# A. Appendix

## A.1. Proof of Theorem 1 connecting NML and inverse counts

We provide the proof of Theorem 1 here for completeness.

**Theorem 2.** *Suppose we are estimating success probabilities $p(e = 1|s)$ in the tabular setting, where we have a separate parameter independently for each state. Let $N(s)$ denote the number of times state $s$ has been visited by the policy, and let $G(s)$ be the number of occurrences of state $s$ in the successful outcomes. Then the CNML probability $p_{CNML}(e = 1|s)$ is equal to $\frac{G(s)+1}{N(s)+G(s)+2}$. For states that are never observed to be successful, we then recover inverse counts $\frac{1}{N(s)+2}$.*

*Proof.* In the fully tabular setting, our MLE estimates for $p(O|s)$ are simply given by finding the best parameter $p_s$ for each state. The proof then proceeds by simple calculation.

For a state with $n = N(s)$ negative occurrences and $g = G(s)$ positive occurrences, the MLE estimate is simply given by $\frac{g}{n+g}$.

Now for evaluating CNML, we consider appending another instance for each class. The new parameter after appending a negative example is then $\frac{g}{n+g+1}$, which then assigns probability $\frac{n+1}{n+g+1}$ to the negative class. Similarly, after appending a positive example, the new parameter is $\frac{g+1}{n+g+1}$, so we try to assign probability $\frac{g+1}{n+g+1}$ to the positive class. Normalizing, we have

$$p_{\text{CNML}}(O = 1|s) = \frac{g+1}{n+g+2}. \tag{10}$$

When considering states that have only been visited on-policy, and are not included in the set of successful outcomes, then the likelihood reduces to

$$p_{\text{CNML}}(O = 1|s) = \frac{1}{n+2}. \tag{11}$$

□

## A.2. Detailed Description of Meta-NML

We provide a detailed description of the meta-NML algorithm described in Section 5, and the details of the practical algorithm.

Given a dataset $\mathcal{D} = \{(x_0, y_0), (x_1, y_1), .., (x_n, y_n)\}$, the meta-NML procedure proceeds by first constructing $k * n$ tasks from these data points, for a $k$ shot classification problem. We will keep $k = 2$ for simplicity in this description, in accordance with the setup of binary success classifiers in RL. Each task $\tau_i$ is constructed by augmenting the dataset with a negative label $\mathcal{D} \cup (x_i, y = 0)$ or a positive label $\mathcal{D} \cup (x_i, y = 1)$. Now that each task consists of solving the maximum likelihood problem for its augmented dataset, we can directly apply standard meta-learning algorithms to this setting. Building off the ideas in MAML (Finn et al., 2017), we can then train a set of model parameters $\theta$ such that after a single step of gradient descent it can quickly adapt to the optimal solution for the MLE problem on any of the augmented datasets. This is more formally written as

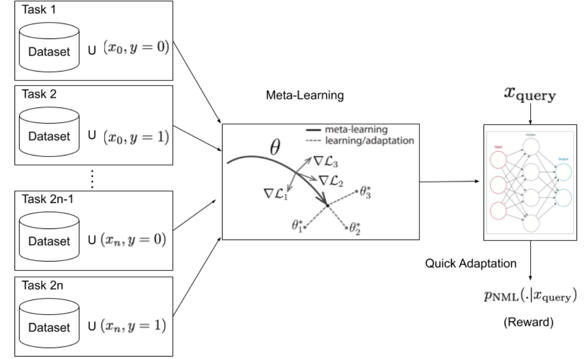


*Figure 9.* Figure illustrating the meta-training procedure for meta-NML.

$$\max_{\theta} \; \mathbb{E}_{\tau \sim \mathcal{S}(\tau)}[\mathcal{L}(\tau, \theta')], \quad s.t \;\; \theta' = \theta - \alpha \nabla_{\theta} \mathcal{L}(\tau, \theta) \tag{12}$$

where $\mathcal{L}$ represents a standard classification loss function, $\alpha$ is the learning rate, and the distribution of tasks $p(\tau)$ is constructed as described above. For a new query point $x$, these initial parameters can then quickly be adapted to provide the CNML distribution by taking a gradient step on each augmented dataset to obtain the approximately optimal MLE solution, and normalizing these as follows:

$$p_{\text{meta-NML}}(y|x; \mathcal{D}) = \frac{p_{\theta_y}(y|x)}{\sum_{y \in \mathcal{Y}} p_{\theta_y}(y|x)}$$
$$\theta_y = \theta - \alpha \nabla_{\theta} \mathbb{E}_{(x_i, y_i) \sim \mathcal{D} \cup (x, y)}[\mathcal{L}(x_i, y_i, \theta)]$$

