8D) because projecting to 2D causes significant information loss, making it difficult to assess convexity in their Q-landscapes.

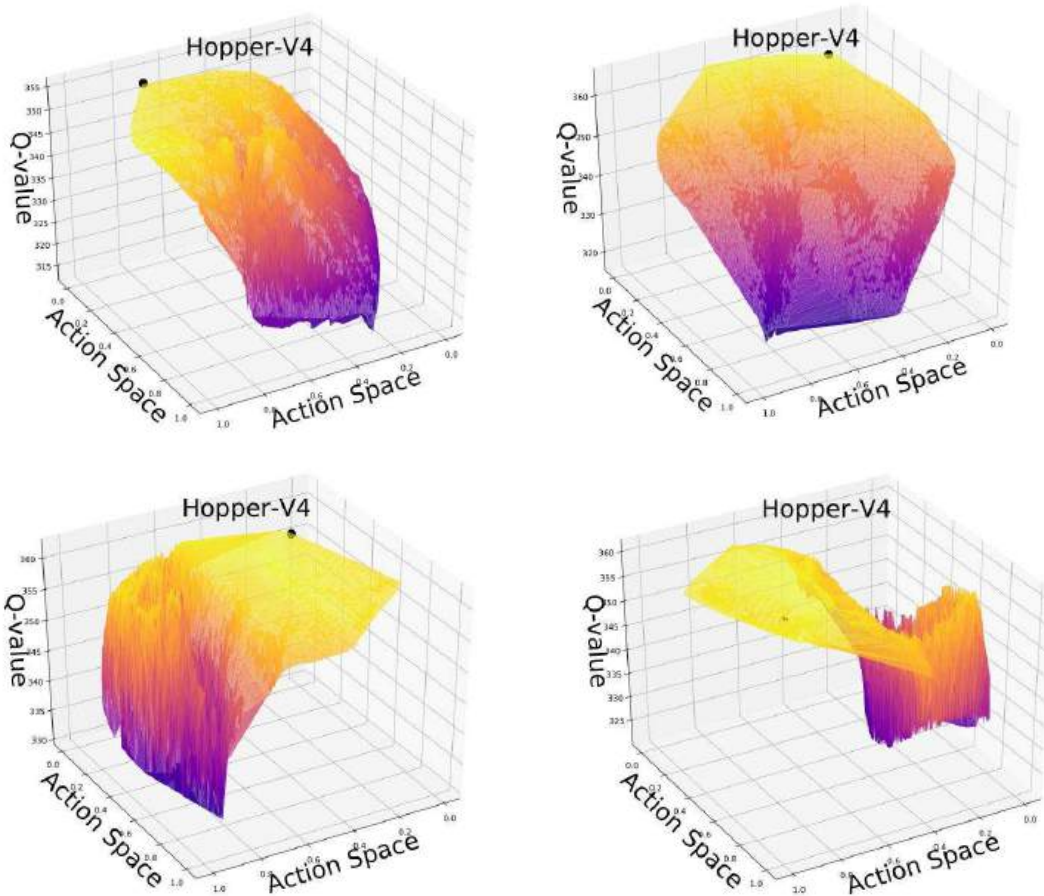


Figure C.13: Hopper-v4: Q landscape visualization at different states show a path to optimum.

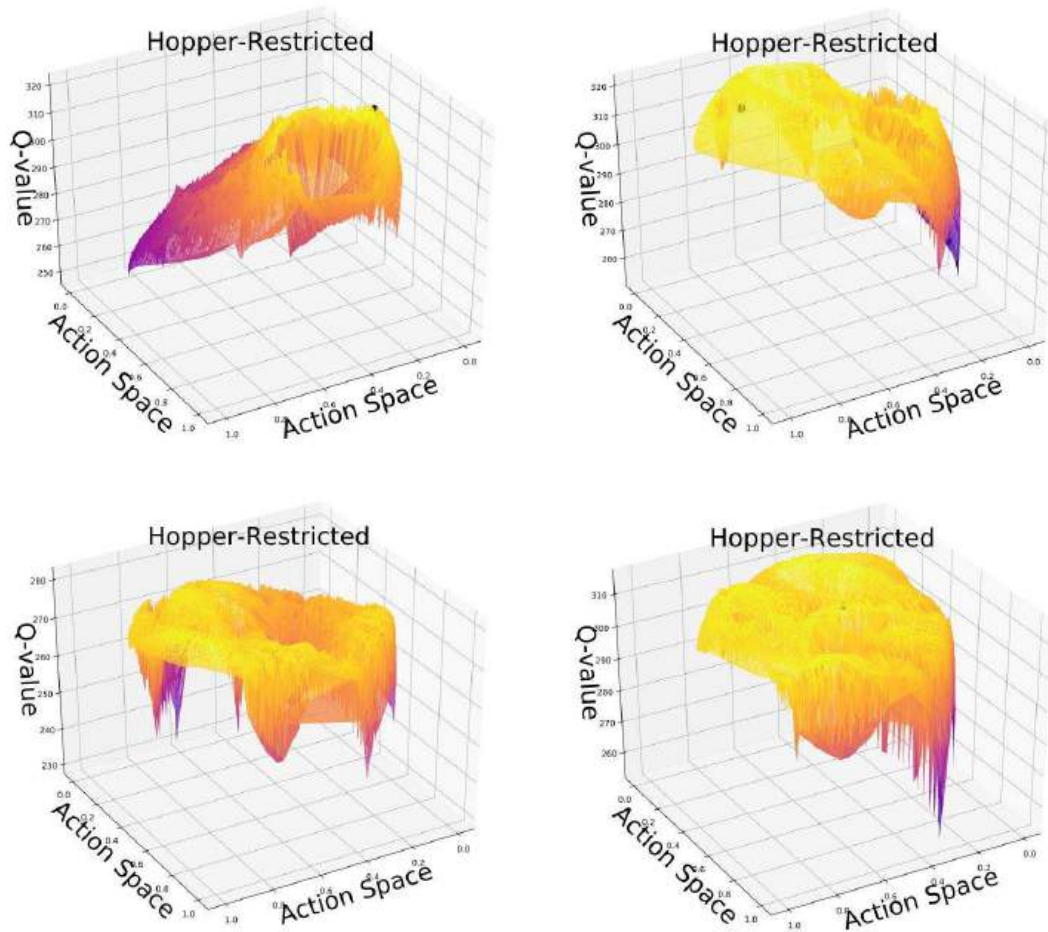


Figure C.14: Hopper-restricted: Q landscape visualization at different states show several local optima.

Appendix D

Sharing Actions in Multi-Task Reinforcement Learning via Q-switch Mixture of Policies

A Qualitative Results

The qualitative result videos are provided at <https://qmp-mtrl.github.io/>

B QMP Derivation

Following Section 5.4.2, we aim to derive the mixture-switch function f_i such that the mixture policy π_i^{mix} is guaranteed to be better than the current task’s policy π_i . We use the generalized policy iteration procedure (Sutton and Barto, 2018) underlying single-task SAC (Haarnoja et al., 2018): policy evaluation learns Q by minimizing the bellman error on the collected data, and policy improvement follows Q by minimizing the KL divergence between the new policy and the exponential of the current Q -function, $Q^{\pi^{\text{old}}}$, shown in Eq. 5.1.

