BATS: Best Action Trajectory Stitching

Ian Char[1]*, **Viraj Mehta**[2]*, **Adam Villaflor**[2], **John M. Dolan**[2], and **Jeff Schneider**[1]

[1]Department of Machine Learning, Carnegie Mellon University
[2]Robotics Institute, Carnegie Mellon University
{ichar, virajm, avillaflor, jdolan, schneide}@cs.cmu.edu

Abstract

The problem of offline reinforcement learning focuses on learning a good policy from a log of environment interactions. Past efforts for developing algorithms in this area have revolved around introducing constraints to online reinforcement learning algorithms to ensure the actions of the learned policy are constrained to the logged data. In this work, we explore an alternative approach by planning on the fixed dataset directly. Specifically, we introduce an algorithm which forms a tabular Markov Decision Process (MDP) over the logged data by adding new transitions to the dataset. We do this by using learned dynamics models to plan short trajectories between states. Since exact value iteration can be performed on this constructed MDP, it becomes easy to identify which trajectories are advantageous to add to the MDP. Crucially, since most transitions in this MDP come from the logged data, trajectories from the MDP can be rolled out for long periods with confidence. We prove that this property allows one to make upper and lower bounds on the value function up to appropriate distance metrics. Finally, we demonstrate empirically how algorithms that uniformly constrain the learned policy to the entire dataset can result in unwanted behavior, and we show an example in which simply behavior cloning the optimal policy of the MDP created by our algorithm avoids this problem.

1 Introduction

The goal of Reinforcement Learning (RL) is to learn a policy which makes optimal actions for a decision making problem or control task. The field of *deep* RL, in which one learns neural network models to represent key quantities for decision making, has recently made great strides [21, 17, 12]. In many deep RL algorithms, this involves learning a neural network for both the policy and the value function, which estimates the value of states or state-action-pairs with respect to the current policy. Many promising model-based methods [4, 10] also learn a deep dynamics function that estimates next states given current states and actions.

In the standard, *online* setting, the policy is repeatedly deployed in the environment during training time, which provides a continual stream of on-policy data that stabilizes the learning procedure. However, the *online* setting is unreasonable for applications, since it requires a way to cheaply and safely gather a large number of on-policy samples. As such, there has been increasing interest in the so-called *offline* setting [15] in which a policy is learned solely from logged off-policy data.

However, the offline setting comes with its own problems. Simply applying deep reinforcement learning algorithms designed for the online setting will often cause exploding value estimates because of distribution mismatch and recursive updates [13]. In model-based methods, the combination of small initial errors and test-time distribution shift often leads to rapidly accumulating model error.

---

*Equal contribution author.

While distribution shift and model exploitation are potential issues in online RL, these problems are more severe in the offline setting, as the agent cannot collect additional experience to rectify compounding errors in estimation or planning. To address these problems, offline RL algorithms add constraints to encourage the agent to only operate in the support of the data by either constraining the policy [25, 13] or penalizing uncertain state-actions [26, 11, 14, 27].

Rather than trying to implicitly constrain the agent to stay in the support of the data, in this work we explore what happens if we plan over the logged data directly. In particular, we create a tabular MDP by planning short trajectories between states in the dataset, and then we do exact value iteration on this MDP. Unlike other model-based methods which are limited to short imagined trajectories, trajectories from our MDP are mostly comprised of real transitions from the dataset and can therefore be rolled out for much longer with confidence. As such, we argue that our algorithm is able to better reason about the dataset as a whole. In this work, we show that re-imagining the offline dataset in this way allows for the following:

- By coupling together long trajectories with exact value iteration, our algorithm is able to better estimate the resulting policy's value. We prove that under the correct distance metrics our algorithm can be used to form upper and lower bounds for the value function. We demonstrate empirically that this aligns well with the value of a policy behavior cloned on these trajectories.

- By performing full rollouts in our tabular MDP, we are able to approximate our optimal policy's occupancy distribution. We show how many algorithms that uniformly constrain the learned policy to actions on the dataset struggle with "undirected" datasets (i.e., data collected without a specific reward function in mind) and demonstrate that our algorithm avoids this problem by filtering out data unrelated to the task.

2 Preliminaries

In this work, we assume the environment can be represented as a *deterministic*, infinite horizon MDP $M = \langle \mathcal{S}, \mathcal{A}, \gamma, T, r, \rho \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\gamma \in (0, 1)$ is the discount factor, $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and ρ is the initial state distribution. We refer to an MDP as *tabular* if it has a finite state and action space. While we assume that the true environment in question has every action available to be played at every state (i.e. $T(s, a)$ is well-defined $\forall s \in \mathcal{S}, a \in \mathcal{A}$), later in this work we also consider MDPs that have only a subset of actions for each state. When this is the case, we denote $\mathcal{A}^s \subset \mathcal{A}$ to be the actions available to be played at state $s \in \mathcal{S}$. Such MDPs are defined as $M = \langle \mathcal{S}, \{\mathcal{A}^s\}_{s \in \mathcal{S}}, \gamma, T, r, \rho \rangle$.

In reinforcement learning, we attempt to learn a stochastic policy $\pi(a|s) : \mathcal{S} \to P(\mathcal{A})$, where $P(\mathcal{A})$ is the set of all distributions over $\mathcal{A}$. We desire our learned policy to maximize the expected discounted sum of rewards, $\mathbb{E}_{\pi, \rho} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right]$, where $s_t = T(s_{t-1}, a_{t-1})$, $a_t \sim \pi(\cdot|s_t)$, and $s_0 \sim \rho$. To facilitate the optimization of this quantity, we can define an optimal state-action value function $Q^* : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that satisfies the following recurrence relation known as the Bellman Equation:

$$Q^*(s, a) = r(s, a) + \gamma \max_{a'} Q^*(T(s, a), a') \tag{1}$$

Then, we can reformulate policy optimization as trying to solve $\pi(s) = \operatorname*{argmax}_{a} Q^*(s, a), \forall s \in \mathcal{S}$, where we can estimate Q^* by iteratively performing the bellman update to $Q_{k+1}(s, a) \leftarrow r(s, a) + \gamma \max_{a'} Q_k(T(s, a), a')$. In tabular MDPs, the Q-function and its updates can be written and performed exactly. Thus, this procedure – known as *value iteration* – will eventually converge, i.e. $Q_k(s, a) \to Q^*(s, a), \forall (s, a) \in \mathcal{S} \times \mathcal{A}$ as $k \to \infty$. However, in general MDPs where there is a possibly infinite number of states or actions, we must rely on function approximation and finite samples to instead perform *approximate value iteration*, which is not guaranteed to converge. For notational convenience, we denote the value of a state as $V^*(s) = \max_a Q^*(s, a)$. Policy π's *occupancy distribution* is defined as $\mu_\pi(s) \propto \sum_{i=0}^{\infty} \gamma^i p(s_i = s)$, where $s_i = T(s_{i-1}, a_{i-1})$, $a_{i-1} \sim \pi(\cdot|s_{i-1})$, and $p(s_0) \equiv \rho$. We denote the value function for π as V^π; that is, $V^\pi(s)$ is the expected, cumulative discounted sum of rewards from playing policy π starting from state s. When it is not clear from context, we denote V_M^π as the value for the function π, specifically over MDP M.

