

APPENDIX

Our code is provided in the supplementary material to facilitate reproducibility.

A THEORETICAL RESULTS

In this section, we will present the proofs for all the Theorems and Corollaries stated in Section 4 and 6.

A.1 PROOF OF THEOREM 4.1

Theorem 4.1. All possible state-action visitation distributions in an MDP form an affine set.

Proof. Any state-action visitation distribution, $d^\pi(s, a)$ must satisfy the Bellman Flow equation:

$$\sum_a d^\pi(s, a) = (1 - \gamma)\mu(s) + \gamma \sum_{s', a'} \mathbb{P}(s|s', a') d^\pi(s', a'). \quad (11)$$

This equation can be written in matrix notation as:

$$\sum_a d^\pi = (1 - \gamma)\mu + \gamma P^T d^\pi. \quad (12)$$

Rearranging the terms,

$$(S - \gamma P^T) d^\pi = (1 - \gamma)\mu, \quad (13)$$

where S is the matrix for \sum_a of size $|\mathcal{S}| \times |\mathcal{S}||\mathcal{A}|$ with only $|\mathcal{A}|$ entries set to 1 corresponding to the state denoted by the row. This equation is an affine equation of the form $Ax = b$ whose solution set forms an affine set. Hence all state-visitiation distributions d^π form an affine set. \square

A.2 PROOF OF COROLLARY 4.2

Corollary 4.2. Any successor measure, M^π , in an MDP forms an affine set and so can be represented as $\sum_i^d \phi_i w_i^\pi + b$ where ϕ_i and b are independent of the policy π and d is the dimension of the affine space.

Proof. Using Theorem 4.1, we have shown that state-action visitation distributions form affine sets. Similarly, successor measures, $M^\pi(s, a, s^+, a^+)$ are solutions of the Bellman Flow equation:

$$M^\pi(s, a, s^+, a^+) = (1 - \gamma)\mathbb{1}[s = s^+, a = a^+] + \gamma \sum_{s', a' \in \mathcal{S}\mathcal{A}} P(s^+|s', a') M^\pi(s, a, s', a') \pi(a^+|s^+). \quad (14)$$

Taking summation over a^+ on both sides gives us an equation very similar to Equation 11 and so can be written by rearranging as,

$$(S - \gamma P^T) M^\pi = (1 - \gamma)\mathbb{1}[s = s^+]. \quad (15)$$

With similar arguments as in Lemma 4.1, M^π also forms an affine set. Any element x of an affine set can be written as $\sum_i^d \phi_i w_i + b$ where $\langle \phi_i \rangle$ are the basis and b is a bias vector. The basis is given by the null space of the matrix operator $(S - \gamma P^T)$. Since the operator $(S - \gamma P^T)$ and the vector $(1 - \gamma)\mathbb{1}[s = s^+]$ are independent of the policy, the basis Φ and the bias b are also independent of the policy. \square

A.3 PROOF OF THEOREM 4.4

Theorem 4.4. For the same dimensionality, $\text{span}\{\Phi^{vf}\}$ represents the set of the value functions spanned by Φ^{vf} and $\{\text{span}\{\Phi\}r\}$ represents the set of value functions using the successor measures spanned by Φ , $\text{span}\{\Phi^{vf}\} \subseteq \{\text{span}\{\Phi\}r\}$.

Proof. We need to show that any element that belongs to the set $\{\text{span}\{\Phi\}r\}$ also belongs to the set $\text{span}\{\Phi^{vf}\}$.

810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863

$$V^\pi(s) = \sum_i \beta_i^\pi \Phi_i^{vf}(s).$$

If we assume a special $\Phi_i(s, s') = \sigma_i(s)\eta_i(s')$,

$$\begin{aligned} V^\pi(s) &= \sum_i w_i^\pi \sum_{s'} \Phi(s, s')r(s') \\ &= \sum_i [w_i^\pi \sum_{s'} \eta_i(s')r(s')] \sigma_i(s). \end{aligned}$$

The two equations match with $\beta_i^\pi = w_i^\pi \sum_{s'} \eta_i(s')r(s')$ and $\sigma_i(s) = \Phi_i^{vf}(s)$. This implies for every instance in the span of Φ^{vf} , there exists some instance in the span of Φ . \square

A.4 PROOF OF THEOREM 6.1

Theorem 6.1. Successor Features $\psi^\pi(s, a)$ belong to an affine set and can be represented using a linear combination of basis functions and a bias.