This algorithm in principle can be optimized using any standard stochastic optimization method such as SGD, as described in Finn et al. (2017), backpropagating through the inner loop gradient update. For the specific problem setting that we consider, we additionally employ some optimization tricks in order to enable learning:

#### A.2.1. IMPORTANCE WEIGHTING ON QUERY POINT

Since only one datapoint is augmented to the training set at query time for CNML, stochastic gradient descent can ignore this datapoint with increasing dataset sizes. For example, if we train on an augmented dataset of size 2048 by cycling through it in batch sizes of 32, then only 1 in 64 batches would include the query point itself and allow the model to adapt to the proposed label, while the others would lead to noise in the optimization process, potentially worsening the model's prediction on the query point.

In order to make sure the optimization considers the query point, we include the query point and proposed label $(x_q, y)$ in every minibatch that is sampled, but downweight the loss computed on that point such that the overall objective remains unbiased. This is simply doing importance weighting, with the query point downweighted by a factor of $\lceil \frac{b-1}{N} \rceil$ where $b$ is the desired batch size and $N$ is the total number of points in the original dataset.

To see why the optimization objective remains the same, we can consider the overall loss over the dataset. Let $f_\theta$ be our classifier, $\mathcal{L}$ be our loss function, $\mathcal{D}' = \{(x_i, y_i)\}_{i=1}^N \cup (x_q, y)$ be our augmented dataset, and $\mathcal{B}_k$ be the $k$th batch seen during training. Using standard SGD training that cycles through batches in the dataset, the overall loss on the augmented dataset would be:

$$\mathcal{L}(\mathcal{D}') = \left( \sum_{i=0}^{N} \mathcal{L}(f_\theta(x_i), y_i) \right) + \mathcal{L}(f_\theta(x_q), y)$$

If we instead included the downweighted query point in every batch, the overall loss would be:

$$\begin{aligned}
\mathcal{L}(\mathcal{D}') &= \sum_{k=0}^{\lceil \frac{b-1}{N} \rceil} \sum_{(x_i, y_i) \in \mathcal{B}_k} \left( \mathcal{L}(f_\theta(x_i), y_i) + \frac{1}{\lceil \frac{b-1}{N} \rceil} \mathcal{L}(f_\theta(x_q), y) \right) \\
&= \left( \sum_{k=0}^{\lceil \frac{b-1}{N} \rceil} \sum_{(x_i, y_i) \in \mathcal{B}_k} \mathcal{L}(f_\theta(x_i), y_i) \right) + \\
&\lceil \frac{b-1}{N} \rceil \frac{1}{\lceil \frac{b-1}{N} \rceil} \mathcal{L}(f_\theta(x_q), y) \\
&= \left( \sum_{i=0}^{N} \mathcal{L}(f_\theta(x_i), y_i) \right) + \mathcal{L}(f_\theta(x_q), y)
\end{aligned}$$

which is the same objective as before.

This trick has the effect of still optimizing the same maximum likelihood problem required by CNML, but significantly reducing the variance of the query point predictions as we take additional gradient steps at query time. As a concrete example, consider querying a meta-CNML classifier on the input shown in Figure 10. If we adapt to the augmented dataset without including the query point in every batch (i.e. without importance weighting), we see that the query point loss is significantly more unstable, requiring us to take more gradient steps to converge.

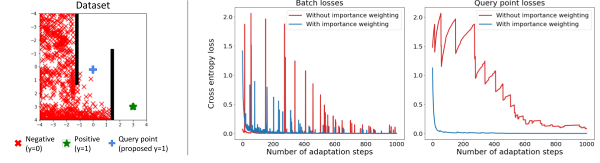


*Figure 10.* Comparison of adapting to a query point (pictured on left with the original dataset) at test time for CNML with and without importance weighting. The version without importance weighting is more unstable both in terms of overall batch loss and the individual query point loss, and thus takes longer to converge. The spikes in the red lines occur when that particular batch happens to include the query point, since that point's proposed label ($y = 1$) is different than those of nearby points ($y = 0$). The version with importance weighting does not suffer from this problem because it accounts for the query point in each gradient step, while keeping the optimization objective the same.