In offline reinforcement learning, one assumes access to a fixed set of environment interactions. In this work, we assume that we have access to a dataset, $D = \bigcup_{j \in [N]} \{(s_{ji}, a_{ji}, s'_{ji}, r_{ji})\}_{i=1}^{t_j}$, which is comprised of N trajectories of possibly varying lengths, t_j. For the remainder of Section 2, we use $s_{ji}, s'_{ji}, a_{ji}, r_{ji}$ to represent the current state, next state, action played, and reward received for the ith timestep of the jth trajectory. Also note that, if $i < t_j$, then $s_{j(i+1)} = s'_{ji}$.

Structures over the Offline Data. Given a dataset, D, collected in MDP $M = \langle \mathcal{S}, \mathcal{A}, \gamma, T, r, \rho \rangle$, one can construct a tabular MDP that incorporates only the states and actions observed in the dataset. We denote this MDP as $M_0 = \langle \mathcal{S}_0, \{\mathcal{A}_0^s\}_{s \in \mathcal{S}_0}, \gamma, T_0, r_0, \rho_0 \rangle$, where $\mathcal{S} = \cup_{j \in [N], i \in [t_j]} (\{s_{ji}\} \cup \{s'_{ji}\})$, $\mathcal{A}_0^s = \{a_{ji} | \forall j \in [N], \forall i \in [t_j] \text{ s.t. } s_{ji} = s\}$, $T_0(s, a) = T(s, a)$, $r_0(s, a) = r(s, a)$, and ρ_0 is a discrete uniform distribution over $\{s_{j0}\}_{j=1}^M$.

It will often be beneficial to describe the offline dataset from a graphical perspective. A graph, $G := (V, E)$, is fully characterized by its vertex set, V, and its edge set, E. We note that the notation for the vertex set is overloaded with the value function, but the difference is often clear from context. For any MDP, we can define a corresponding graph that has a vertex set which is the same as the MDP's state space and an edge set which matches the MDP's transition function. For example, the graph corresponding to M_0, $G_0 = (V_0, E_0)$, has vertex set, $V_0 = \mathcal{S}_0$, and edge set, $E_0 = \{(s, T_0(s, a)) | s \in \mathcal{S}_0, a \in \mathcal{A}_0^s\}$. Specific to this paper, we also consider the undirected, *neighbor graph*, G_ϵ, which has the same vertex set, but has edge set such that $\{s, s'\}$ is an edge iff $\|s - s'\| \leq \epsilon$ for a specified norm and $\epsilon > 0$.

Bisimulation Metric. In this work, we use the *on-policy bisimulation distance* from Castro [2], which also gives a sampling-based algorithm for approximating such a metric. We denote this as $d_\sim^\pi(\cdot, \cdot)$. A key result about this metric is the following:

Theorem 1 (Theorem 3 from Castro [2]). *Given states $s, t \in \mathcal{S}$ in an MDP, M, and a policy, π,*

$$|V^\pi(s) - V^\pi(t)| \leq d_\sim^\pi(s, t).$$

In other words, $d_\sim^\pi(\cdot, \cdot)$ is a metric over states for which the value function is 1-Lipschitz continuous. We discuss bisimulation further in Appendix B.

3 Method

The MDP, M_0, as described in Section 2, has several desirable properties. First, it is tabular, so one can easily apply value iteration to find the optimal policy. Second, policies defined over M_0 will be conservative since policies can only choose actions that appear in the dataset. Unfortunately, M_0 is so conservative that it is uninteresting since there is little to no choice in what actions can be made at each state. We must make additions in order to create a more interesting MDP to optimize. Our solution is to create transitions via planning: an operation we call *stitching*.

3.1 The Stitching Operation

Broadly speaking, the so-called stitching operation simply adds a transition (or sequence of transitions) from one pre-existing state in an MDP to another pre-existing state via planning in a learned model. To flesh this operation out, suppose that, for a tabular MDP, $\hat{M} = \langle \hat{S}, \{\hat{A}^s\}_{s \in \hat{S}}, \gamma, \hat{T}, \hat{r}, \rho_0 \rangle$, we would like to add the ability to transition from $s \in \hat{S}$ to $s' \in \hat{S}$. Here, $\hat{M}$ is either M_0, or M_0 with some additional states, actions, and transitions included. Using a learned dynamics model, $\tilde{T}$, as a proxy to the the true dynamics, T, we can find actions that transition from s to s' via *planning*, i.e. we can solve the following optimization problem:

$$\underset{a_0, \ldots, a_{k-1} \in \mathcal{A}}{\operatorname{argmin}} \|s' - s_k\| \quad \text{where } s_j = \tilde{T}(s_{j-1}, a_{j-1}), \forall j = 1, \ldots, k \quad \text{and } s_0 = s \quad (2)$$

where k is the number of actions allowed and $\mathcal{A}$ is the set of actions available in the environment. We choose to optimize this objective with the Cross Entropy Method (CEM), as used in Chua et al. [4]. For a specified tolerance, $\delta \in \mathbb{R}$, we consider it possible to transition from s to s' if there exists a solution such that $\|s' - s_k\| < \delta$. If the tolerance cannot be achieved, we leave $\hat{M}$ unchanged. Otherwise, we set $\hat{S} = \hat{S} \cup \{s_i\}, \hat{A}^{s_i} = \hat{A}^{s_i} \cup \{a_i\}$ for $i = 0, \ldots, k-1$, where $\hat{A}^{s_i} = \emptyset$ if
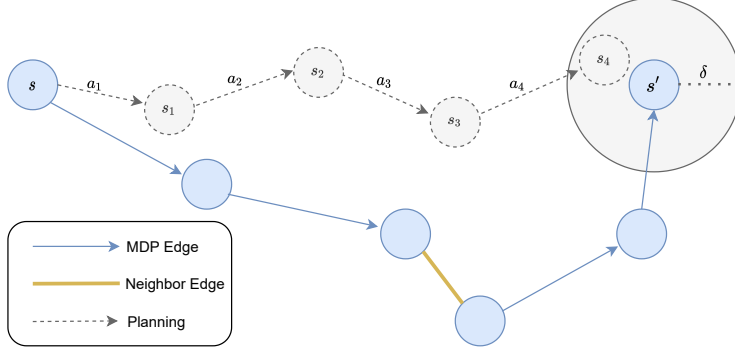
Figure 1: **A diagram of the stitching procedure in BATS.** The blue edges come from the directed graph, G_i, the yellow edge comes from the neighbor graph, G_ϵ, and dashed gray edges are the planned actions. Here, the (s, s') state pair is a viable candidate to try to stitch with $k = 4$ actions because there are 4 blue edges and one yellow edge forming a path from s to s'. This would be considered a successful stitch since s_4 is within δ of s'.

$s_{i-1} \notin \hat{\mathcal{S}}$. If $i < k-1$, we set , $\hat{T}(s_i, a_i) = s_{i+1}$, and otherwise set $\hat{T}(s_{k-1}, a_{k-1}) = s'$. Lastly, $\hat{r}(s_i, a_i) = \tilde{r}(s_i, a_i) - cd(s_k, s')$ for $i = 0, \ldots, k-1$, where $\tilde{r}$ is a learned estimate of the reward function, c is a penalty coefficient, and d is an appropriate distance metric. The addition of the penalty term encourages policies to choose transitions that occur in the dataset over the possibly erroneous ones that are added via stitching. Choosing d to be a bisimulation distance has theoretical ramifications which we will discuss in Section 4.