In practice, the gradient updates in SAC are gradual and do not instantly achieve this optimization in Eq. 5.1, leaving a suboptimality gap to catch up to the Q -function. We observe that due to the potential similarity of some tasks in MTRL, this suboptimality gap can be bridged using other policies. Concretely, a mixture policy π_i^{mix} that selects the best policy from a set of all given policy candidates, *including the current policy*, ensures that π_i^{mix} is an improvement over π_i for the current state s :

Given a set of policies $\{\pi_1 \dots \pi_N\}$ including the current task policy π_i and a given state s , consider the following mixture policy:

$$\pi_i^{\text{mix}} = \arg \min_{\pi' \in \{\pi_i, \dots, \pi_N\}} \text{D}_{\text{KL}} \left(\pi'(\cdot | s) \left\| \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(s, \cdot))}{Z^{\pi_i}(s)} \right\| \right) \quad (\text{B.1})$$

This π_i^{mix} is a better policy improvement solution to Eq. 5.1 than π_i , because:

$$\min_{\pi' \in \{\pi_i, \dots, \pi_N\}} \text{D}_{\text{KL}} \left(\pi'(\cdot | s) \left\| \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(s, \cdot))}{Z^{\pi_i}(s)} \right\| \right) \leq \text{D}_{\text{KL}} \left(\pi_i(\cdot | \mathbf{s}_t) \left\| \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(\mathbf{s}_t, \cdot))}{Z^{\pi_i}(\mathbf{s}_t)} \right\| \right)$$

Now, we can simplify Eq. B.1 to obtain Definition 5.4.2:

$$\begin{aligned} \pi_i^{\text{mix}} &= \arg \min_{\pi' \in \{\pi_i, \dots, \pi_N\}} \text{D}_{\text{KL}} \left(\pi'(\cdot | s) \left\| \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(s, \cdot))}{Z^{\pi_i}(s)} \right\| \right) \\ &= \arg \min_{\pi' \in \{\pi_i, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} \left[\log \pi'(a | s) - \log \left\{ \frac{\exp(\frac{1}{\alpha} Q^{\pi_i}(s, a))}{Z^{\pi_i}(s)} \right\} \right] \\ &= \arg \max_{\pi' \in \{\pi_i, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} \left[-\log \pi'(a | s) + \frac{1}{\alpha} Q^{\pi_i}(s, a) - \log Z^{\pi_i}(s) \right] \\ &= \arg \max_{\pi' \in \{\pi_i, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} [-\log \pi'(a | s)] + \mathbb{E}_{a \sim \pi'(\cdot | s)} \left[\frac{1}{\alpha} Q^{\pi_i}(s, a) \right] \\ &= \arg \max_{\pi' \in \{\pi_i, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} [Q^{\pi_i}(s, a)] + \alpha \mathcal{H} [\pi'(\cdot | s)] \end{aligned}$$

Thus, the following mixture policy guarantees improvement over π_i

$$\pi_i^{\text{mix}} = \arg \max_{\pi' \in \{\pi_i, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} [Q^{\pi_i}(s, a)] + \alpha \mathcal{H} [\pi'(\cdot | s)]$$

C QMP Convergence Guarantees

We derive the convergence guarantees for *mixture soft policy iteration* used in the QMP Algorithm 4. We augment the derivation of soft policy iteration in SAC (Haarnoja et al., 2018), which

is our base algorithm, with our proposed QMP’s mixture policy. Soft policy iteration follows generalized policy iteration (Sutton and Barto, 2018) which refers to the general idea of repeated application of (1) policy evaluation to update the critics and (2) policy improvement based on the updated critics, until convergence. Like SAC, we consider the tabular setting and show that QMP’s modification to soft policy iteration converges to the optimal policy. Further, QMP can lead to an improved policy improvement step when there are shareable behaviors between tasks, consequently improving the sample efficiency. The derivation sketch follows:

1. *Soft Policy Evaluation*: QMP modifies the off-policy data collection pipeline by replacing the primary task policy π_i with the mixture policy π_i^{mix} . However, it does not affect the soft Bellman backup operator of SAC, as shown in Haarnoja et al. (2018), and therefore the Q function still converges as in SAC.
2. *Mixture Soft Policy Improvement*: QMP performs policy improvement in two steps: SAC’s policy update from $\pi_i^{\text{old}} \rightarrow \pi_i$ and applying the mixture of policies from $\pi_i \rightarrow \pi_i^{\text{mix}}$.
 - *Soft Policy Improvement*: Since QMP does *not* modify the SAC update procedure $\pi_i^{\text{old}} \rightarrow \pi_i$, we directly use SAC’s guarantees of policy improvement following Lemma 2 from Haarnoja et al. (2018).
 - *Mixture Policy Improvement*: We demonstrate QMP’s mixture policy π_i^{mix} guarantees a better policy improvement over the per-task policies π_i that compose the mixture. In Theorem C.1, we show convergence guarantee by proving that the expected return following π_i^{mix} is better than following π_i^{old} .
3. *Mixture Soft Policy Iteration*: In Theorem C.2, we show that the repeated application of the above steps in QMP converges to an optimal policy for each task. Furthermore, the convergence rate is faster because of a greedier policy improvement due to *Mixture Policy Improvement*.

For a given stochastic policy π and task $\mathbb{T}_i \in \{\mathbb{T}_1 \dots \mathbb{T}_N\}$, define V_i^π as the expected return of acting with π . Given another stochastic policy π' , define $Q_i^\pi(s, \pi'(s)) = \mathbb{E}_{a \sim \pi'(s)} Q_i^\pi(s, a)$ as the expected return of acting with π' only in s and thereafter with π .

Theorem C.1 (Mixture Soft Policy Improvement). *Consider π_i^{old} and its associated Q -function Q_i . Apply SAC's policy improvement $\pi_i^{\text{old}} \rightarrow \pi_i$ and then $\pi_i \rightarrow \pi_i^{\text{mix}}$ from Eq. 5.3. Then, $Q^{\pi_i^{\text{mix}}}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_i}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t)$ for all tasks \mathbb{T}_i and for all $(\mathbf{s}_t, \mathbf{a}_t) \in \mathcal{S} \times \mathcal{A}$ with $|\mathcal{A}| < \infty$.*

Proof. From Soft Policy Improvement, Lemma 2 of [Haarnoja et al. \(2018\)](#), we have

$$\mathbb{E}_{\mathbf{a}_t \sim \pi_i} \left[Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_i(\mathbf{a}_t | \mathbf{s}_t) \right] \geq V^{\pi_i^{\text{old}}}(\mathbf{s}_t).$$

Rewrite the difference as $\delta(\mathbf{s}_t)$,

$$\delta(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi_i} \left[Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_i(\mathbf{a}_t | \mathbf{s}_t) \right] - V^{\pi_i^{\text{old}}}(\mathbf{s}_t) \geq 0.$$

From Eq. 5.3,

$$\pi_i^{\text{mix}} = \arg \max_{\pi' \in \{\pi_1, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi'(\cdot | s)} [Q^{\pi_i}(s, a)] + \alpha \mathcal{H}[\pi'(\cdot | s)].$$

Therefore, we have a positive difference $\omega(\mathbf{s}_t)$,

$$\omega(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi_i^{\text{mix}}} \left[Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_i^{\text{mix}}(\mathbf{a}_t | \mathbf{s}_t) \right] - \mathbb{E}_{\mathbf{a}_t \sim \pi_i} \left[Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) - \log \pi_i(\mathbf{a}_t | \mathbf{s}_t) \right] \geq 0.$$

We use δ to expand the soft Bellman equation to derive the relationship between $Q^{\pi_i^{\text{old}}}$ and Q^{π_i} ,

$$\begin{aligned}
Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) &= r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V^{\pi_i^{\text{old}}}(\mathbf{s}_{t+1}) \right] \\
&= r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[\mathbb{E}_{\mathbf{a}_{t+1} \sim \pi_i} \left(Q^{\pi_i^{\text{old}}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \log \pi_i(\mathbf{a}_{t+1} | \mathbf{s}_{t+1}) \right) - \delta(\mathbf{s}_{t+1}) \right] \\
&\vdots \\
&= \underbrace{\sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{\mathbf{s}_{t+k} \sim p, \mathbf{a}_{t+k} \sim \pi_i} [r(\mathbf{s}_{t+k}, \mathbf{a}_{t+k}) - \log \pi_i(\mathbf{a}_{t+k} | \mathbf{s}_{t+k})]}_{Q^{\pi_i}(\mathbf{s}_t, \mathbf{a}_t)} - \underbrace{\sum_{k=1}^{\infty} \gamma^k \mathbb{E}_{\mathbf{s}_{t+k} \sim p} [\delta(\mathbf{s}_{t+k})]}_{\Delta_1} \\
&= Q^{\pi_i}(\mathbf{s}_t, \mathbf{a}_t) - \Delta_1
\end{aligned}$$