#### A.2.2. KERNEL WEIGHTED TRAINING LOSS

The augmented dataset consists of points from the original dataset $\mathcal{D}$ and one augmented point $(x_q, y)$. Given that we mostly care about having the proper likelihood on the query point, with an imperfect optimization process, the meta-training can yield solutions that are not very accurately representing true likelihoods on the query point. To counter this, we introduce a kernel weighting into the loss function in Equation 12 during meta-training and subsequently meta-testing. The kernel weighting modifies the training loss function as:

$$\begin{aligned}
&\max_\theta \mathbb{E}_{\tau \sim \mathcal{S}(\tau)} [\mathbb{E}_{(x,y) \sim \tau} \mathcal{K}(x, x_\tau) \mathcal{L}(x, y, \theta')] \\
&\quad s.t \;\; \theta' = \theta - \alpha \nabla_\theta \mathbb{E}_{(x,y) \sim \tau} \mathcal{K}(x, x_\tau) \mathcal{L}(x, y, \theta)
\end{aligned}$$

where $x_\tau$ is the query point for task $\tau$ and $\mathcal{K}$ is a choice of kernel. We typically choose exponential kernels centered around $x_\tau$. Intuitively, this allows the meta-optimization to mainly consider the datapoints that are copies of the query point in the dataset, or are similar to the query point, and ensures that they have the correct likelihoods, instead of receiving interfering gradient signals from the many other points in the dataset. To make hyperparameter selection intuitive, we designate the strength of the exponential kernel by a parameter $\lambda_{dist}$, which is the Euclidean distance away from the query point at which the weight becomes 0.1. Formally, the weight of a point $x$ in the loss function for query point $x_\tau$ is computed as:

$$K(x, x_\tau) = \exp\{-\frac{2.3}{\lambda_{dist}}||x - x_\tau||_2\} \tag{13}$$

#### A.2.3. Meta-Training at Fixed Intervals

While in principle meta-NML would retrain with every new datapoint, in practice we retrain meta-NML once every $k$ epochs. (In all of our experiments we set $k = 1$, but we could optionally increase $k$ if we do not expect the meta-task distribution to change much between epochs.) We warm-start the meta-learner parameters from the previous iteration of meta-learning, so every instance of meta-training only requires a few steps. We find that this periodic training is a reasonable enough approximation, as evidenced by the strong performance of MURAL in our experimental results in Section 6.

### A.3. Meta-NML Visualizations

#### A.3.1. Meta-NML with Additional Gradient Steps

Below, we show a more detailed visualization of meta-NML outputs on data from the Zigzag Maze task, and how these outputs change with additional gradient steps. For comparison, we also include the idealized NML rewards, which come from a discrete count-based classifier.

Meta-NML is able to resemble the ideal NML rewards fairly well with just 1 gradient step, providing both an approximation of a count-based exploration bonus and better shaping towards the goal due to generalization. By taking additional gradient steps, meta-NML can get arbitrarily close to the true NML outputs, which themselves correspond to inverse counts of $\frac{1}{n+2}$ as explained in Theorem 4.1. While this would give us more accurate NML estimates, in practice we found that taking one gradient step was sufficient to achieve good performance on our RL tasks.

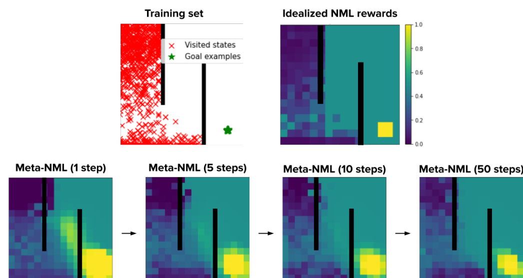


*Figure 11.* Comparison of idealized (discrete) NML and meta-NML rewards on data from the Zigzag Maze Task. Meta-NML approximates NML reasonably well with just one gradient step at test time, and converges to the true values with additional steps.