Proof. Given basic state features, $\varphi : S \rightarrow \mathbb{R}^d$, the successor feature is defined as, $\psi^\pi(s, a) = \mathbb{E}_\pi[\sum_t \gamma^t \varphi(s_{t+1})]$. It can be correspondingly connected to successor measures as $\psi^\pi(s, a) = \sum_{s'} M(s, a, s')\varphi(s')$ (replace $\sum_{s'}$ with $\int_{s'}$ for continuous domains). In Linear algebra notations, let M^π be a $(S \times A) \times S$ dimensional matrix representing successor measure. Define Φ_s as the $S \times d$ matrix containing φ for each state concatenated row-wise. The $(S \times A) \times d$ matrix representing Ψ^π can be given as,

$$\begin{aligned} \Psi^\pi &= M^\pi \Phi_s \\ \implies \Psi^\pi &= \sum_i \phi_i w_i^\pi \Phi_s \quad (M^\pi \text{ is affine for basis } \phi) \\ \implies \Psi^\pi &= \sum_{s'} \sum_i \phi_i(\cdot, \cdot, s') w_i^\pi \varphi(s') \\ \implies \Psi^\pi &= \sum_i \sum_{s'} \phi_i(\cdot, \cdot, s') \varphi(s') w_i^\pi \\ \implies \Psi^\pi &= \sum_i \phi_{\psi, i} w_i^\pi \quad (\phi_\psi = \sum_{s'} \phi_i(\cdot, \cdot, s') \varphi(s')) \\ \implies \Psi^\pi &= \Phi_\psi w^\pi \end{aligned}$$

Hence, the successor features are affine with policy independent basis Φ_ψ . \square

A.5 PROOF OF THEOREM 6.3

Theorem 6.3. If $M^\pi(s, a, s^+) = \phi(s, a, s^+)w^\pi$ and $\phi(s, a, s^+) = \phi_\psi(s, a)^T \phi_s(s^+)$, the successor feature $\psi^\pi(s, a) = \phi_\psi(s, a)w^\pi$ for the basic feature $\phi_s(s)^T (\phi_s \phi_s^T)^{-1}$.

Proof. Consider $\phi(s, a, s^+) \in \mathbb{R}^d$ as the set of $d - 1$ basis vectors and the bias with $w^\pi \in \mathbb{R}^d$ being the $d - 1$ weights to combine the basis and $w_d^\pi = 1$. Clearly from Theorem 4.2, $M^\pi(s, a, s^+)$ can be represented as $\phi(s, a, s^+)w^\pi$. Further, $\phi(s, a, s^+) = \phi_\psi(s, a)^T \phi_s(s^+)$ where $\phi_\psi(s, a) \in \mathbb{R}^{d \times d}$ and $\phi_s(s^+) \in \mathbb{R}^d$. So,

864
865
866
867
868
869
870
871
872
873
874
875

$$\begin{aligned}
M^\pi(s, a, s^+) &= \sum_i \sum_j \phi_\psi(s, a)_{ij} \phi_s(s^+)_j w_i^\pi \\
\implies M^\pi(s, a, s^+) &= \sum_j \sum_i \phi_\psi(s, a)_{ij} w_i^\pi \phi_s(s^+)_j \\
\implies M^\pi(s, a, s^+) &= \sum_j \phi_\psi(s, a)_j^T w^\pi \phi_s(s^+)_j \\
\implies M^\pi(s, a, s^+) &= \sum_j \psi^\pi(s, a)_j \phi_s(s^+)_j \quad (\text{Writing } \phi_\psi(s, a)^T w^\pi \text{ as } \psi^\pi(s, a)) \\
\implies M^\pi(s, a, s^+) &= \psi^\pi(s, a)^T \phi_s(s^+)
\end{aligned}$$

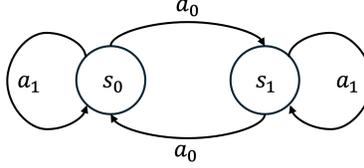
876 From Lemma 6.2, $\psi^\pi(s, a)$ is the successor feature for the basic feature $\phi_s(s)^T (\phi_s \phi_s^T)^{-1}$.

877
878 Note: In continuous settings, we can use the dataset marginal density as described in Section 5. The
879 basic features become $\phi_s(s)^T (\mathbb{E}_\rho[\phi_s \phi_s^T])^{-1}$. \square

880
881

A.6 DERIVING A BASIS FOR THE TOY EXAMPLE

882
883
884
885
886
887
888
889



890
891

Figure 5: The Toy MDP described in Section 4.