Likewise, we use δ and ω to derive the relationship between $Q^{\pi_i^{\text{old}}}$ and $Q^{\pi_i^{\text{mix}}}$,

$$\begin{aligned}
Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t) &= r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V^{\pi_i^{\text{old}}}(\mathbf{s}_{t+1}) \right] \\
&= r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[\mathbb{E}_{\mathbf{a}_{t+1} \sim \pi_i} \left(Q^{\pi_i^{\text{old}}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \log \pi_i(\mathbf{a}_{t+1} | \mathbf{s}_{t+1}) \right) - \delta(\mathbf{s}_{t+1}) \right] \\
&= r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[\mathbb{E}_{\mathbf{a}_{t+1} \sim \pi_i^{\text{mix}}} \left(Q^{\pi_i^{\text{old}}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \log \pi_i^{\text{mix}}(\mathbf{a}_{t+1} | \mathbf{s}_{t+1}) \right) - \delta(\mathbf{s}_{t+1}) - \omega(\mathbf{s}_{t+1}) \right] \\
&\vdots \\
&= \underbrace{\sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{\mathbf{s}_{t+k} \sim p, \mathbf{a}_{t+k} \sim \pi_i^{\text{mix}}} [r(\mathbf{s}_{t+k}, \mathbf{a}_{t+k}) - \log \pi_i^{\text{mix}}(\mathbf{a}_{t+k} | \mathbf{s}_{t+k})]}_{Q^{\pi_i^{\text{mix}}}(\mathbf{s}_t, \mathbf{a}_t)} \\
&\quad - \underbrace{\sum_{k=1}^{\infty} \gamma^k \mathbb{E}_{\mathbf{s}_{t+k} \sim p} [\delta(\mathbf{s}_{t+k})]}_{\Delta_2} - \underbrace{\sum_{k=1}^{\infty} \gamma^k \mathbb{E}_{\mathbf{s}_{t+k} \sim p} [\omega(\mathbf{s}_{t+k})]}_{\Omega} \\
&= Q^{\pi_i^{\text{mix}}}(\mathbf{s}_t, \mathbf{a}_t) - \Delta_2 - \Omega,
\end{aligned}$$

We assume that the effect of the difference $\Delta_2 - \Delta_1$ due to different state coverage is lower compared to the effect of Ω because ω is accumulated at every state, i.e., $\Delta_2 + \Omega = \Delta_1 + (\Delta_2 - \Delta_1) + \Omega \geq \Delta_1$

Since $\Delta_1, \Delta_2 \geq 0$ and $\Omega \geq 0$, we have

$$Q^{\pi_i^{\text{mix}}}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_i}(\mathbf{s}_t, \mathbf{a}_t) \geq Q^{\pi_i^{\text{old}}}(\mathbf{s}_t, \mathbf{a}_t)$$

□

Theorem C.2 (Mixture Soft Policy Iteration). *Repeated application of (i) soft policy evaluation and (ii) mixture soft policy improvement (Theorem C.1) to any $\pi_i \in \Pi$ converges to an optimal policy π_i^* such that $Q_i^{\pi_i^*}(\mathbf{s}_t, \mathbf{a}_t) \geq Q_i^{\pi_i}(\mathbf{s}_t, \mathbf{a}_t)$ for all $\pi_i \in \Pi$ and $(\mathbf{s}_t, \mathbf{a}_t) \in \mathcal{S} \times \mathcal{A}$ with $|\mathcal{A}| < \infty$. Furthermore, the sample efficiency and rate of convergence is at least as good as SAC in the presence of mixture policy improvement.*

Proof. Let π_i^k be the policy at iteration k . By SAC's soft policy iteration, the sequence $Q_i^{\pi_i^k}$ is monotonically increasing, because π_i^{mix} only modifies the online data collected and SAC is an off-policy algorithm. Thus, Theorem 1 (Soft Policy Iteration) from [Haarnoja et al. \(2018\)](#) Appendix B.3 directly applies here and proves that repeated application of soft policy evaluation and soft policy improvement converges to an optimal policy π_i^* .

Mixture soft policy improvement (Theorem C.1) shows that π_i^{mix} is a greedier policy improvement over π_i with respect to each estimate of $Q_i^{\pi_i^k}$. Thus, the expected returns in the data collected by QMP policy, $Q_i^{\pi_i^{\text{mix};k}}$, is greater than or equal to that collected by the individual task policy, $Q_i^{\pi_i^k}$. Therefore, every mixture soft policy improvement step constitutes a *larger policy improvement step* than SAC's soft policy improvement step. This makes the convergence of mixture soft policy iteration (repeated application of soft policy evaluation and Theorem C.1) an improvement over soft policy iteration.

□

D Environment Details

D.1 Multistage Reacher

We implement our multistage reacher tasks on top of the Open AI Gym (Brockman et al., 2016) Reacher environment simulated in the MuJoCo physics engine (Todorov et al., 2012) by defining a sequence of 3 subgoals per task which are specified in Table D.1. For all tasks, the reacher is initialized at the same start position with a small random perturbation sampled uniformly from $[-0.01, 0.01]$ for each coordinate. The observation includes the agent’s proprioceptive state and how many sub-goals have been reached but not subgoal locations, as they must be inferred from the respective task’s reward function. We set up the tasks to ensure that we can evaluate behavior sharing when the task rewards are qualitatively different (see Figure 5.6a):

- For every task except Task 3, the reward function is the default gym reward function based on the distance to the goal, plus an additional bonus for every subgoal completed.
- For Task 1, the reward is shifted by -2 at every timestep.
- Task 3 receives only a sparse reward of 1 for every subgoal reached.
- Task 4 has one fixed goal set at its initial position.

	Subgoal 1	Subgoal 2	Subgoal 3
T_0	(0.2, 0.3, 0.5)	(0.3, 0, 0.3)	(0.4, -0.3, 0.4)
T_1	(0.2, 0.3, 0.5)	(0.3, 0, 0.3)	(0.4, 0.3, 0.2)
T_2	(0.3, 0, 0.3)	(0.4, 0.3, 0.2)	(0.4, -0.3, 0.4)
T_3	(0.3, 0, 0.3)	(0.4, -0.3, 0.4)	(0.2, 0.3, 0.5)
T_4	initial	initial	initial

Table D.1: Coordinates of subgoal locations for each task in Multistage Reacher. Shared subgoals are highlighted in the same color. For Task 4, the only goal is to stay in the initial position.

QMP-Domain: Section 5.7.4 ablates the importance of an adaptive and state-dependent Q-switch by replacing it with a domain-dependent distribution over other tasks based on apparent task similarity. Specifically, to define the mixture probabilities for QMP-Domain in Multistage

Reacher, we use the domain knowledge of the subgoal locations of the tasks to determine the mixture probabilities. We use the ratio of *shared sub-goal sequences* between each pair of tasks (not necessarily the shared subgoals) over the total number of sub-goal sequences, 3, to calculate how much behavior must be shared between two tasks. For that ratio of shared behavior, we distribute the probability mass uniformly between all task policies that share that behavior. For Task 4, the conflicting task, we do not do any behavior sharing and only use π_4 to gather data.

Each Task \mathbb{T}_i consists of 3 sub-goal sequences $\{S_0, S_1, S_2\}$ (e.g. [initial \rightarrow Subgoal 1], [Subgoal 1 \rightarrow Subgoal 2], and [Subgoal 2 \rightarrow Subgoal 3]). For each sequence $s \in \{S_0, S_1, S_2\}$, we divide equally the contribution of each task \mathbb{T}_j 's policy π_j that shares the sequence s (i.e. if \mathbb{T}_0 and \mathbb{T}_1 both contain sequence s , where we use the notation $\mathbb{1}(s \in \mathbb{T}_i)$ as the indicator function for whether Task \mathbb{T}_i contains sequence s , then π_0 and π_1 both have $\frac{1}{2}$ contribution for s). Each sequence contributes equally to the overall mixture probabilities for Task i (i.e. all policies that shares sequence S_i contributes in total $\frac{1}{3}$ to the mixture probability for the behavior policy of Task \mathbb{T}_i). Thus, the contribution probability of Policy π_j to Task \mathbb{T}_i is:

$$p_{j \rightarrow i} = \sum_{s \in \{S_0, S_1, S_2\}} \frac{1}{3} \cdot \frac{\mathbb{1}(s \in \mathbb{T}_j)}{\sum_k \mathbb{1}(s \in \mathbb{T}_k)}$$

$$\pi_i^{\text{mix}} = \sum_j p_{j \rightarrow i} \pi_j$$

Reusing notation for mixture probabilities, we have,

$$\begin{aligned} \pi_0^{\text{mix}} &= \frac{2}{3}\pi_0 + \frac{1}{3}\pi_1 \\ \pi_1^{\text{mix}} &= \frac{1}{3}\pi_0 + \frac{2}{3}\pi_1 \\ \pi_2^{\text{mix}} &= \frac{5}{6}\pi_2 + \frac{1}{6}\pi_3 \\ \pi_3^{\text{mix}} &= \frac{1}{6}\pi_2 + \frac{5}{6}\pi_3 \\ \pi_4^{\text{mix}} &= \pi_4 \end{aligned}$$



Figure D.12: Ten tasks defined for the Maze Navigation. The start and goal locations in each task are shown in green and red spots, respectively, and an example path is shown in green.