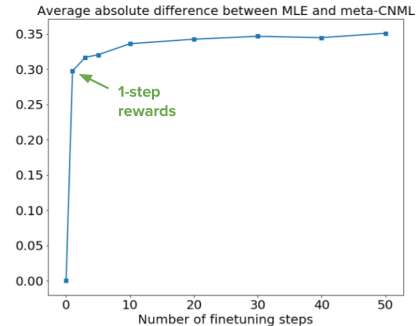


*Figure 12.* Average absolute difference between MLE and meta-NML goal probabilities across the entire maze state space from Figure 11 above. We see that meta-NML learns a model initialization whose parameters can change significantly in a small number of gradient steps. Additionally, most of this change comes from the first gradient step (indicated by the green arrow), which justifies our choice to use only a single gradient step when evaluating meta-NML probabilities for MURAL.

#### A.3.2. Comparison of Reward Classifiers

In Fig 13, we show the comparison between different types of reward. classifiers in the 2D maze navigation problem.

*Figure 13.* A comparison of the rewards given by various classifier training schemes on the 2D Zigzag maze. From left to right: (1) An MLE classifier when trained to convergence reduces to an uninformative sparse reward; (2) An MLE classifier trained with regularization and early stopping has smoother contours, but does not accurately identify the goal; (3) The idealized NML rewards correspond to inverse counts, thus providing a natural exploration objective in the absence of generalization; (4) The meta-NML rewards approximate the idealized rewards well in visited regions, while also benefitting from better shaping towards the goal due to generalization.

#### A.3.3. Runtime Comparisons

We provide the runtimes for feedforward inference, naive CNML, and meta-NML on each of our evaluation domains. We list both the runtimes for evaluating a single input (Table 1), and for completing a full epoch of training during RL (Table 2).

These benchmarks were performed on an NVIDIA Titan X Pascal GPU. Per-input runtimes are averaged across 100 samples, and per-epoch runtimes are averaged across 10

epochs.

## A.4. Experimental Details

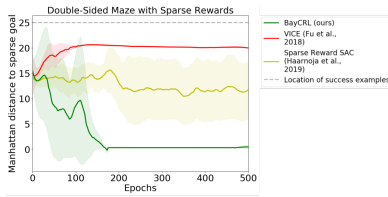


*Figure 14.* Performance of MURAL, VICE, and SAC with sparse rewards on a double-sided maze where some sparse reward states are not provided as goal examples. MURAL is still able to find the sparse rewards, thus receiving higher overall reward, whereas ordinary classifier methods (i.e. VICE) move only towards the provided examples and thus are never able to find the additional rewards. Standard SAC with sparse rewards, also included for comparison, is generally unable to find the goals. The dashed gray line represents the location of the goal examples initially provided to both MURAL and VICE.

### A.4.1. Environments

**Zigzag Maze and Spiral Maze:** These two navigation tasks require moving through long corridors and avoiding several local optima in order to reach the goal. For example, on Spiral Maze, the agent must not get stuck on the other side of the inner wall, even though that position would be close in L2 distance to the desired goal. On these tasks, a sparse reward is not informative enough for learning, while ordinary classifier methods get stuck in local optima due to poor shaping near the goal.

Both of these environments have a continuous state space consisting of the $(x, y)$ coordinates of the agent, ranging from $(-4, -4)$ to $(4, 4)$ inclusive. The action space is the desired velocity in the $x$ and $y$ directions, each ranging from $-1$ to 1 inclusive.

**Sawyer 2D Pusher:** This task involves using a Sawyer arm, constrained to move only in the $xy$ plane, to push a randomly initialized puck to a fixed location on a table. The state space consists of the $(x, y, z)$ coordinates of the robot end effector and the $(x, y)$ coordinates of the puck. The action space is the desired $x$ and $y$ velocities of the arm.

**Sawyer Door Opening:** In this task, the Sawyer arm is attached to a hook, which it must use to open a door to a desired angle of 45 degrees. The door is randomly initialized each time to be at a starting angle of between 0 and 15 degrees. The state space consists of the $(x, y, z)$ coordinates of the end effector and the door angle (in radians); the action space consists of $(x, y, z)$ velocities.