892
893

894 Consider the MDP shown in Figure 5. The state action visitation distribution is written as $d =$
895 $(d(s_0, a_0), d(s_1, a_0), d(s_0, a_1), d(s_1, a_1))^T$. The corresponding dynamics can be written as,

896
897
898

$$P = \begin{matrix} & \begin{matrix} s_0, a_0 & s_1, a_0 & s_0, a_1 & s_1, a_1 \end{matrix} \\ \begin{matrix} s_0 \\ s_1 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \end{matrix}$$

899
900

The Bellman Flow equation thus becomes,

901
902
903
904
905
906
907
908
909
910
911
912

$$\begin{aligned}
\sum_a d(s, a) &= (1 - \gamma)\mu(s) + \gamma \sum_{s', a'} P(s|s', a') d(s', a') \\
\implies \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{pmatrix} d(s_0, a_0) \\ d(s_1, a_0) \\ d(s_0, a_1) \\ d(s_1, a_1) \end{pmatrix} &= (1 - \gamma) \begin{pmatrix} \mu(s_0) \\ \mu(s_1) \end{pmatrix} + \gamma \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} d(s_0, a_0) \\ d(s_1, a_0) \\ d(s_0, a_1) \\ d(s_1, a_1) \end{pmatrix} \\
\implies \begin{bmatrix} 1 & 1 - \gamma & -\gamma & 0 \\ -\gamma & 0 & 1 & 1 - \gamma \end{bmatrix} \begin{pmatrix} d(s_0, a_0) \\ d(s_1, a_0) \\ d(s_0, a_1) \\ d(s_1, a_1) \end{pmatrix} &= (1 - \gamma) \begin{pmatrix} \mu(s_0) \\ \mu(s_1) \end{pmatrix}
\end{aligned}$$

913
914

This affine equation can be solved in closed form using Gauss Elimination to obtain

915
916
917

$$\begin{pmatrix} d(s_0, a_0) \\ d(s_1, a_0) \\ d(s_0, a_1) \\ d(s_1, a_1) \end{pmatrix} = w_1 \begin{pmatrix} \frac{-\gamma}{1+\gamma} \\ \frac{-1}{1+\gamma} \\ 1 \\ 0 \end{pmatrix} + w_2 \begin{pmatrix} \frac{-1}{1+\gamma} \\ \frac{-\gamma}{1+\gamma} \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} \frac{\mu(s_0) + \gamma\mu(s_1)}{1+\gamma} \\ \frac{\mu(s_1) + \gamma\mu(s_0)}{1+\gamma} \\ 0 \\ 0 \end{pmatrix}. \quad (16)$$

B EXPERIMENTAL DETAILS

B.1 ENVIRONMENTS

B.1.1 GRIDWORLDS

We use https://github.com/facebookresearch/controllable_agent code-base to build upon the gridworld and 4 room experiments. The task is to reach a goal state that is randomly sampled at the beginning of every episode. The reward function is 0 at all non-goal states while 1 at goal states. The episode length for these tasks are 200.

The state representation is given by (x, y) which are scaled down to be in $[0, 1]$. The action space consists of *five* actions: $\{up, right, down, left, stay\}$.

B.1.2 FETCH

We build on top of https://github.com/ahmed-touati/controllable_agent which contains the Fetch environments with discretized action spaces. The state space is unchanged but the action space is discretized to produce manhattan style movements i.e. move one-coordinate at a time. These six actions are mapped to the true actions of Fetch as: $\{0 : [1, 0, 0, 0], 1 : [0, 1, 0, 0], 2 : [0, 0, 1, 0], 3 : [-1, 0, 0, 0], 4 : [0, -1, 0, 0], 5 : [0, 0, -1, 0]\}$. Fetch has an episode length of 50.