D.2 Maze Navigation

The maze layout and dynamics follow [Fu et al. \(2020\)](#), but since their original design aims to train a single agent to reach a fixed goal from multiple start locations, we modified it to have start-goal locations fixed in each task, as in [Nam et al. \(2022\)](#). The start location is still perturbed with a small noise to avoid memorizing the task. The observation consists of the agent’s current position and velocity. It lacks the goal location, which should be inferred from the dense reward based on the distance to the goal. The action space is the target 2D velocity of the point mass agent.

The layout we used is `LARGE_MAZE` which is an 8×11 maze with paths blocked by walls. The complete set of 10 tasks is visualized in Figure D.12, where green and red spots correspond to the start and goal locations, respectively. The environment provides an agent a dense reward of $\exp(-dist)$ where $dist$ is a linear distance between the agent’s current position and the goal location. It also gives a penalty of 1 at each timestep to prevent the agent from exploiting the reward by staying near the goal. The episode terminates either as soon as the goal is reached by having $dist < 0.5$ or when 600 timesteps have passed.

D.3 Meta-World Manipulation

For Meta-World CDS, we reproduce the Meta-world environment proposed by [Yu et al. \(2021\)](#) using the Meta-world codebase ([Yu et al., 2019](#)), where the door and drawer are both placed

side-by-side on the tabletop for all tasks (see Figure 5.6c). The observation space consists of the robot’s proprioceptive state, the drawer handle state, the door handle state, and the goal location, which varies based on the task.

Unlike [Yu et al. \(2021\)](#), we additionally remove the previous state from the observation space so the policies cannot easily infer the current task, making it a challenging multi-task environment. The environment also uses the default Meta-World reward functions which is composed of two distance-based rewards: distance between the agent end effector and the object, and distance between the object and its goal location. We use this modified environment instead of the Meta-world benchmark because our problem formulation of simultaneous multi-task RL requires a consistent environment across tasks. For Meta-World MT10, we directly use the implementation provided in ([Yu et al., 2019](#)) without changes.

In both cases, the observation space consists of the robot’s proprioceptive state, locations for objects present in the environment (ie. door and drawer handle for CDS, the single target object location for MT10) and the goal location. In Meta-World CDS, due to the shared environment, there are no directly conflicting task behaviors, since the policies either go to the door or the drawer, they should ignore the irrelevant behaviors of policies interacting with the other object. In Meta-World MT10, each task interacts with a different object but in an overlapping state space so there is a mix of shared and conflicting behaviors.

D.4 Walker2D

Walker2D is a 9 DoF bipedal walker agent with the multi-task set of 4 tasks proposed and implemented by [Lee et al. \(2019\)](#): walking forward at a target velocity, walking backward at a target velocity, balancing under random external forces, and crawling under a ceiling. Each of these tasks involves different gaits or body positions to accomplish successfully without any obviously identifiable shared behavior in the optimal policies. Behavior sharing can still be effective during training to aid exploration and share helpful intermediate behaviors, like balancing. However, there is no obviously identifiable conflicting behavior either in this task set. Because each task requires

a different gait, it is unlikely for states to recur between tasks and even less likely for states that are shared to require conflicting behaviors. For instance, it is common for all policies to struggle and fall at the beginning of training, but all tasks would require similar stabilizing and correcting behavior over these states.

D.5 Kitchen

We modify the Franka Kitchen environment proposed by [Gupta et al. \(2019\)](#) and based on the implementation from [Fu et al. \(2020\)](#). Since this environment is typically used for long horizon or offline RL, we chose shorter tasks that are learnable with online RL. Furthermore, we added a dense reward based on the Meta-World reward function. We formed our 10 task MTRL set by choosing 10 available tasks in the kitchen environment that interacted with the same objects: turning the top burner on or off, turning the bottom burner on or off, turning the light switch on and off, open or closing the sliding cabinet, and opening and closing the hinge cabinet. The observation space consists of the robot’s state, the location of the target object, and the goal location for that object. Similar to the Meta-World CDS environment, these tasks may share behaviors navigating around the kitchen to the target object but have plenty of irrelevant behavior between tasks that interact with different objects and conflicting behaviors when opening or closing the same object.

E Additional Results

E.1 Multistage Reacher Per Task Results

Additional results and analysis on Multistage Reacher are shown in Figure D.8. QMP outperforms all the baselines in this task set, as shown in Figure 5.8. Task 3 receives only a sparse reward and, thus, can benefit the most from shared exploration. We observe that QMP gains the most performance boost due to selective behavior-sharing in Task 3. The No-Shared-Behavior baseline is unable to solve Task 3 at all due to its sparse reward nature. The other baselines which share

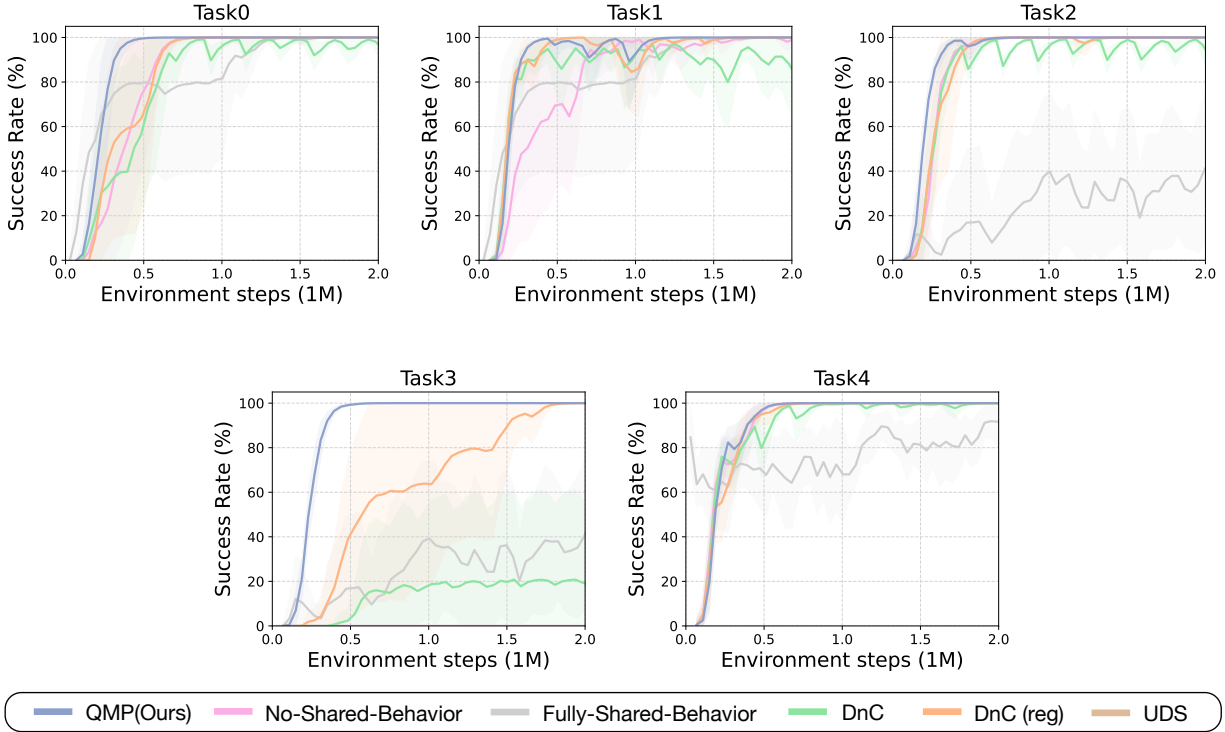


Figure D.8: Success rates for individual tasks in Multistage Reacher. Our method especially helps in learning Task 3, which requires extra exploration because it only receives a sparse reward.

uniformly suffer at Task 3, likely because they also share behaviors from other conflicting tasks, especially Task 4. We explore this further in the following Section F.