3.2 The BATS Algorithm

Given unlimited compute, the ideal algorithm would be to attempt to stitch all pairs of states in the graph and then perform value iteration on the resulting tabular MDP. However this is not often feasible, which is where our algorithm, *Best Action Trajectory Stitching (BATS)*, comes into play. BATS is an iterative algorithm where, for each $i = 0, 1, \ldots, n-1$, we perform value iteration for MDP, M_i, to find optimal policy π_i, we narrow the pool of candidate stitches to those that are both feasible and impactful, and lastly we run the stitching procedure over M_i and set the results as M_{i+1}. We will now discuss the heuristics for identifying feasible and impactful stitches. The full algorithm is written out concretely in Appendix A.

Identifying Feasible Stitches. We first establish a notion of *feasible* stitches. We operate on the following heuristic: if there exists a sequence of actions that lead to s' from s, there is likely a similar sequence of actions that lead to s' starting at a state neighboring s. Concretely, for iteration i, we only consider stitching a state, s, to another state, s', if there exists a path from s to s' that uses at most K edges from graph G_i (i.e. the graph corresponding to M_i) and exactly one edge from the nearest neighbor graph G_ϵ (this is visualized in Figure 1). If we find that k edges from G_i are used in the path from s to s', we limit the planning procedure in the stitching operation to optimize for k actions. To introduce more conservatism, we also only consider $s' \in \mathcal{S}_0$; that is, we do not consider stitching to any "imagined" states that may be the result of previous stitching. This constraint enforces the agent to stay in distribution.

Identifying Impactful Stitches. To help identify impactful stitches during iteration i, we focus on making stitches that maximizes $\mathbb{E}_{s \sim \mu_{\pi_{i+1}}} \left[V_{M_{i+1}}^{\pi_{i+1}}(s) \right]$. To do this heuristically, we first sample $s_1, \ldots, s_m$ from μ_{π_i} and find all feasible destinations each sample could stitch to. Let s be one such sample, and let s' be a feasible destination state. Suppose there is a path connecting these states that uses exactly k edges from G_i and one edge from G_ϵ. Let s'_k be the state that π_i transitions to after acting k times in M_i. Then, we consider (s, s') to be a candidate stitch if $V^{\pi_i}(s') > V^{\pi_i}(s'_k)$. In other words, if it is believed that s' can be reached from s in k transitions, then the stitch between s and s' only deserves our attention if s' is more valuable than the state that π_i currently transitions to in k steps.

After running the BATS algorithm for n iterations, we are left with an optimal policy, π_n for the stitched, tabular MDP, M_n; however, this policy cannot be deployed on the true, continuous MDP, M, since the domain of π_n is only a subset of $\mathcal{S}$. To remedy this, we collect a large number of trajectories

4

using π_n in M_n to generate a dataset of state-action tuples to train a parametric policy with behavioral cloning. However, we note that alternative policy learning algorithms could be used to make a policy well-defined over $\mathcal{S}$.

Hyperparameters. Our algorithm has a number of hyperparameters of interest. Dynamics model training, metric learning, behavior cloning, and CEM all have parameters which trade off computation and performance. However, these are well-known methods which operate independently and for which hyperparameter selection is a relatively well-understood problem. The BATS algorithm itself requires a tolerance for value iteration, ϵ for the neighbors graph, δ for planning tolerance, m for the number of samples from occupancy distribution per iteration, K for the max number of actions in a stitch, and n for the number of iterations. We further discuss how we determined hyperparameters for our experiments in the Appendix D.1.

4   Analyzing BATS with Bisimulation Distance

Assumptions   Let M_0 be a tabular MDP formed from an offline dataset collected in M, as previously formulated. We can extend π to the domain $\mathcal{S}$ by behavior cloning; that is, by finding the member of a parameterized policy class which has minimum training error regularized with weight norm. For a slight simplification of our analysis, we assume that the hypothesis class is rich enough that it can interpolate the training set; that is, it can reach zero behavior cloning error for all $s \in \hat{\mathcal{S}}$. We often refer to both a policy and its extension as π.

We also assume that on a finite MDP an optimal policy can be found efficiently, and that the learned dynamics model, $\tilde{T}$, is accurate for short transitions in the support of the data. Although in practice, we will learn the reward function, in this analysis we also assume the reward function, r, is known. Lastly, we assume that we are able to learn an embedding, $\phi^\pi : \mathcal{S} \to Z$, such that the L2 norm in the latent space, Z, is the on-policy bisimulation metric. That is, we can learn a ϕ^π such that if $||\phi^\pi(s) - \phi^\pi(s')|| < \epsilon$ then $|V^\pi(s) - V^\pi(s')| < \epsilon$.

Sandwich Bound on Value   Consider the collection of ℓ tuples $\{(b_j, c_j, a_j)\}_{j=1}^\ell$, where $b_j, c_j \in \mathcal{S}_0$, $a_j \in \mathcal{A}$, and $a_j \notin \mathcal{A}_0^{b_j}$ for all $j \in [\ell]$. Then define M^- as the MDP derived by starting from M_0 and, for each $j \in [\ell]$, setting $\mathcal{A}_0^{b_j} = \{a_j\} \cup \mathcal{A}_0^{b_j}$, $T_0(b_j, a_j) = c_j$, and $r_0(b_j, a_j) = r(b_j, a_j) - \gamma\epsilon_j$, where $\epsilon_j > 0$ is some notion of penalty. In other words, M^- is the result of making ℓ stitches in M_0 where $K = 1$. There exists a policy π^- which is the optimal policy on M^- and extends by behavior cloning to $\mathcal{S}$. Similarly, we can construct MDP M^+ in the exact same way as M^-, but by setting reward to $\hat{r}(b_j, a_j) = r(b_j, a_j) + \gamma\epsilon_j$ for each $j \in [\ell]$. In this setting we can bound the value of π^- in the true MDP on either side by its value attained in the finite MDPs just defined. We formalize this notion in the following Theorem.

Theorem 2. *For $j \in [\ell]$, let each penalty term, ϵ_j, be such that $||\phi^{\pi^-}(T(b_i, a_i)) - \phi^{\pi^-}(c_i)|| < \epsilon_j$.*
Then

$$\forall s \in \mathcal{S}_0, V_{M^-}^{\pi^-}(s) \leq V_M^{\pi^-}(s) \leq V_{M^+}^{\pi^-}(s).$$

In other words, under the correct assumptions, we can construct a pessimistic and optimistic MDP. The value of policy π^- in the pessimistic and optimistic MDP will bound the true value of this policy from below and above, respectively. We give the proof in Appendix C and a short sketch here: The value function can be written as $V_M^{\pi^-}(s) = \sum_{s_i} \gamma^i r(s_i, \pi(s_i))$, which can be lower and upper-bounded using Theorem 1 for every transition in the expansion which does not exist in the dataset. This is accomplished by taking into account a pessimistic planning error of the dynamics model. Although Theorem 2 is for the case where we limit stitches to have at most one action, it is likely easy to extend this result to more actions.