B.1.3 DM-CONTROL ENVIRONMENTS

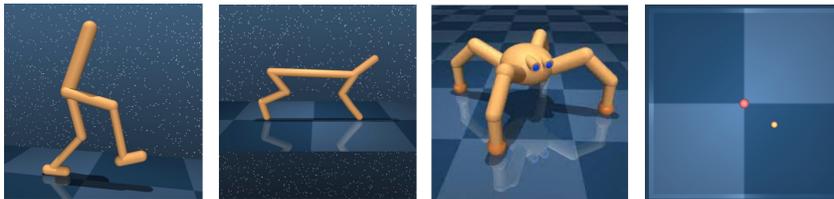


Figure 6: **DM Control Environments:** Visual rendering of each of the four DM Control environments we use: (from left to right) Walker, Cheetah, Quadruped, Pointmass

These continuous control environments have been discussed in length in DeepMind Control Suite (Tassa et al., 2018). We use these environments to provide evaluations for PSM on larger and continuous state and action spaces. The following four environments are used:

Walker: It has 24 dimensional state space consisting of joint positions and velocities and 6 dimensional action space where each dimension of action lies in $[-1, 1]$. The system represents a planar walker. At test time, we test the following four tasks: *Walk, Run, Stand and Flip*, each with complex dense rewards.

Cheetah: It has 17 dimensional state space consisting of joint positions and velocities and 6 dimensional action space where each dimension of action lies in $[-1, 1]$. The system represents a planar biped “cheetah”. At test time, we test the following four tasks: *Run, Run Backward, Walk and Walk Backward*, each with complex dense rewards.

Quadruped: It has 78 dimensional state space consisting of joint positions and velocities and 12 dimensional action space where each dimension of action lies in $[-1, 1]$. The system represents a 3-dimensional ant with 4 legs. At test time, we test the following four tasks: *Walk, Run, Stand and Jump*, each with complex dense rewards.

Pointmass: The environment represents a 4-room planar grid with 4-dimensional state space (x, y, v_x, v_y) and 2-dimensional action space. The four tasks that we test on are *Reach Top Left, Reach Top Right, Reach Bottom Left and Reach Bottom Right* each being goal reaching tasks for the four room centers respectively.

All DM Control tasks have an episode length of 1000.

972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

B.2 DATASETS

Gridworld: The exploratory data is collected by uniformly spawning the agent and taking a random action. Each of the three method is trained on the reward-free exploratory data. At test time, a random goal is sampled and the optimal Q function is inferred by each.

Fetch: The exploratory data is collected by running DQN (Mnih et al., 2013) training with RND reward (Burda et al., 2019) taken from <https://github.com/iDurugkar/adversarial-intrinsic-motivation>. 20000 trajectories, each of length 50, are collected.

DM Control: We use publically available datasets from ExoRL Suite [] collected using RND exploration.

B.3 IMPLEMENTATION DETAILS

B.3.1 BASELINES

We consider a variety of baselines that represent different state of the art approaches for zero-shot reinforcement learning. In particular, we consider Laplacian, Forward-Backward, and HILP.

1. **Laplacian (Wu et al., 2018; Koren, 2003):** This method constructs a graph Laplacian for the MDP induced by a random policy. Eigenfunctions of this graph Laplacian gives a representation for each state $\phi(s)$, or the state feature. These state-features are used to learn the successor features; and trained to optimize a family of reward functions $r(s) = \langle \phi(s) \cdot z \rangle$, where z is usually sampled from a unit hypersphere uniformly (same for all baselines). The reward functions are optimized via TD3.

2. **Forward-Backward (Blier et al., 2021a; Touati & Ollivier, 2021; Touati et al., 2023):** Forward-backward algorithm takes a slightly different perspective: instead of training a state-representation first, a mapping is defined between reward function to a latent variable ($z = \sum_s \phi(s).r(s)$) and the optimal policy for the reward function is set to π_z , i.e the policy conditioned on the corresponding latent variable z . Training for optimizing all reward functions in this class allows for state-features and successor-features to coemerge. The reward functions are optimized via TD3.

3. **HILP (Park et al., 2024a):** Instead of letting the state-features coemerge as in FB, HILP proposes to learn features from offline datasets that are sufficient for goal reaching. Thus, two states are close to each other if they are reachable in a few steps according to environmental dynamics. HILP uses a specialized offline RL algorithm with different discounting to learn these state features which could explain its benefit in some datasets where TD3 is not suitable for offline learning.

Implementation: We build upon the codebase for FB https://github.com/facebookresearch/controllable_agent and implement all the algorithms under a uniform setup for network architectures and same hyperparameters for shared modules across the algorithms. We keep the same method agnostic hyperparameters and use the author-suggested method-specific hyperparameters. The hyperparameters for all methods can be found here:

Proto Successor Measures (PSM): PSM differs from baselines in that we learn richer representations compared to Laplacian or HILP as we are not biased by behavior policy or only learn representations sufficient for goal reaching. Compared to FB, our representation learning phase is more stable as we learn representations by Bellman evaluation backups and do not need Bellman optimality backups. Thus, our approach is not susceptible to learning instabilities that arise from overestimation that is common in Deep RL and makes stabilizing FB hard. The hyperparameters are discussed in Appendix Table 2.