For all tasks, QMP outperforms or is comparable to No-Shared-Behavior, which shows that selective behavior-sharing can help accelerate learning when task behaviors are shareable and is robust when tasks conflict. Fully-Shared-Behavior especially underperforms in Tasks 2 and 3, which require conflicting behaviors upon reaching Subgoal 1, as defined in Table D.1. In contrast, it excels at the beginning of Task 0 and Task 1 as their required behaviors are completely shared. However, it suffers after Subgoal 2, as the task objectives diverge.

E.2 Data Sharing Results

In Figure D.10c, we report multiple sharing percentiles for UDS and for CDS (Yu et al., 2021), which assumes access to ground truth task reward functions, which it uses to re-label the shared data.

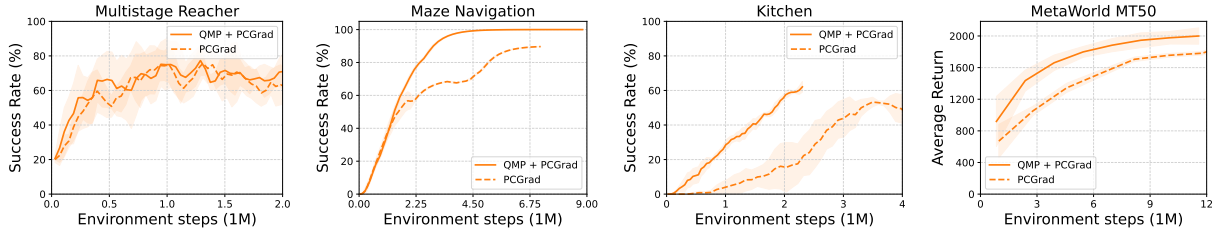
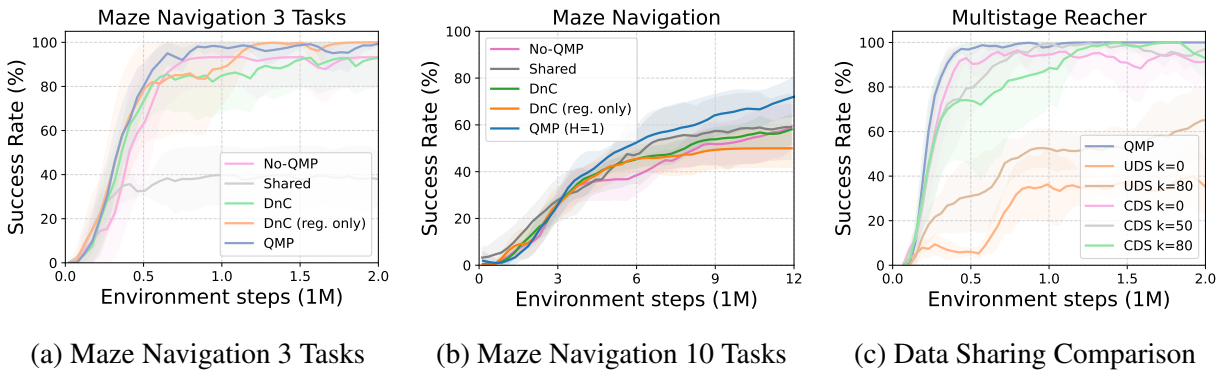


Figure D.9: Combining QMP with PCGrad yields complementary improvement in 3 out of the 4 environments we tested on. Dashed lines are PCGrad only and solid lines are QMP + PCGrad.



(a) Maze Navigation 3 Tasks (b) Maze Navigation 10 Tasks (c) Data Sharing Comparison

Figure D.10: QMP scales well from (a) 3 tasks to (b) 10 tasks in Maze Navigation, especially in comparison to other behavior sharing methods. (c) Online data sharing is very efficient when given task reward functions (all CDS versions), but suffers without (all UDS versions).

When the shared data is relabeled with task reward functions, thereby bypassing the conflicting behavior problem, online data sharing approaches can work very well. But when unsupervised, we see that online data sharing can actually harm performance in environments with conflicting tasks, with the more conservative data sharing approach (UDS $k=80$) out-performing sharing all data. k is the percentile above with we share a transition between tasks, with higher k representing more conservative data sharing. Details on our online UDS and CDS implementation are in Section H.6

E.3 PCGrad Results

We evaluate whether QMP combined with PCGrad (Yu et al., 2020) results in complementary benefits. PCGrad is a popular MTRL algorithm that learns a policy with shared parameters and alleviates negative interference between tasks by modifying the multi-task gradients. We see

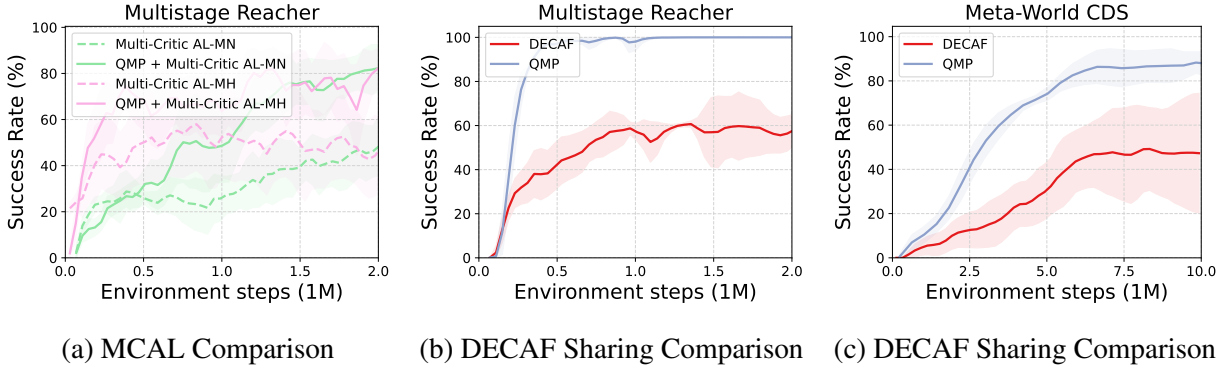


Figure D.11: Combining our method with another parameter sharing method, MCAL, shows complementary benefits in (a). Our method outperforms DECAF in Multistage Reacher (b) and Meta-World CDS(c), demonstrating that learning to directly use Q-functions from other tasks is more challenging and sample inefficient than using the current task’s Q-function to evaluate other tasks’ policies.

in Figure D.9 that QMP + PCGrad significantly improves PCGrad performance in 3 out of 4 environments.

E.4 QMP Scales with Task Set Size in Maze Navigation

We look at the behavior sharing methods in the Maze Navigation task for a task set with 3 tasks (Figure D.10a) and 10 tasks (Figure D.10b) and see that QMP scales well from 3 to 10 tasks, even compared to other behavior sharing methods. Similar to Meta-World, we hypothesize QMP scales better with a larger task set size of similar tasks due to there being more shareable behaviors between tasks. We see that by selectively sharing behaviors, QMP is able to identify and share helpful behaviors in the larger tasks sets whereas other behavior sharing methods struggle.

