A MBOLD IMPLEMENTATION DETAILS

A.1 DISTANCE FUNCTION

This section explains the implementation details for our distance function. Following prior work (Fujimoto et al., 2018), we learn two independent Q-functions and use the *minimum* for performing Bellman backups. Recall that we sampled goals from two distributions: future states in the same trajectories, and states from different trajectories where the robot arm was in a similar position. To implement the second strategy, we fit a k-nearest neighbors graph on 200000 (about 60% of total) dataset observations, and use the ℓ_2 arm joint distance as the similarity key. Each batch contains equal numbers of transitions generated from each goal sampling method. For computational efficiency, we implement the k-NN search using the GPU-enabled FAISS library (Johnson et al., 2017).

We also modify the reward specification scheme by providing a small positive reward at each step where the goal is not reached, and then a large positive reward upon reaching the goal. Specifically, we choose to give a reward of 1 by default and 10 when the goal is reached (compared to 0 and 1 respectively as presented in the discussion in Section 4), although we do not extensively tune this parameter. We find that it does not affect performance in a statistically significant way (results for each reward choice are within 1 standard deviation of one another) to choose this reward over the $(0, 1)$ rewards. Note that this does not change the interpretation of the Q-function as a shortest path distance, merely slightly complicating the conversion calculations from Q-values to distances in timesteps.

Finally, we add an additional loss term to perform conservative Q-learning (CQL) (Kumar et al., 2020), a method for offline model-free RL, which penalizes Q-values of randomly selected actions and increases Q-values of in-dataset actions. We use the Lagrangian version of CQL to automatically tune the weighting term, and detail the parameters below. We find using CQL improves performance on the door sliding task from a mean success rate of 41% to 58%, but does not significantly impact performance on the others.

The Q-function network architecture consists of convolutional and fully connected layers. We define a network called the *convolutional encoder*, which will be used throughout the appendix. This takes as input an image of shape $64 \times 64 \times 6$, containing the starting and goal images concatenated channel-wise, and consists of 4 2D convolutional layers, with $[8, 16, 32, 64]$ filters, respectively, with all with kernel size $(4, 4)$ and strides of $(2, 2)$. We use Leaky ReLU activations after each intermediate convolutional layer, and batch-norm layers after the second and third Leaky ReLUs.

We flatten the output of the convolutional encoder and feed the features through 6 fully-connected linear layers of 128 units each, with the final layer outputting a single value. Each intermediate fully-connected layer is followed by a ReLu activation and a batch-norm layer.

The actor network architecture first contains the above "convolutional encoder", whose outputs are flattened and input into a 10 layer MLP with 128 fully connected units each, and ReLu activations and batch-norm layers in between. The final output, of dimension 4, is passed through a $\tanh$ activation to constrain it to the normalized action space $[-1, 1]$.

Additional training hyperparameters are detailed in Table 2.

A.2 MODEL-PREDICTIVE CONTROL

In Table 3, we describe the parameters for model-based planning in our experiments. These parameters are shared across all tasks and planning costs (in ablation experiments). Most values are selected based on prior work (Ebert et al., 2018b). We find that replanning every 6 steps produces slightly better performance than replanning every 13 steps, but not by a large margin, and we do not tune this further due to computation constraints. We sample actions using the filtering scheme described in Nagabandi et al. (2020) to make sequences smoother in time. We initialize sampling distributions using each environment's data collection parameters, as shown in Table 4.

To compute the planning cost described in Equation 3, we maximize over α by feeding in the final predicted state to the policy network learned by TD3, and using the outputted action as the maximizer.

Dataset size	10000 trajectories
Train/test/val split	$0.9/0.05/0.05$
Trajectory length	30 steps
Observation dimensions	$64 \times 64 \times 3$
State observations in kNN graph	200000
Goal relabeling sampling parameter (p)	0.3 (tuned over $[0.2, 0.3]$)
Discount factor (γ)	0.8
Learning rate	3e-4
Target network update Polyak factor	0.995
Batch size	64
Actor network noise σ	0.1
Actor network maximum noise magnitude	0.2
Training iterations	93750 (300 epochs)
Optimizer	Adam
CQL Lagrange multiplier learning rate	1e-3
CQL slack parameter τ (object pushing)	3.0
CQL slack parameter τ (reaching)	3.0
CQL slack parameter τ (door sliding)	10.0
CQL number of randomly selected actions	10

Table 2: Hyperparameters for distance learning

Planning horizon (h)	13 steps
Actions executed per planning step (k)	6 actions
CEM Iterations	3 iterations
Elite sample fraction	0.05 (10 samples)
Samples per CEM iteration	200 samples

Table 3: Hyperparameters for model-based planning

A.3 Environments

The Sawyer environments are adapted from the Meta-World benchmark (Yu et al., 2019a), and the *door sliding* environment is based off of the environment presented by Lynch et al. (2020). For each task, we define the 4-dimensional action space $\mathcal{A}$ such that actions control the Cartesian position of the robot's end-effector, as well as the robot's gripper.