**Sawyer 3D Pick and Place:** The Sawyer robot must pick up a ball, which is randomly placed somewhere on the table each time, and raise it to a fixed $(x, y, z)$ location high above the table. This represents the biggest exploration challenge out of all the manipulation tasks, as the state space is large and the agent would normally not receive any learning signal unless it happened to pick up the ball and raise it, which is unlikely without careful reward shaping.

The state space consists of the $(x, y, z)$ coordinates of the end effector, the $(x, y, z)$ coordinates of the ball, and the tightness of the gripper (a continuous value between 0 and 1). The robot can control its $(x, y, z)$ arm velocity as well as the gripper value.

**Ant Locomotion:** In this task, the quadruped ant robot has to navigate from one end of a maze to the other. This represents a high dimensional action space of 8 dimensions, and a high dimensional state space of 15 dimensions as well. The state space consists of the center of mass of the object as well as the positions of the various joints of the ant, and the action space controls the torques on all the joints.

**Hand Manipulation:** In this task, a 16 DoF robotic hand is mounted on a robot arm and has to reposition an object on a table. The task is challenging due to high dimensionality of the state and action spaces. The state space consists of the arm position, hand joint positions and object positions. In this task, we allow the classifier privileged access to the object position only, but provide the full state space as input to the policy. All the other baseline techniques are provided this same information as well (e.g. the classifier for VICE receives the object position as input).

### A.4.2. Ground Truth Distance Metrics

In addition to the success rate plots in Figure 5, we provide plots of each algorithm's distance to the goal over time according to environment-specific distance metrics. The distance metrics and success thresholds, which were used to compute the success rates in Figure 5, are listed in the table on the next page.

## A.5. Additional Ablations

### A.5.1. Learning in a Discrete, Randomized Environment

In practice, many continuous RL environments such as the ones we consider in section 6 have state spaces that are correlated at least roughly with the dynamics. For instance, states that are closer together dynamically are also typically closer in the metric space defined by the states. This correlation does not need to be perfect, but as long as it exists, MURAL can in principle learn a smoothly shaped reward towards the goal.

However, even in the case where states are unstructured and completely lack identity, such as in a discrete gridworld

| | Feedforward | Meta-NML | Naive CNML |
|---|---|---|---|
| **Mazes (zigzag, spiral)** | 0.0004s | 0.0090s | 15.19s |
| **Sawyer 2D Pusher** | 0.0004s | 0.0092s | 20.64s |
| **Sawyer Door** | 0.0004s | 0.0094s | 20.68s |
| **Sawyer 3D Pick** | 0.0005s | 0.0089s | 20.68s |
| **Ant Locomotion** | 0.0004s | 0.0083s | 17.26s |
| **Dexterous Manipulation** | 0.0004s | 0.0081s | 17.58s |

*Table 1.* Runtimes for evaluating a single input point using feedforward, meta-NML, and naive CNML classifiers. Meta-NML provides anywhere between a 1600x and 2300x speedup compared to naive CNML, which is crucial to making our NML-based reward classifier scheme feasible on RL problems.

| | Feedforward | Meta-NML | Naive CNML |
|---|---|---|---|
| **Mazes (zigzag, spiral)** | 23.50s | 39.05s | 4hr 13min 34s |
| **Sawyer 2D Pusher** | 24.91s | 43.81 | 5hr 44min 25s |
| **Sawyer Door** | 19.77s | 38.52s | 5hr 45min 00s |
| **Sawyer 3D Pick** | 20.24s | 40.73s | 5hr 45min 00s |
| **Ant Locomotion** | 37.15s | 73.72s | 4hr 47min 40s |
| **Dexterous Hand Manipulation** | 48.37s | 69.97s | 4hr 53min 00s |

*Table 2.* Runtimes for completing a single epoch of RL according to Algorithm 2. We collect 1000 samples in the environment with the current policy for each epoch of training. The naive CNML runtimes are extrapolated based on the per-input runtime in the previous table, while the feedforward and meta-NML runtimes are averaged over 10 actual epochs of RL. These times indicate that naive CNML would be computationally infeasible to run in an RL algorithm, whereas meta-NML is able to achieve performance much closer to that of an ordinary feedforward classifier and make learning possible.