There are 3 major implications of the theorem. First, if the behavior cloning, dynamics model, and bisimulation assumptions hold, the value function estimates must be accurate. Second, reasoning by contraposition gives that if the value function estimates are bad, it is due to errors in these components. As such, one should recompute the edge penalties as the policy changes by fine-tuning the bisimulation metric. Third, if the current lower bound is higher than a previous upper bound on the value, the policy is guaranteed to have improved.
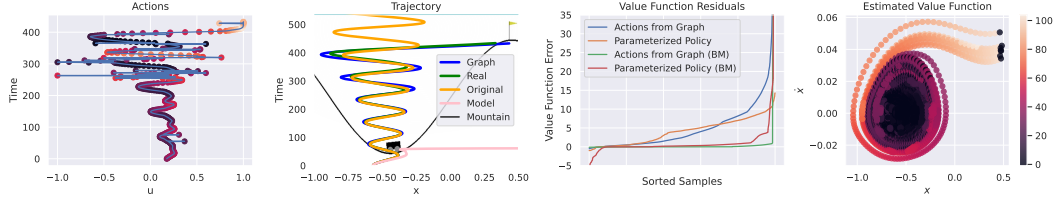
5

Figure 2: **Mountain Car Example**. The plots show example actions, associated trajectories, the value function lower bound, and residuals for the standard dynamics model and the bisimulation metric.

We formalize the third fact in the following corollary. Starting with the setup from before, let M'^- and M'^+ be tabular MDPs constructed using the alternative sequence of tuples $\{(b'_j, c'_j, a'_j)\}_{j=1}^{\ell'}$. Let ϵ'_j be the penalty term used in formulating these MDPs, and let π'^- be the optimal policy for M'^-.

Corollary 1. *Let ϵ_j and ϵ'_j satisfy the assumptions of Theorem 2 for mappings ϕ^{π^-} and $\phi^{\pi'^-}$, respectively. If for some $s \in \mathcal{S}_0$, $V_{M'^+}^{\pi^-}(s) < V_{M'^-}^{\pi'^-}(s)$, then $V_M^{\pi^-}(s) < V_M^{\pi'^-}(s)$.*

This corollary is a natural consequence of Theorem 2 and implies the policy $\hat{\pi}^-$ is better at state s than π^-. If this holds on average for $s \sim \rho$, then we can conclude that $\hat{\pi}^-$ is the better policy.

5 Illustrative Example: Mountain Car

As an initial test of the algorithm and analysis, we trained BATS on a small dataset consisting mostly of random actions along with 5 expert trajectories on a continuous mountain car environment. Behavior cloning fails on this dataset, but running BATS and subsequently behavior cloning trajectories from the stitched MDP solves the problem, reliably getting returns of 95 (90 is considered solved). In Figure 2, we show how this happens. Starting at the left, the actions are in general concomitant with a good mountain car policy but with occasional spikes. The spikes result from the planning step in BATS, which has the objective solely to reach the next intended state as accurately as possible. Although the large control inputs are costly in this problem, they are only intermittently added and in general result in a policy that solves the problem.

The second panel makes clear how BATS leverages real transitions to produce realistic, trustworthy trajectories. To show this, we show a stitched trajectory (blue) that was originally (gold) unable to reach the goal. Replaying the actions from this trajectory in the real environment, we find that the trajectory closely matches what happens in the environment (green). However, replaying the same actions in our learned dynamics model results in horrendous error before 100 timesteps (pink). This demonstrates how our method can produce more effective novel long-term trajectories than purely relying on rolling out a learned model.

The third panel shows the error in the graph value function estimates over a sampling of states from the graph from the true returns experienced both by executing the actions associated with the edges taken in the graph and by executing a policy cloned from the graph. We also train a bisimulation model following work from Zhang et al. [28], Castro [2] and execute BATS according to this metric (see Appendix D.2). We find that doing this results in the value function estimates that are quite accurate. One interesting feature is that on the left side, there are actually states where the cloned policy does *better* than the graph policy. We believe this is likely due to the cloned policy smoothing out some of the control spikes output by the planning process and using less control input. This panel admits a natural decomposition of the errors in BATS. The small errors in executing the bisimulation graph policy (green) show that the dynamics model training and bisimulation metric is likely working here, while the additional errors induced by the corresponding cloned policy (red) show that here, the behavior cloning process induces a slight additional error. We also note that the value function errors are much smaller when the bisimulation metric is used (red / green) than when the Euclidean metric is used (blue / orange), providing empirical evidence for its theoretical benefits. Finally, on the right we see a very sensible looking value function plot, where values are higher as the policy winds its way out of the valley.

6 Related Work

Offline Reinforcement Learning   In the past two years, there has been substantial interest in offline reinforcement learning and many algorithms have been developed to address the problem. In order to mitigate the issues of distribution shift and model exploitation, recent prior work in offline RL have explored incorporating many different types of conservatism in the optimization process. These approaches can be broadly grouped based on the type of conservatism they incorporate.

The first set of approaches use actor-critic methods [25, 13, 24, 18], but incorporate policy constraints to limit the difference between the learned policy and the behavioral policy that collected the data in order to mitigate distribution shift during the Bellman updates and at test time. The second set of approaches use model-based RL [26, 11], but leverage uncertainty-aware dynamics models to perform Model-Based Policy Optimization (MBPO) [10] while deterring the policy from taking action with high model uncertainty. The third set of approaches add conservatism directly to the Q-function [14, 27] in order to optimize a lower-bound on the true value function and avoid extrapolating optimistically. Finally, an alternate approach attempts to filter out the best available transitions in the data for behavior cloning by learning an upper bound on the value function [3].

Graphical Methods in Reinforcement Learning   There have been recent prior works which leverage finite MDPs and their graphical representations in order to estimate value functions. The simplest take the highest-returning actions from a particular state [1] or leverage differentiable memory to give a weighted combination of recent experiences [19]. Marklund et al. [16] gives a method of Exact then Approximate Dynamic Programming. The method quantizes the state space into variable-sized bins and treats the quantized MDP as finite. They solve this MDP via Value Iteration and then use these values to warm-start DDQN [23]. This method is close to ours, but assumes discrete action space, quantizes the state space, and does not leverage a dynamics model.

Another method, DeepAveragers [20], constructs a finite MDP from the dataset and extends the optimal value function via kNN. Their theoretical analysis relies on assumptions on the Lipschitzness of Bellman backups that directly affects the value of a cost hyperparameter, while we use the properties of the bisimulation metric to guarantee our bounds. It also only works on problems with discrete actions and doesn't add to the dataset in any way.