We randomly generate a set of 100 different test goals for each setting. Each task is defined by a goal image and starting state, on which all methods are tested. We define success for each task in terms of the final distance to the goal of each relevant object, e.g. object position for the object repositioning task. A trial is considered successful if the final distance is below a certain threshold ϵ manually chosen for each task, listed in the table below. We evaluate the success rate of each method over 5 different random training seeds.

We generate offline datasets for each task by running random policies for $1e4$ episodes of 30 timesteps each. The random policy actions are drawn using a filtering technique, which smooths random zero-mean Gaussian samples across time. We apply the correlated noise scheme described by Nagabandi et al. (2020), setting the hyperparameter $\beta = 0.5$. The parameters of the multi-variate Gaussian samples in each dimension are listed in Table 4.

	Reaching	Object pushing	Door sliding
Data colln. stdev ($diag(\Sigma)$)	[0.6, 0.6, 0.3, 0.3]	[0.6, 0.6, 0.3, 0.3]	[0.3, 0.3, 0.3, 0.15]
Object compared in success threshold	Arm end effector	Object	Slide
Success distance threshold	0.05m	0.05m	0.075m

Table 4: Environment and task details

B COMPARATIVE EVALUATION IMPLEMENTATION DETAILS

B.1 RIG

In this section, we will discuss implementation details of our adaptation of RIG. We begin by training a β-VAE with latent dimension 8. The VAE is trained on randomly sampled states from the entire offline dataset. For the loss, we use a combination of a maximum likelihood term and a KL divergence term which constrains the latent space to a unit Gaussian. In particular, we compute the mean pixel error, that is, $\frac{1}{HW}\|s - \hat{s}\|_2^2$, where s is the original image, and $\hat{s}$ is the reconstruction, both normalized to be in $[0, 1]$. We add this to the KL divergence between the latent distribution and the unit Gaussian, with a weighting factor of $1e^{-3}$ on the KL penalty.

The architecture of the VAE encoder consists of the "convolutional encoder" described in section A.1, whose features are passed through two FC layers with 128 units with a ReLu activation and batch-norm layer in between. The VAE decoder takes as input latent states into two FC layers with 128 units with a batch-norm layer and ReLu activation after each. This is followed by the inverted architecture of the encoder, consisting of transposed 2D convolutions.

Then, we perform model-free RL in a modified MDP, using encoded observations as a substitute for environment observations, and computing rewards as negative ℓ_2 distances in latent space. We sample random goals from the multivariate Gaussian prior ($\mathcal{N}(0, I)$) at the beginning of every episode. We use the open-source implementation of soft actor-critic (SAC) in RLKit, and use the default SAC parameters and architecture found in the implementation, making the following modifications: We increase the number of layers of all MLP networks from 2 to 6. We use a maximum path length of 30 steps for consistency with our other experiments, and a discount factor of 0.95. Along with the goal sampled from the prior at the beginning of each episode, we find that relabeling goals with the achieved observation at the end of the trajectory improves performance, and add these transitions to the replay buffer as well. Note that unlike in the original RIG formulation, we do *not* update the weights of the learned VAE using data collected online. We evaluate the learned policy after 600 epochs of training, long after environment returns plateau.

B.2 DREAMER

Dreamer, a model-based method for image-based tasks, also uses a combination of value functions and planning. We adapt Dreamer from its original single-task setting to learn a goal-conditioned policy, reward predictor, and value function; however, we do not condition the dynamics model on the goal. Dreamer has been previously demonstrated only in settings where the environment provides rewards to the agent, so we modify the method to learn from unlabeled, offline data by using experience replay. We find that using an indicator reward function as in our method or a heuristically defined reward function, image MSE, causes Dreamer to struggle to learn. We thus additionally demonstrate the performance of Dreamer using a manually specified arm distance reward for the Sawyer reaching task.