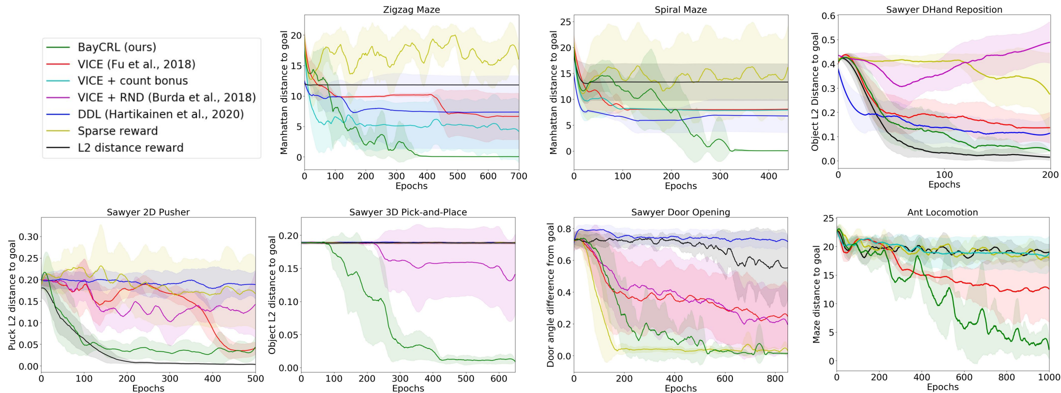


*Figure 15.* Performance of MURAL compared to other algorithms according to ground truth distance metrics. We note that while other algorithms seem to be making progress according to these distances, they are often actually getting stuck in local minima, as indicated by the success rates in Figure 5 and the visitation plots in Figure 8.

environment, the CNML classifier would still reduce to providing an exploration-centric reward bonus, as indicated by Theorem 1, ensuring reasonable worst-case performance.

To demonstrate this, we evaluate MURAL on a variant of the Zigzag Maze task where states are first discretized to a $16 \times 16$ grid, then "shuffled" so that the $xy$ representation of a state does not correspond to its true coordinates and the states are not correlated dynamically. MURAL manages to solve the task, while a standard classifier method (VICE) does not. Still, MURAL is more effective in the original state space where generalization is possible, suggesting that both the exploration and reward shaping abilities of the CNML classifier are crucial to its overall performance.

### A.5.2. Finding "Hidden" Rewards Not Indicated by Success Examples

The intended setup for MURAL (and classifier-based RL algorithms in general) is to provide a set of success examples to learn from, thus removing the need for a manually specified reward function. However, here we instead con-

| Environment | Distance Metric Used | Success Threshold |
|---|---|---|
| Zigzag Maze | Maze distance to goal | 0.5 |
| Spiral Maze | Maze distance to goal | 0.5 |
| Sawyer 2D Pusher | Puck L2 distance to goal | 0.05 |
| Sawyer Door Opening | Angle difference to goal (radians) | 0.035 |
| Sawyer 3D Pick-and-Place | Ball L2 distance to goal | 0.06 |
| Ant Locomotion | Maze distance to goal | 5 |
| Dexterous Manipulation | Object L2 distance to goal | 0.06 |

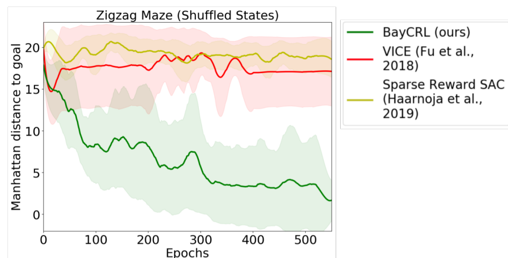


*Figure 16.* Comparison of MURAL, VICE, and SAC with sparse rewards on a discrete, randomized variant of the Zigzag Maze task. MURAL is still able to solve the task on a majority of runs due to its connection to a count-based exploration bonus, whereas ordinary classifier methods (i.e. VICE) experience significantly degraded performance in the absence of any generalization across states.

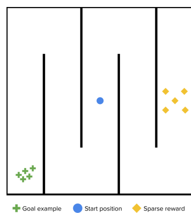

*Figure 17.* Visualization of the Double-Sided Maze environment. Only the goal examples in the bottom left corner are provided to the algorithm.