E.5 Additional Comparisons

Multi-Critic Actor Learning (MCAL) (Mysore et al., 2022) is a parameter sharing MTRL method that aims to tackle potential negative interference between tasks by learning separate critics for each task while training a single multi-task actor. We add QMP to two variants of MCAL, Multi-Critic AL-MN which maintains separate networks for each critic and Multi-Critic AL-MH

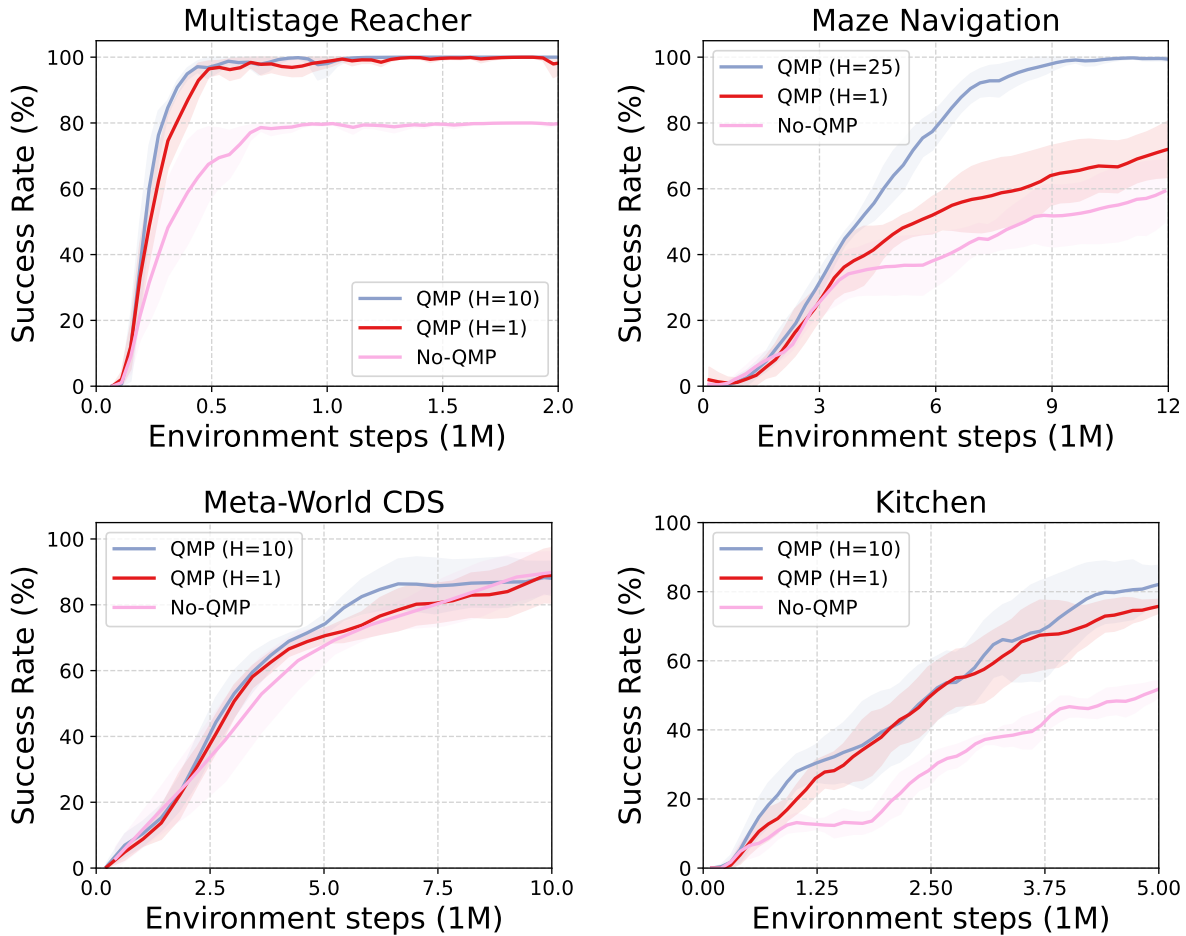


Figure D.12: In each case above, QMP with H-step rollouts of the behavioral policy (blue) performs no worse than QMP with 1-step rollouts (red), with the H-step rollouts helping significantly in some tasks. Additionally both versions of QMP outperform the No-QMP baseline.

which uses a single multi-head network for the critic, in Multistage Reacher in Figure D.11 a. We see that adding QMP provides around a 20% final success rate gain in both variants and is more sample efficient.

We also compare our method with DECAF (Glatt et al., 2020), a MTRL method which shared Q-functions between tasks instead of behavioral policies. DECAF learns task specific weights to linearly combine the task Q-functions which is used to train the task policy. In contrast, our method uses the task Q-function to evaluate different tasks’ policies to incorporate into the task’s behavioral policy. Our method only modifies the data collection process, not the RL objective, and does not

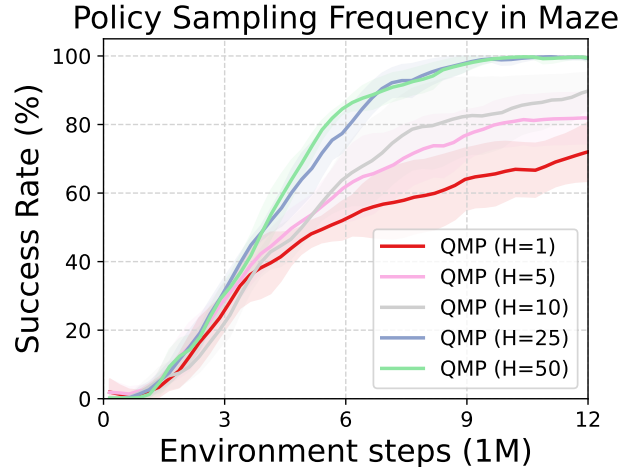


Figure D.13: QMP consistently improves performance as H increases in Maze.

have a learned component. In Multistage Reacher (Figure D.11b) and Meta-World CDS (Figure D.11c), we see that QMP outperforms DECAF by more than 20% final success rate.

E.6 Temporally-Extended Behavior Sharing

Motivated by prior work in hierarchical RL (Machado et al., 2017; Jinnai et al., 2019b;a; Hansen et al., 2019; Zhang et al., 2020) and skill learning (Pertsch et al., 2021), we explore temporally extended behavior sharing by simply following the actions of the policy π_j selected by π_{mix}^i for H steps before re-evaluating π_{mix}^i . Furthermore, a recent work Xu et al. (2024) provides theoretical results that shows myopic (ϵ -greedy) policy sharing can be sample efficient in sufficiently diverse multi-task settings, providing theoretical support for temporally extended multi-task behavior sharing in some settings.

We study the effect of sharing temporally extended behaviors of length H in Maze Navigation in Figure D.13, by rolling out the chosen task policy for 1, 5, 10, 25, and 50 timesteps. We see that performance improves when sharing longer behaviors (25 and 50 timesteps) which are more coherent and temporally extended. This is true even though we choose the behavioral policy greedily, only evaluating the current state s every H steps. Importantly, the guarantees from Theorem C.1 do not extend to H -step policy roll-outs and increasing H does not help in all environments. We compare the performance of No-QMP, QMP, and QMP with temporally extended behavior sharing

where we choose the best performance out of $H = 10$ and $H = 25$ in Table D.2 and Figure D.12. Nevertheless, the impressive results in Maze suggest that multi-task temporally extended behavior sharing is worth exploring in future work.

Environment	H-value	No-QMP	QMP	QMP (H_{i1})
Reacher	10	80 ± 0	100 ± 0	100 ± 0
Maze	25	57.9 ± 0.09	72.9 ± 0.1	99.9 ± 0.0
MT-CDS	10	97.5 ± 4.5	93.7 ± 8.5	98.8 ± 2.0
MT10	10	79.1 ± 5.97	89.0 ± 0.01	$82. \pm 4.48$
Kitchen	10	65.5 ± 11.0	77.3 ± 5.3	84.5 ± 8.7
Walker	10	3110 ± 220	3205 ± 218	3310 ± 203

Table D.2: Temporally Extended Behavior Sharing

F QMP Behavior Sharing Analysis

QMP learns to not share from conflicting tasks: We visualize the mixture probabilities per task of other policies in Figure D.14 for Multistage Reacher, highlighting the conflicting Task 4 in red. Throughout training, we see that QMP learns to share less behavior from Policy 4 than other policies in Tasks 0-3 and shares the least total cross-task behavior in Task 4. This supports our claim that the Q-switch can identify conflicting behaviors that should not be shared. We note that Task 3 has a relatively larger amount of sharing than other tasks. Since Task 3 has sparse rewards, it benefits the most from exploration via selective behavior-sharing from other tasks.