Other methods, like Zhu et al. [29], Hu et al. [9], use episodic memory techniques to organize the replay buffer, using averaging techniques and implicit planning to generalize from the replay buffer. However, they cannot plan novel actions to explicitly connect the dataset and are not designed with the offline setting in mind.
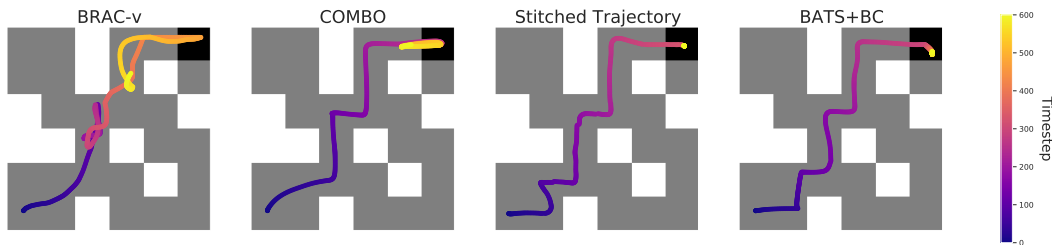
7 Experiments

Figure 3: **Trajectories in maze2d-medium.** The two leftmost plots show the top baselines on this task. While BRAC-v is able to make it to the goal, it is apparent that the constraints imposed make it sub-optimal as it makes several loops throughout the trajectory. While COMBO is able to efficiently navigate to the goal, it too is unable to remain in the goal region.

In this section we explore BATS experimentally. By planning in the approximate MDP, BATS can identify high-value sequences of actions. Unlike other state-of-the-art offline RL algorithms, BATS can also reason about which regions of the state space are important for the task at hand. We find that this is crucial for "undirected" datasets (i.e., datasets in which the data were collected without a specific reward function in mind).

Task	BATS+BC (Ours)	BC	SAC	BRAC-v	BEAR	CQL	COMBO
umaze	**141.8 ± 4.4**	3.8	88.2	-16.0	3.4	5.7	76.2 ± 0.5
medium	**133.6 ± 11.6**	30.3	26.1	33.8	29.0	5.0	74.8 ± 35.2
large	107.7 ± 22.0	5.0	-1.9	40.6	4.6	12.5	**150.3 ± 22.0**

Table 1: **D4RL Maze 2D.** The above shows undiscounted returns for each algorithm with the higest average bolded. For algorithms that we ran, we include the standard error. Results are averaged over three seeds, except for BATS+BC which was averaged over three seeds on each of the three graphs.

In the following experiments we first use BATS to find a good policy in the approximate, finite MDP, and then learn a policy for the true, continuous MDP by behavior cloning on optimal trajectories from the approximate MDP. We also assume that we have access to the start state distribution, and we use this to go through the data and label additional states as start states, since many datasets have few starts. Once BATS is complete, data is collected by unrolling trajectories with the optimal policy. Because we find that not all logged trajectories were stitched to good regions of state space, we filter out any trajectory that does not meet a required value threshold. Then, a neural network policy that outputs the mean and variance of a normal distribution is learned on the collected data.

To implement the algorithm, we rely on the *graph-tool* library. We used compute provided by XSEDE [22] to run BATS three times for each task. In order to save on compute, we execute the BATS algorithm with no penalty and with relaxed hyperparameters. After, we perform simple grid search of hyperparameters by relabelling and deleting stitched transitions and re-running value iteration on the resulting MDP. For the penalty term, we use $L2$ distance scaled by a constant hyperparameter. For more details on this procedure see Appendix D.1.

For baselines, we compare to methods reported in D4RL: off-policy SAC [8], CQL [14], BRAC [25], BEAR [13]. Since D4RL does not report any model-based methods, we compare against the COMBO algorithm, which alters the CQL algorithm to include short model rollouts. We used our own implementation of COMBO, which we find gets comparable results to the results reported (see Appendix D.3 for details). Final results can be seen in Table 1.

2D Maze Environments. We evaluate our procedure on D4RL's *maze2d* environments. While the baseline algorithms we compare against produce amazing results on the Mujoco control tasks, they struggle to match the performance of an expert policy on these relatively simple maze environments. We assert that a key reason for this is that the dataset is undirected. For each of the maze tasks, the dataset was collected by a PD controller that randomly wandered through the mazes. This presents a problem for algorithms which try to constrain the learned policy to be close to the behavioral policy, since there may be many instances where the behavioral policy performs an action that is in direct opposition to a specified goal. We see this concretely in the maze case, where most of the baseline policies are able to find their way to the goal, but start moving away from the goal once it is reached (see Figure 3). This happens because the policies are trained on a significant amount of data where the behavioral policy leaves the goal, but there are few to no data where the behavioral policy remains stationary in the goal. Even COMBO, which leverages a learned dynamics model, is unable to stay in the goal cell for the umaze and medium maze tasks.

BATS addresses this problem directly by filtering out any data that are unimportant for the task at hand. Training solely on the data seen by the optimal agent in the approximate MDP ensures that the policy for the continuous MDP is never trained on any disastrous actions or any data that are too far out of the policy's state distribution. At the same time, BATS can use the learned dynamics model to reason about how to stay stationary within the goal cell, as shown by Figure 3.

8 Conclusion

In this work, we presented an algorithm which eschews learning a deep value function by instead maintaining a finite MDP of transitions from the dataset and augmenting it with transitions planned using a learned model. The algorithm's performance is promising on low-dimensional tasks with theoretical support for its value function estimates. While stitching is hard on higher-dimensional tasks, we believe this work can be improved by incorporating low dimensional representation learning. Ideally this representation should be related to the bisimulation metric. Although we achieved promising results using the model architecture as described in Zhang et al. [28] on mountain car, we were unable to leverage the same model in our other experiments. We hope that new developments in learning bisimulation metrics will unlock additional potential in BATS. We also hope to formalize the

equivalent algorithm for when transitions are stochastic, and we hope that this extension will help the algorithm generalize to more complex problems.

Acknowledgments and Disclosure of Funding

References

[1] C. Blundell, B. Uria, A. Pritzel, Y. Li, A. Ruderman, J. Z. Leibo, J. Rae, D. Wierstra, and D. Hassabis. Model-free episodic control, 2016.

[2] P. S. Castro. Scalable methods for computing state similarity in deterministic markov decision processes, 2019.

[3] X. Chen, Z. Zhou, Z. Wang, C. Wang, Y. Wu, and K. Ross. Bail: Best-action imitation learning for batch deep reinforcement learning, 2020.

[4] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *CoRR*, abs/1805.12114, 2018. URL http://arxiv.org/abs/1805.12114.

[5] N. Ferns, P. Panangaden, and D. Precup. Bisimulation metrics for continuous markov decision processes. *SIAM J. Comput.*, 40(6):1662–1714, Dec. 2011. ISSN 0097-5397. doi: 10.1137/10080484X. URL https://doi.org/10.1137/10080484X.

[6] N. Ferns, P. Panangaden, and D. Precup. Metrics for finite markov decision processes. *CoRR*, abs/1207.4114, 2012. URL http://arxiv.org/abs/1207.4114.

[7] R. Givan, T. Dean, and M. Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1):163–223, 2003. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(02)00376-4. URL https://www.sciencedirect.com/science/article/pii/S0004370202003764.

[8] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018. URL http://arxiv.org/abs/1801.01290.

[9] H. Hu, J. Ye, Z. Ren, G. Zhu, and C. Zhang. Generalizable episodic memory for deep reinforcement learning. *CoRR*, abs/2103.06469, 2021. URL https://arxiv.org/abs/2103.06469.

[10] M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization, 2019.

[11] R. Kidambi, A. Rajeswaran, P. Netrapalli, and T. Joachims. Morel: Model-based offline reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 21810–21823. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/f7efa4f864ae9b88d43527f4b14f750f-Paper.pdf.