Table 2: Hyperparameters for baselines and PSM.

Hyperparameter	Value
Replay buffer size	5×10^6 (10×10^6 for maze)
Representation dimension	128
Batch size	1024
Discount factor γ	0.98 (0.99 for maze)
Optimizer	Adam
Learning rate	3×10^{-4}
Momentum coefficient for target networks	0.99
Stddev σ for policy smoothing	0.2
Truncation level for policy smoothing	0.3
Number of gradient steps	2×10^6
Batch size for task inference	10^4
Regularization weight for orthonormality loss (ensures diversity)	1
FB specific hyperparameters	
Hidden units (F)	1024
Number of layers (F)	3
Hidden units (b)	256
Number of layers (b)	2
HILP specific hyperparameters	
Hidden units (ϕ)	256
Number of layers (ϕ)	2
Hidden units (ψ)	1024
Number of layers (ψ)	3
Discount Factor for ϕ	0.96
Discount Factor for ψ	0.98 (0.99 for maze)
Loss type	Q-loss
PSM specific hyperparameters	
Hidden units (ϕ, b)	1024
Number of layers (ϕ, b)	3
Hidden units (w)	1024
Number of layers (w)	3
Double GD lr	$1e-4$

B.3.2 PSM REPRESENTATION LEARNING PSUEDOCODE

```

1061 def psm_loss(
1062     self,
1063     obs: torch.Tensor,
1064     action: torch.Tensor,
1065     discount: torch.Tensor,
1066     next_obs: torch.Tensor,
1067     next_goal: torch.Tensor,
1068     z: torch.Tensor,
1069     step: int
1070 ) -> tp.Dict[str, float]:
1071     metrics: tp.Dict[str, float] = {}
1072     # Create a batch_size x batch_size for learning  $M^{\pi}(s, a, s')$ 
1073     idx = torch.arange(obs.shape[0]).to(obs.device)
1074     mesh = torch.stack(torch.meshgrid(idx, idx, indexing='xy')).T.
1075     reshape(-1, 2)
1076     m_obs = obs[mesh[:, 0]]
1077     m_next_obs = next_obs[mesh[:, 0]]
1078     m_action = action[mesh[:, 0]]
1079     m_next_goal = next_goal[mesh[:, 1]]
1080     perm = torch.randperm(obs.shape[0])
1081
1082     # compute PSM loss
1083     with torch.no_grad():

```

```

1080 23         target_phi, target_b = self.psm_target(m_next_obs,
1081 m_next_goal)
1082 24         target_w = self.w_target(z)
1083 25         target_phi = target_phi[torch.arange(target_phi.shape[0]),
next_actions.squeeze(1)]
1084 26         target_b = target_b[torch.arange(target_b.shape[0]),
next_actions.squeeze(1)]
1085 27         target_M = torch.einsum("sd, sd -> s", target_phi, target_w)
1087 + target_b
1088 28
1089 29         phi, b = self.psm(m_obs, m_next_goal)
1090 30         phi = phi[torch.arange(phi.shape[0]), m_action.squeeze(1)]
1091 31         b = b[torch.arange(b.shape[0]), m_action.squeeze(1)]
1092 32         M = torch.einsum("sd, sd -> s", phi, self.w(z)) + b
1093 33         M = M.reshape(obs.shape[0], obs.shape[0])
1094 34         target_M = target_M.reshape(obs.shape[0], obs.shape[0])
1095 35         I = torch.eye(*M.size(), device=M.device)
1096 36         off_diag = ~I.bool()
1097 37         psm_offdiag: tp.Any = 0.5 * (M - discount * target_M)[off_diag].
1098 38         pow(2).mean()
1099 39         psm_diag: tp.Any = -((1 - discount) * (M.diag().unsqueeze(1))).
1100 40         mean()
1101 41         psm_loss = psm_offdiag + psm_diag
1102 42
1103 43         # optimize PSM
1104 44         self.opt.zero_grad(set_to_none=True)
1105 45         self.actor_opt.zero_grad(set_to_none=True)
1106 46         psm_loss.backward()
1107 47         self.opt.step()
1108 48         self.actor_opt.step()
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

```

Compute: All our experiments were trained on Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz CPUS and NVIDIA GeForce GTX TITAN GPUs. Each training run took around 10-12 hours.