Figure D.15 analyzes the effectiveness of the Q-switch in identifying shareable behaviors by visualizing the average proportion that each task policy is selected for another task over the course of training in the Multistage Reacher environment. This average mixture composition statistic intuitively analyzes whether QMP identifies shareable behaviors between similar tasks and avoids behavior sharing between conflicting or irrelevant tasks. As we expect, the Q-switch for Task 4 utilizes the least behavior from other policies (bottom row), and Policy 4 shares the least with other tasks (rightmost column). Since the agent at Task 4 is rewarded to stay at its initial position, this behavior conflicts with all the other goal-reaching tasks. Of the remaining tasks, Task 0 and 1 share

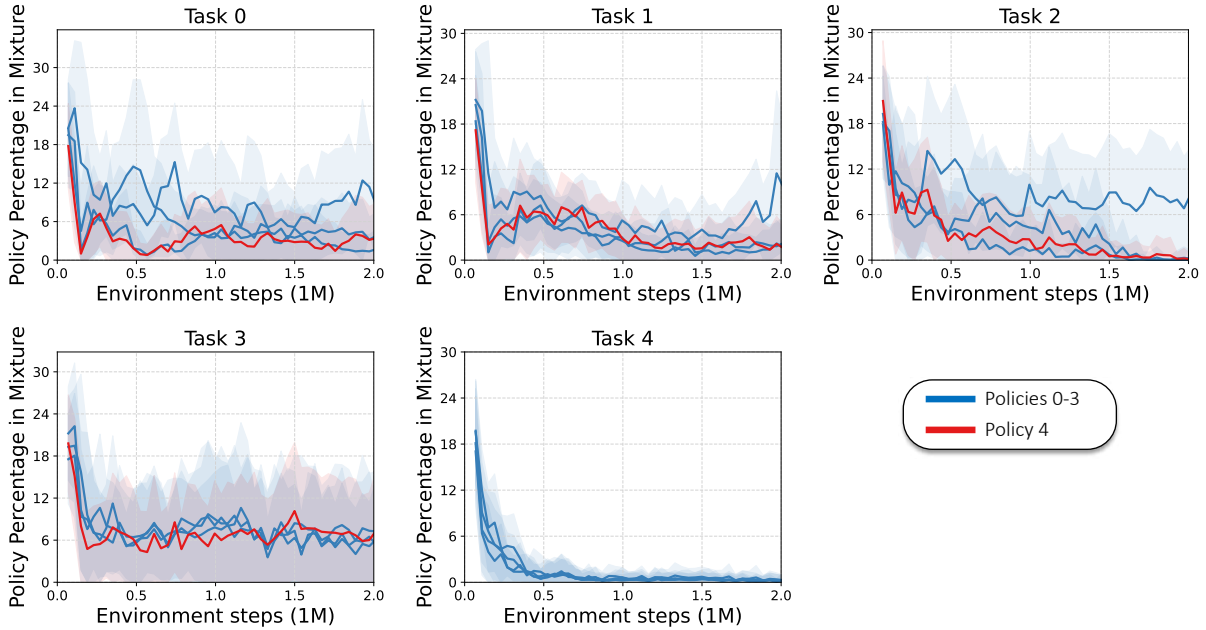


Figure D.14: Mixture probabilities per task of other policies over the course of training for Multistage Reacher. The conflicting task Policy 4, which requires staying stationary, is highlighted in red.

the most similar goal sequence, so it is intuitive why they benefit from shared exploration and are often selected by their respective Q-switches. Finally, unlike the other tasks, Task 3 receives only a sparse reward and therefore relies heavily on shared exploration. In fact, QMP demonstrates the greatest advantage in this task (Appendix Figure D.8).

Behavior-sharing reduces over training: Figure D.14 shows that the total amount of behavior-sharing decreases over the course of training in all tasks, which demonstrates a naturally arising preference in the Q-switch for the task-specific policy as it becomes more proficient at its own task.

F.1 Qualitative Visualization of Behavior-Sharing

We qualitatively analyze behavior sharing by visualizing a rollout of QMP during training for the Drawer Open task in Meta-World Manipulation (Figure 5.9b). To generate this visualization, we use a QMP rollout during training before the policy converges to see how behaviors are shared and aid learning. For clarity, we first subsample the episodes timesteps by 10 and only report timesteps when the activated policy changes to a new one (ie. from timestep 80 to 110, QMP chose the Drawer Open policy). We qualitatively break down the episode into when the agent is approaching the

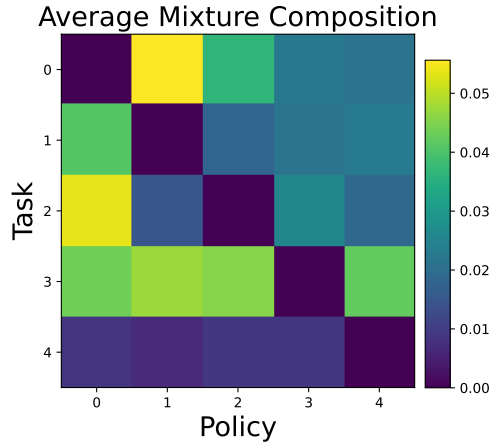


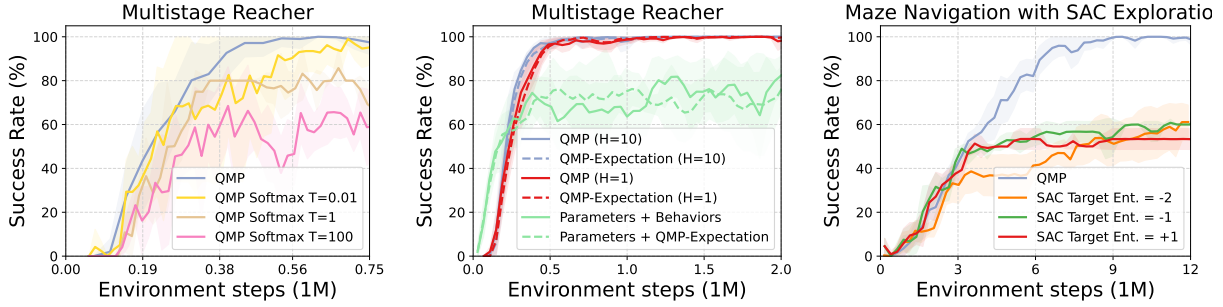
Figure D.15: Average contribution of each Policy j (col j) in each Task i 's (row i) data collection on Reacher Multistage (diagonal zeroed for contrast).

drawer (top row; Steps 1-60), grasping the handle (top row; Steps 61-80), and pulling the drawer open (bottom row). This allows us to see that it switches between all task policies as it approaches the drawer, uses drawer-specific policies as it grasps the handle, and opening-specific policies as it pulls the drawer open. This suggests that in addition to ignoring conflicting behaviors, QMP is able to identify helpful behaviors to share. We note that QMP is not perfect at policy selection throughout the entire rollout, and it is also hard to interpret these shared behaviors exactly because the policies themselves are only partially trained, as this rollout is from the middle of training. However, in conjunction with the overall results and analysis, this supports our claim that QMP can effectively identify shareable behaviors between tasks.

G Additional Ablations and Analysis

G.1 Probabilistic Mixture v/s Arg-Max

A probabilistic mixture of policies is a design choice of our approach where arg-max is replaced with softmax. However, in our initial experiments, we found no significant improvement in performance and it came with an additional hyperparameter of tuning the temperature coefficient. As we see in Figure D.16a, QMP actually outperforms a probabilistic mixture over a range of



(a) Probabilistic Mixture Ablation (b) Expected Q-value Approximations (c) Single-Task Exploration

Figure D.16: (a) Using probabilistic mixtures with QMP by using a softmax over Q values with temperature T, which determines the spread of the distribution. (b) Across different QMP versions, evaluating mean policy actions (solid lines) vs. sampling 10 actions to estimate expected Q-values (dashed lines) result in similar performance. (c) Single-task exploration by varying SAC target entropy. QMP reaches a higher success rate because it shares exploratory behavior **across** tasks.

softmax temperatures, justifying the design choice of argmax over softmax due to its reliable performance and simplicity.

G.2 Approximation Expected Q-value Over Policy Action Distribution

QMP’s behavior policy is defined as $\pi_i^{\text{mix}} = \arg \max_{\pi_j \in \{\pi_1, \dots, \pi_N\}} \mathbb{E}_{a \sim \pi_j(s)} Q_i(s, a)$, which picks the task policy with the best expected Q value over its action distribution. We approximate the expectation by evaluating the Q-value of only the mean of each policy’s action distribution which is computationally cheaper $\pi_i^{\text{mix}} \approx \arg \max_{\pi_j \in \{\pi_1, \dots, \pi_N\}} Q_i(s, \mathbb{E}_{a \sim \pi_j(s)}[a])$. We compare this to a empirical estimate that samples 10 actions from the policy distribution and picks the policy with highest average Q-value in Figure D.16b, and find no significant performance difference between the two approximations. This validates that our simple approximation works well in practice, which we hypothesize is due to the low variance of the task policies.