[12] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274, Sept. 2013. ISSN 0278-3649. doi: 10.1177/0278364913495721. URL https://doi.org/10.1177/0278364913495721.

[13] A. Kumar, J. Fu, G. Tucker, and S. Levine. Stabilizing off-policy q-learning via bootstrapping error reduction, 2019.

[14] A. Kumar, A. Zhou, G. Tucker, and S. Levine. Conservative q-learning for offline reinforcement learning, 2020.

[15] S. Levine, A. Kumar, G. Tucker, and J. Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020.

[16] H. Marklund, S. Nair, and C. Finn. Exact (then approximate) dynamic programming for deep reinforcement learning. *Bian and Invariances Workshop, ICML*, 2020.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. 2013. URL http://arxiv.org/abs/1312.5602. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[18] A. Nair, M. Dalal, A. Gupta, and S. Levine. Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359*, 2020.

[19] A. Pritzel, B. Uria, S. Srinivasan, A. P. Badia, O. Vinyals, D. Hassabis, D. Wierstra, and C. Blundell. Neural episodic control. *CoRR*, abs/1703.01988, 2017. URL http://arxiv.org/abs/1703.01988.

[20] A. K. Shrestha, S. Lee, P. Tadepalli, and A. Fern. Deepaveragers: Offline reinforcement learning by solving derived non-parametric {mdp}s. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=eMP1j9efXtX.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[22] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, et al. Xsede: accelerating scientific discovery. *Computing in science & engineering*, 16(5): 62–74, 2014.

[23] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[24] Z. Wang, A. Novikov, K. Zolna, J. T. Springenberg, S. E. Reed, B. Shahriari, N. Y. Siegel, J. Merel, Ç. Gülçehre, N. Heess, and N. de Freitas. Critic regularized regression. *CoRR*, abs/2006.15134, 2020. URL https://arxiv.org/abs/2006.15134.

[25] Y. Wu, G. Tucker, and O. Nachum. Behavior regularized offline reinforcement learning, 2019.

[26] T. Yu, G. Thomas, L. Yu, S. Ermon, J. Zou, S. Levine, C. Finn, and T. Ma. Mopo: Model-based offline policy optimization, 2020.

[27] T. Yu, A. Kumar, R. Rafailov, A. Rajeswaran, S. Levine, and C. Finn. Combo: Conservative offline model-based policy optimization, 2021.

[28] A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction. *CoRR*, abs/2006.10742, 2020. URL https://arxiv.org/abs/2006.10742.

[29] G. Zhu, Z. Lin, G. Yang, and C. Zhang. Episodic reinforcement learning with associative memory. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HkxjqxBYDB.

A The BATS Algorithm

Below we state the steps of the BATS algorithm. Algorithm 1 shows the main loop, while Algorithm 2 shows the subroutines.

Algorithm 1 The BATS Algorithm

---

procedure BATS(Offline Dataset $D = \bigcup_{j\in[M]}\{(s_{ji}, a_{ji}, s'_{ji}, r_{ji})\}_{i=1}^{t_j}$, Max Stitching Length K, Number of Iteration n, Number of Samples per Iteration m, Neighborhood Radius ϵ, Planning Tolerance δ, Discount Factor γ, Penalty Coefficent c, and Distance Metric d)

$\quad M_0 = \langle \mathcal{S}_0, \{\mathcal{A}_0^s\}_{s\in\mathcal{S}_0}, \gamma, T_0, r_0, \rho_0 \rangle$
$\quad$ Learn dynamics estimate, $\tilde{T}$, and reward estimate, $\tilde{r}$ from D
$\quad$ **for** $i = 0, 1, \ldots, (n-1)$ **do**
$\quad\quad \hat{V}_i^*(\cdot), \pi_i(\cdot) \leftarrow$ valueIteration(M_i)
$\quad\quad$ neighbors $\leftarrow$ getNeighbors$(\mathcal{S}, \pi, M_i, \epsilon)$
$\quad\quad M_{i+1} \leftarrow M_i$
$\quad\quad$ **for** $j = 1, 2, \ldots, m$ **do**
$\quad\quad\quad s \sim \mu_{M_i}(s \mid \pi_i)$
$\quad\quad\quad E \leftarrow$ getCandidateEdges$(s, M_i, \text{neighbors}, j)$
$\quad\quad\quad$ **for all** $(s, s') \in E$ **do**
$\quad\quad\quad\quad$ actions $=$ stitch$((s, s'), K, \delta)$
$\quad\quad\quad\quad s'' = s$
$\quad\quad\quad\quad$ **for all** $a \in$ actions **do**
$\quad\quad\quad\quad\quad \mathcal{A}_{i+1}^{s''} \geq \{a\} \cup \mathcal{A}_{i+1}^{s''}$
$\quad\quad\quad\quad\quad$ **if** a is last action in actions **then**
$\quad\quad\quad\quad\quad\quad T_{i+1}(s'', a) \leftarrow \tilde{T}(s'', a)$
$\quad\quad\quad\quad\quad\quad r_{i+1}(s'', a) \leftarrow \tilde{r}(s'', a) - cd\left(\tilde{T}(s'', a), T(s'', a)\right)$
$\quad\quad\quad\quad\quad\quad s'' \leftarrow \tilde{T}(s'', a)$
$\quad\quad\quad\quad\quad\quad \mathcal{S}_{i+1} \leftarrow \{s''\} \cup \mathcal{S}_{i+1}$
$\quad\quad\quad\quad\quad$ **else**
$\quad\quad\quad\quad\quad\quad T_{i+1}(s'', a) \leftarrow s'$
$\quad\quad\quad\quad\quad\quad r_{i+1}(s'', a) \leftarrow \tilde{r}(s'', a) - cd\left(\tilde{T}(s'', a), s'\right)$
$\quad\quad\quad\quad\quad$ **end if**
$\quad\quad\quad\quad$ **end for**
$\quad\quad\quad$ **end for**
$\quad\quad$ **end for**
$\quad$ **end for**
$\quad V(\cdot), \pi(\cdot) \leftarrow$ valueIteration(M_n) **return** M_n
end procedure

Algorithm 2 Subroutines for BATS

function getCandidateEdges($s, t, \hat{M}$, neighbors, j)
    if k == 0 **then return** []
    end if
    edges = []
    vertexNeighbors $\leftarrow$ neighbors[t]
    MDPNeighbors $\leftarrow \hat{M}[s, \hat{\mathcal{A}}_s]$
    for all $n \in$ vertexNeighbors **do**
        edges += getFutureEdges($s, n, \hat{M}, j$)
    end for
    for all $n \in$ MDPNeighbors **do**
        successorNeighbors $\leftarrow$ neighbors[t]
        for all $n' \in$ successorNeighbors **do**
            neighbors += [(s, n')]
        end for
        neighbors += getCandidateEdges($s, n, \hat{M}$, neighbors, $j - 1$)
    end for
    return edges
end function
function getFutureEdges($s, n, \hat{M}, j$)
    MDPNeighbors $\leftarrow \hat{M}[s, \hat{A}_s]$
    edges $\leftarrow$ []
    for all $n' \in$ MDPNeighbors **do**
        edges += [(s, n')]
        edges += getFutureEdges($s, n', \hat{M}, j - 1$)
    end for
    return edges
end function
function stitch(e, k, ϵ)
    minDistance $\leftarrow \infty$
    bestActions $\leftarrow$ []
    for i = 1:j **do**
        distance, actions $\leftarrow$ CEM(e, k)
        if distance < minDistance **then**
            minDistance $\leftarrow$ distance
            bestActions $\leftarrow$ actions
        end if
    end for
    if minDistance < ϵ **then return** []
    end if
    return bestActions
end function