We build off of the open source implementation of Dreamer by the original authors, written in TensorFlow2 and found at https://github.com/danijar/dreamer. Specifically, to modify the networks to support goal-conditioning, we add independent convolutional encoders which take the goal image as input to each network. Each encoder consists of 2D convolution layers with $[32, 64, 128, 256]$ filters and kernel sizes of 4 to each network, and we concatenate the flattened features to the inputs of each network. We additionally increase the number of fully-connected layers for the value and actor networks from 3 and 2 respectively to 10. We use a discount factor of $\gamma = 0.95$. All other hyperparameter values are defaults from the public implementation.

For training, we relabel trajectories sampled from the fixed, offline dataset with a uniformly randomly selected observation from the trajectory as the goal. In most of our experiments, we compute the negative pixel-wise MSE as the reward, but in one reaching experiment, we use the negative ℓ_2 Euclidean distance between the arm end-effector position and the goal end-effector position. We train for 2000 iterations for each experiment, although initial experiments in which we trained for 20x longer did not yield improved results.

B.3 GOAL-CONDITIONED BEHAVIOR CLONING

To train a goal-conditioned behavior cloning policy, we begin by relabeling random transitions from the dataset with goals which are later achieved in those trajectories. Specifically, we sample state-goal pairs from trajectories in the dataset by first selecting the initial state index t_i uniformly from all timesteps, and then selecting the goal state index t_g uniformly from timesteps greater than t_i. We then train a neural network to predict the transition action a_i given the state s_i and the relabeled goal s_g, using a mean-squared error loss.

The network architecture is the same as that of the actor network used in Q-learning for MBOLD, described in Appendix A.1. We train the model for 3125000 iterations (1000 epochs) using a batch size of 32, and use the same optimizer and learning rate as the distance learned for MBOLD.

B.4 SEARCH ON THE REPLAY BUFFER

For SoRB, we train a distributional Q-function to represent distances as in the original paper. Distributional RL discretizes possible value estimates into a set of bins – we use 10 for all of our experiments. We train this distributional Q-function for 300 epochs, as in the distance function training for MBOLD. We also use the same architecture and training scheme, altering the number of outputs to 10 bins and using the KL-divergence loss for the distributional Q-function as in Eysenbach et al. (2019). However, unlike in Eysenbach et al. (2019), we train on just the fixed, offline dataset. We then perform the planning portion of SoRB with the "maxdist" parameter set to 4, after manual tuning. We use a graph size of 2000 states for all experiments, due to computational constraints.

We find that the policy learned through Q-learning performs very poorly at reaching subgoals, so we instead substitute the GCBC policy for this purpose. We find that this greatly improves performance across all tasks.

B.5 VISUAL FORESIGHT

To compare MBOLD to visual foresight, we use the same dynamics model and planning setup as in MBOLD, however, we substitute the learned dynamical distance function with the ℓ_2 pixel error cost used in visual foresight.

C ABLATION EXPERIMENTS IMPLEMENTATION DETAILS

C.1 VAE DISTANCE

We use the same architecture as the VAE used in the RIG comparison described in Appendix B. We set the latent space dimension to 256 and weight the KL divergence term using a factor of $1e^{-5}$. We train the model for 3125000 iterations (1000 epochs) using a batch size of 32, and use the same optimizer and learning rate as the distance learned for MBOLD.

C.2 TEMPORAL DISTANCE REGRESSION

To train the temporal distance regression model, we sample state-goal pairs from trajectories in the dataset by first selecting the initial state index t_i uniformly from all timesteps, and then selecting the goal state index t_g uniformly from timesteps greater than t_i. We compute the label for this pair as $\min(t_g - t_i, maxdist)$, where $maxdist$ is a hyperparameter we set to 10. The $maxdist$ parameter helps to improve the optimality of distances on average. We train the neural network to regress this target label using an ℓ_2 error loss. We train the network for 3125000 iterations (1000 epochs) with a batch size of 32, and use the same optimizer and learning rate as the distance learned for MBOLD.

The architecture for the temporal distance regression model begins with the convolutional encoder described in Appendix B. Its flattened outputs are fed into 5 fully-connected layers of 256 units each, with batch-norm and ReLu activations after each intermediate layer.

C.3 Q-FUNCTION POLICY

We find that the policy directly learned by our method when learning distances performs extremely poorly. However, performing Q-learning using random shooting over 100 uniformly random actions selected from $[-1, 1]^4$ to optimize over actions to compute target values produces much better results when used directly as a policy, compared to using an actor network to perform this optimization as in our method. Therefore, we report results from acting according to this random shooting method. At test time, we estimate the optimal action $a^\star = \arg\max_a Q(s_t, a, g)$ by again sampling 100 uniformly random actions, and selecting the best one.