G.3 QMP v/s Increasing Single Task Exploration

Since QMP seeks to gather more informative training data for the task by modifying the behavioral policy, it can be viewed as a form of multi-task exploration. We briefly investigate

how single task exploration differs from multi-task exploration by tuning the target entropy in SAC in Figure D.16c which influences the policy entropy and therefore exploration. We see that while tuning this exploration parameter affects the sample efficiency by more quickly learning each individual task, QMP achieves a higher final success rate by incorporating behaviors from other tasks, and therefore doing multi-task exploration. The benefit of exploration or behavior sharing algorithms specialized for multi-task RL is precisely this ability to transfer and share information between tasks.

G.4 QMP Runtime

While QMP does require more policy and q-function evaluations to sample from π_{mix}^i in comparison to the base RL method, these evaluations can be greatly parallelized and do not significantly increase runtime (see Figure D.3) for average runtimes for our experiments). Each sample from π_{mix}^i requires querying N policy proposals and N Q-values. In QMP + Parameter-Sharing, thanks to the multihead architectures of the policy and Q-networks, all N evaluations are done in one single pass. Thus, with two passes through neural networks, we can get N action candidates and their N Q-values. Therefore, the increase in time is negligible. Even without parameter-sharing, $Q_i(s, a_j)$ evaluations can be batched $\forall j$ and the policy evaluations $\pi_j(a_j|s)$ are all independent, and can be obtained in parallel. In our implementation, we batch the Q evaluations, but do not parallelize the policy evaluations.

Table D.3: Runtime Comparison of MTRL frameworks with and without QMP

Environment	No-Sharing	QMP + No-Sharing	Parameter-Sharing	QMP + Parameter-Sharing
Reacher Multistage	12.5 hr	14.2 hr	14 hr	16.2 hr
MT50	–	–	7 days, 3hr	7 days, 6 hr

H Implementation Details

The SAC implementation we used in all our experiments is based on the open-source implementation from Garage ([garage contributors, 2019](#)). We used fully connected layers for the policies and Q-functions with the default hyperparameters listed in Table D.4. For DnC baselines, we reproduced the method in Garage to the best of our ability with minimal modifications.

We used PyTorch ([Paszke et al., 2019](#)) for our implementation. We run the experiments primarily on machines with either NVIDIA GeForce RTX 2080 Ti or RTX 3090. Most experiments take around one day or less on an RTX 3090 to run. We use the Weights & Biases tool ([Biewald, 2020b](#)) for logging and tracking experiments. All the environments were developed using the OpenAI Gym interface ([Brockman et al., 2016](#)).

H.1 Hyperparameters

Table D.4 details the list of important hyperparameters on all the 3 environments. For all environments, we used a 2 layer fully connected network with hidden dimension 256 and a tanh activation function for the policies and Q functions. We use a target network for the Q function with target update $\tau = 0.995$ and trained with an RL discount of $\gamma = 0.99$.

H.2 No-Shared-Behaviors

All N networks have the same architecture with the hyperparameters presented in Table D.4.

H.3 Fully-Shared-Behaviors

Since it is the only model with a single policy, we increased the number of parameters in the network to match others and tuned the learning rate. The hidden dimension of each layer is 600 in Multistage Reacher, 834 in Maze Navigation, and 512 in Meta-World Manipulation, and we kept the number of layers at 2. The number of environment steps as well as the number of gradient steps per update were increased by N times so that the total number of steps could match those in other

Hyperparameter	Multistage Reacher	Maze Navigation	Meta-World CDS
Minimum buffer size (per task)	10000	3000	10000
# Environment steps per update (per task)	1000	600	500
# Gradient steps per update (per task)	100	100	50
Batch size	32	256	256
Learning rates for π , Q and α	0.0003	0.0003	0.0015

Hyperparameter	Meta-World MT10	Walker	Kitchen
Minimum buffer size (per task)	500	2500	200
# Environment steps per update (per task)	500	1000	200
# Gradient steps per update (per task)	50	1500	50
Batch size	2560	256	1280
Learning rates for π , Q and α	0.0015	0.0003	0.0003

Table D.4: Hyperparameters for QMP and baselines.

models. For the learning rate, we tried 4 different values (0.0003, 0.0005, 0.001, 0.0015) and chose the most performant one. The actual learning rate used for each experiment is 0.0003 in Multistage Reacher and Maze Navigation, and 0.001 in Meta-World Manipulation.

This modification also applies to the Shared Multihead baseline, but with separate tuning for the network size and learning rates. In Multistage Reacher, we used layers with hidden dimensions of 512 and 0.001 as the final learning rate. In Maze Navigation, we used 834 for hidden dimensions and 0.0003 for the learning rate.

H.4 DnC

We used the same hyperparameters as in Separated, while the policy distillation parameters and the regularization coefficients were manually tuned. Following the settings in the original DnC (Ghosh et al., 2018), we adjusted the period of policy distillation to have 10 distillations over the course of training. The number of distillation epochs was set to 500 to ensure that the distillation is completed. The regularization coefficients were searched among 5 values (0.0001, 0.001, 0.01, 0.1, 1), and we chose the best one. Note that this search was done separately for DnC and DnC

with regularization only. For DnC, the coefficients we used are: 0.001 in Multistage Reacher and Maze Navigation, and 0.001 in Meta-World Manipulation. For Dnc with regularization only, the values are: 0.001 in Multistage Reacher, 0.0001 in Maze Navigation, and 0.001 in Meta-World Manipulation.

H.5 QMP (Ours)

Our method also uses the default hyperparameters. QMP does not require any task specific hyperparameters. The exception is Meta-World MT10, where we found it helpful to have more conservative behavior sharing by choosing the task-specific policy 70% of the time. The remaining 30% we use the Q-filter to select a policy as usual.

Like in Baseline Multihead (Parameters-Only), the QMP Multihead architecture (Parameters+Behaviors) also required a separate tuning. Since QMP Multihead effectively has one network, we increased the network size in accordance with Baseline Multihead and tuned the learning rate in addition to the mixture warmup period. The best-performing combinations of these parameters we found are 0 and 0.001 in Multistage Reacher, and 100 and 0.0003 in Maze Navigation, respectively.

H.6 Online UDS

[Yu et al. \(2022\)](#) proposes an offline multi-task RL method (UDS) that shares data between tasks if their conservative Q value falls above the k^{th} percentile of the task data. Specifically, before training, you would go through all the tasks' data and share some data from Task j to Task i if the Task i Q value of that data is greater than $k\%$ of the Q values of Task i 's data. UDS does not require access to task reward functions like other data-sharing approaches. It simply re-labels any shared data with the minimum task reward, making it applicable to our problem setting as we also do not assume that reward relabeling is possible.

In order to adapt UDS to online RL, instead of doing data sharing once on the given multi-task dataset, we apply UDS data sharing before every training iteration to the data in the multi-task replay buffers. Concretely, we implement this on-the-fly for every batch of sampled data by sampling one

batch of data from Task i 's replay buffer, β_i , and one batch of data from the other task's replay buffers $\beta_{j \neq i}$. Then following UDS, we would form the effective batch β_i^{eff} by sharing data from $\beta_{j \neq i}$ if it falls above the k^{th} percentile of Q values for β_i :

$$UDS_{\text{online}} : (s, a, r_i, s') \sim \beta_{j \neq i} \in \beta_i^{\text{eff}}$$

$$\text{if } \Delta^\pi(s, a) := \hat{Q}^\pi(s, a, i) - P_{k^{\text{th}}}[\hat{Q}^\pi(s', a', i) : s', a' \sim \beta_i] \geq 0$$

Note the differences here: (i) the ‘data’ used for data-sharing is the sampled replay buffer batch instead of the offline dataset, and (ii) we use the standard Q-function to evaluate data instead of the conservative Q-function since we are doing online (not offline) RL. We implement it this way as a practical approximation to avoid having to process the entire replay buffer every training iteration.

We use the same default hyperparameters as the other baseline methods. Additionally, we need to tune the sharing percentile k . For this, we tried 0th percentile (sharing all data) and 80th percentile, and chose the best-performing one.