B   Additional Explanation of Bisimulation

In this work, we make use of *bisimulation metrics* as introduced by Givan et al. [7] in order to guarantee accurate value function estimates in our stitched graph. Informally, bisimulation metrics compare states based solely on their expected future reward distribution and thus ignore information that does not affect the reward. Specifically, they are constructed in Ferns et al. [6] as a generalization of the following equivalence relation:

Definition 1. $\sim$ *is a bisimulation relation if* $s \sim s'$ *implies that* $\forall a \in \mathcal{A}, r(s, a) = r(s', a)$ *and* $T(s, a) \sim T(s', a)$.

This means that two states are *bisimilar* (related via $\sim$) if they attain the same one-step rewards and their future state distributions also return the same under the same actions. Intuitively, this relation serves to ignore attributes of the state space which are not related to the returns attained executing actions. This relation and its derived equivalence classes group together the states in which the agent receives the same rewards under the same actions for an arbitrary choice of actions. However, the restriction that states are only similar if they give the same future rewards under any sequence of actions is quite strong–if an obviously bad action executed in a pair of otherwise bisimilar states gives different rewards which are both bad, those states will not be bisimilar. Any competent reinforcement learning algorithm will not take an obviously bad action, so the states will be indistinguishable for RL purposes.

To rectify this, Castro [2] give a coarser bisimulation relation and associated metric which is based on the actions a particular policy would take:

Definition 2. *The on-policy bisimulation relation* $\sim_\pi$ *for a stochastic policy* π *is the strongest relation such that* $s \sim_\pi s'$ *iff* $\sum_a \pi(a \mid s) r(s, a) = \sum_a \pi(a \mid s') r(s', a)$ *and* $\forall C \in S/ \sim_\pi$, $P^\pi(C \mid s) = P^\pi(C \mid s')$ *where* $S/ \sim_\pi$ *is the partition induced by* $\sim_\pi$ *and* $P^\pi(C \mid s) = \sum_{a \in \mathcal{A}} (\pi(a \mid s) \mathbf{1}[T(s, a) \in C])$.

These works also give metrics which relax these equivalence classes, preserving a notion of approximate bisimilarity. In our work, we use the *on-policy bisimulation distance* from Castro [2], which also gives a sampling-based algorithm for approximating such a metric, which we'll denote $d_\sim^\pi(\cdot, \cdot)$. As one might imagine, it turns out that this distance is closely related to the value function for π by the following theorem from the paper:

Theorem 3 (Theorem 3 from Castro [2])**.** *Given any two states* $s, t \in \mathcal{S}$ *in an MDP* M *and a policy* π, $|V^\pi(s) - V^\pi(t)| \leq d_\sim^\pi(s, t)$.

This result gives us a metric over states for which the value function is 1-Lipschitz continuous. This property allows us to unify the graphical perspective of stitching we take with the more traditional value function approach to RL.

There is an extensive literature on bisimulation, and further discussion, theory, and empirical investigation can be found in [5, 6, 28, 2].

C   Proof of Theorem 2

We prove theorem 2 in this section. The proof proceeds by taking the infinite expansion of a value function and correcting for planning errors using Theorem 1. We can do this for each transition which is added by BATS instead of being take from the dataset, allowing us to compare the upper and lower bounds with their true values.

Proof. WLOG, suppose that for a fixed $z \leq \ell$, $\pi^-(b_i) = a_i$ if $i \leq z$ and $\pi^-(b_i) \neq a_i$ if $z < i \leq \ell$. In other words, the π^- chooses to take advantage of z of the stitches made. Note that z can possibly be 0, but in this case the theorem holds trivially.

Let T^- be the transition function in M^-, and let $s_0, s_1, \ldots$ be the infinite sequence of states that π^- visits in M^- starting from $s_0 = s$ and where $\hat{s}_i = T^-(\hat{s}_{i-1}, \pi^-(\hat{s}_{i-1}))$. Let τ be the ordered set hitting times of the states where π^- uses a stitched transition plus 0 and ∞, i.e. $\tau = \{0\} \cup \{t | t \in \mathbb{N}^+ \text{ s.t. } s_t \in \{b_i\}_{i=1}^z\} \cup \{\infty\}$, and let t_i be the ith sorted element of τ.

We can expand the value function $V(s)$ as follows:

$$V_M^{\pi^-}(s) = \sum_{i=0,s_0=s,s_i=T(s_{i-1},\pi^-(s_{i-1}))} \gamma^i r(s_i, \pi^-(s_i))$$

$$= \sum_{t_i \in \tau} \left(\sum_{j=t_i, s_j=T(s_{j-1},\pi^-(s_{j-1}))}^{j=t_{i+1}-1} \gamma^j r(s_j, \pi^-(s_j)) \right).$$

Leveraging our mapping, ϕ, and Theorem 1, note that

$$V_M^{\pi^-}(s_{t_i}) = r(s_{t_i}, \pi(s_{t_i})) + \gamma V_M^{\pi^-}(s_{t_i+1})$$

$$= r(s_{t_i}, \pi(s_{t_i})) + \gamma V(s_{t_i+1}) + \gamma V_M^{\pi^-}(c_{t_i+1}) - \gamma V_M^{\pi^-}(c_{t_i+1})$$

$$\leq r(s_{t_i}, \pi(s_{t_i})) - \gamma \|\phi(s_{t_i+1}) - \phi(c_{t_i})\| + \gamma V_M^{\pi^-}(c_{t_i})$$

$$\leq r(s_{t_i}, \pi(s_{t_i})) - \gamma \epsilon_{t_i} + \gamma V_M^{\pi^-}(c_{t_i})$$

We can apply this inequality at each hitting time to get the below:

$$V_M^{\pi^-}(s) \geq \sum_{t_i \in \tau} \left(\gamma^{t_i} r(s_{t_i}, \pi^-(s_{t_i})) - \gamma^{t_i+1}\epsilon_i + \sum_{j=t_i+1, s_{t_i}=c_{t_i}, s_j=T^-(s_{j-1},\pi^-(s_{j-1}))}^{j=t_{i+1}-1} \gamma^j r(s_j, \pi^-(s_j)) \right)$$

$$V_M^{\pi^-}(s) \leq \sum_{t_i \in \tau} \left(\gamma^{t_i} r(s_{t_i}, \pi^-(s_{t_i})) + \gamma^{t_i+1}\epsilon_i + \sum_{j=t_i+1, s_{t_i}=c_{t_i}, s_j=T^-(s_{j-1},\pi^-(s_{j-1}))}^{j=t_{i+1}-1} \gamma^j r(s_j, \pi^-(s_j)) \right).$$

Note that, by construction, these lower and upper bounds equal the value functions in the MDPs M^- and M^+, respectively.

$$V_{M^-}^{\pi^-}(s) = \sum_{t_i \in \tau} \left(\gamma^{t_i} r(b_{t_i}, \pi^-(b_{t_i})) - \gamma^{t_i+1}\epsilon_i + \sum_{j=t_i+1, s_{t_i}=c_{t_i}, s_j=T^-(s_{j-1},\pi^-(s_{j-1}))}^{j=t_{i+1}-1} \gamma^j r(s_j, \pi^-(s_j)) \right)$$

$$\tag{3}$$

and

$$V_{M^+}^{\pi^-}(s) = \sum_{t_i \in \tau} \left(\gamma^{t_i} r(b_{t_i}, \pi^-(b_{t_i})) + \gamma^{t_i+1}\epsilon_i + \sum_{j=t_i+1, s_{t_i}=c_{t_i}, s_j=T^-(s_{j-1},\pi^-(s_{j-1}))}^{j=t_{i+1}-1} \gamma^j r(s_j, \pi^-(s_j)) \right).$$

$$\tag{4}$$

Combining the above gives the desired result for our arbitrary s:

$$V_{M^-}^{\pi^-}(s) \leq V_M^{\pi^-}(s) \leq V_{M^-}^{\pi^-}(s).$$

$\square$

D   Experiment Details

D.1   Hyperparameters and Training Procedure

Dynamics Models. To learn dynamics models, we use the architecture introduced by Chua et al. [4], and follow the procedure described in Yu et al. [26], making a few minor changes. Like Yu et al. [26], we train seven different dynamics models and take the best five based on a validation set of 1,000 points. Each model is a neural network with 4 hidden layers of size 200 and 2 heads at the end: one predicting mean and one predicting log-variance. These models are trained using batches of size 256 and using negative log likelihood as the loss. We use ReLU for our hidden activation function, and unlike Yu et al. [26], we do not use spectral normalization. Following their procedure, we use a validation set of one thousand points to select the best model to use after training.

BATS. For each of the experiments in Section 7, we perform BATS three times, using different learned dynamics models, to produce three different stitched MDPs. When using CEM to plan new

	umaze	medium	large
Planning Error Threshold (δ)		0.425	
Penalty Coefficient (c)		20	10
Policy Layer Sizes	$64, 64$	$256, 256$	$256, 256, 256$
Behavior Cloning Batch Sizes		256	
Behavior Cloning Batch Updates	$10,000$		$20,000$

Table 2: **Table of BATS Hyperparameters.**

stitches, we use the mean output of each member of the dynamics model and check if the 80th quantile is under some planning threshold. This planning threshold was set to 0.425 with for the mazes and 10 (after normalization) for Mujoco tasks. All experiments imposed a restriction of $K = 1$ number of actions that could be taken for stitching Additionally, when forming the nearest neighbor graph for finding potential stitches, we consider neighbors up to 0.225 away for umaze and medium maze and 0.15 away for large maze. These were set as large as memory constraints would allow for.

We assume for these experiments that we have access to the start state distribution. Since there is only one trajectory for each of the maze tasks, we label every state that is in the support of the start state distribution as a start state. The large maze dataset does not contain possible start states for all cells. For cells in which there is not a start state in the dataset, we widen the distribution slightly so that enough starts are included.

For the maze tasks we attempt $m = 50,000$ stitches every iteration, and we run BATS for $n = 10$ iterations for umaze and medium maze and 20 iterations for the large maze.

To increase exploration for the stitches to consider, we apply Boltzmann exploration when selecting next actions to perform in the stitched MDP. That is, we select actions according to:
$$\mathbb{P}(a|s) \propto \exp\left(Q(s, a)/T\right)$$
where T is a temperature parameter, which we choose to set to 0.25 for all experiments.

After running BATS, we searched for good hyperparameters by relabeling or removing stitched edges accordingly. The best found parameters are shown in Table 2. When looking at the distribution of returns from trajectories in the resulting MDPs, there is a clear value for the returns that separates successfully stitched trajectories from those that were not able to be stitched to high value areas. As such, we only behavior clone on trajectories above 100 (umaze), 200 (medium maze), 300 (large maze), $1,000$ (hopper and walker2d), and $4,000$ (halfcheetah). These thresholds were selected by inspection.

Mountaincar.

For the mountain car example, we found that doing 20 iterations of stitching (each trying to make 100 stitches) was sufficient. We use the nearest 25 neighbors to determine which stitches can be made, and we allow for up to $K = 5$ actions to be used when stitching between states. We found that smaller dynamics models were sufficient for this problem, and in particular, each member of the ensemble had 3 layers with 64 hidden units each. Lastly, we set the the temperature for Boltzmann exploration to $T = 0.1$. We behavior clone using a policy network with two hidden layers with 256 units each.

D.2 Bisimulation Implementation

The model architecture that we use for bisimulation is that of Zhang et al. [28]. That is, we have a network that takes state observations as inputs and outputs a latent representation, and we have a dynamics model that operates on this latent representation. For the encoder network, we use three hidden layers with 256, 128, and 64 units, respectively, and we set the latent dimension to be 6. Unlike the model in Zhang et al. [28]. however, we have one network that predicts both next transitions in bisimulation space and next rewards (the same dynamics model as described in Appendix D.1). We also use the same loss function as described in Zhang et al. [28]. In particular, we draw batches of pairs of state observations and optimize according to

$$J(\phi) = \left(\|z_i - z_j\| - |r_i - r_j| - \gamma W_2\left(\hat{P}(\cdot|\bar{z}_i, a_i), \hat{P}(\cdot|\bar{z}_j, a_j)\right)\right)^2$$

where z_k, r_k, a_k are the latent encoding, the predicted reward, and the observed action for the kth sample, respectively. $\hat{P}$ is the learned dynamics model for the latent space, and we use a bar over z to signify that we stop gradients. For more details, please refer to Zhang et al. [28]. Although in their work they iteratively update their model to reflect a changing policy, in our work we train with respect to a fixed policy. The on-policy nature of our training procedure resembles Castro [2].

D.3 COMBO Implementation

Because the original COMBO [27] paper did not include results on the maze environments and does not yet have a public implementation, we made our best attempt at reimplementing their method in order to properly compare results. For running COMBO, we based our dynamics model training on Yu et al. [26] and used all of their associated hyperparameters. For the conservative Q-learning and policy learning components, we mostly followed the public implementation of Kumar et al. [14], but had to make some small tweaks to make it consistent with the descriptions in Yu et al. [27].

For the COMBO hyperparameters, we did a grid search over conservative coefficient β in $\{0.5, 1.0, 5.0\}$ and rollout length h in $\{1, 5\}$ for all the maze tasks. We found the best parameters to be $h = 1, \beta = 1$ on umaze, $h = 1, \beta = 0.5$ on medium, and $h = 1, \beta = 0.5$ on large. For all the other hyperparameters, we followed the halcheetah parameters used in Yu et al. [27]. Specifically, 3-layer feedforward neural networks with 256 hidden units for Q-networks and policy, Q-learning rate $3.e - 4$, policy learning rate $1.e - 4$, $\rho(a|s)$ being the soft-maximum of the Q-values and estimated with log-sum-exp, $\mu(a|s) = \pi(a|s)$, and deterministic backups.