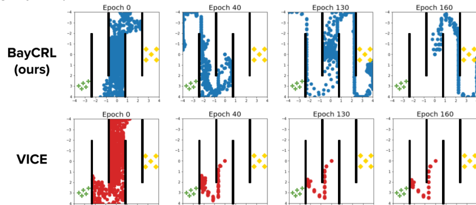


*Figure 18.* Plot of visitations for MURAL vs. VICE on the double-sided maze task. MURAL is initially guided towards the provided goals in the bottom left corner as expected, but continues to explore in both directions, thus allowing it to find the hidden sparse rewards as well. Once this happens, it focuses on the right side of the maze instead because those rewards are easier to reach. In contrast, VICE moves only towards the (incomplete) set of provided goals on the left, ignoring the right half of the maze entirely and quickly getting stuck in a local optima.

sider the case where a ground truth reward function exists which we do not fully know, and can only query through interaction with the environment. In this case, because the human expert has limited knowledge, the provided success examples may not cover all regions of the state space with high reward.

An additional advantage of MURAL is that it is still capable of finding these ”unspecified” goals because of its built-in exploration behavior, whereas other classifier methods would operate solely based on the goal examples provided. To see this, we evaluate our algorithm on a two-sided variant of the Zigzag Maze with multiple goals, visualized in Figure 17 to the right. The agent starts in the middle and is provided with 5 goal examples on the far left side of the maze; unknown to it, the right side contains 5 sparse reward regions which are actually closer from its initial position.

As shown in Figures 14 and 18, MURAL manages to find the sparse rewards while other methods do not. MURAL, although initially guided towards the provided goal examples on the left, continues to explore in both directions and eventually finds the ”hidden” rewards on the right. Meanwhile, VICE focuses solely on the provided goals, and gets stuck in a local optima near the bottom left corner.

### A.6. Hyperparameter and Implementation Details

We describe the hyperparameter choices and implementation details for our experiments here. We first list the general hyperparameters that were shared across runs, then provide tables of additional hyperparameters we tuned over for each domain and algorithm.

**Goal Examples:** For the classifier-based methods in our experiments (VICE and MURAL), we provide 150 goal examples for each environment at the start of training. These are used as the pool of positive examples when training the success classifier.

**DDL Reward:** We use the version of DDL proposed in (Hartikainen et al., 2019) where we provide the algorithm with the ground truth goal state $\mathbf{g}$, then run SAC with a reward function of $r(\mathbf{s}) = -d^{\pi}(\mathbf{s}, \mathbf{g})$, where $d^{\pi}$ is the learned dynamical distance function.

#### A.6.1. GENERAL HYPERPARAMETERS

| **SAC** | |
|---|---|
| Learning Rate | $3 \times 10^{-4}$ |
| Discount Factor $\gamma$ | $0.99$ |
| Policy Type | Gaussian |
| Policy Hidden Sizes | $(512, 512)$ |
| Policy Hidden Activation | ReLU |
| RL Batch Size | $1024$ |
| Reward Scaling | $1$ |
| Replay Buffer Size | $500,000$ |
| Q Hidden Sizes | $(512, 512)$ |
| Q Hidden Activation | ReLU |
| Q Weight Decay | $0$ |
| Q Learning Rate | $3 \times 10^{-4}$ |
| Target Network $\tau$ | $5 \times 10^{-3}$ |
| **MURAL** | |
| Adaptation batch size | 64 |
| Meta-training tasks per epoch | 128 |
| Meta-test set size | 2048 |
| **VICE** | |
| Classifier Learning Rate | $1 \times 10^{-4}$ |
| Classifier Batch Size | 128 |
| Classifier Optimizer | Adam |
| RL Algorithm | SAC |
| **RND** | |
| Hidden Layer Sizes | $(256, 256)$ |
| Output Units | 512 |

*Table 3.* General hyperparameters used across all domains.

#### A.6.2. ZIGZAG MAZE HYPERPARAMETERS

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [(512, 512), **(2048, 2048)**] |
| $\lambda_{dist}$ | $[\mathbf{0.5}, 1]$ |
| $k_{query}$ | **1** |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Weight Decay $\lambda$ | $[0, \mathbf{5 \times 10^{-3}}]$ |
| **VICE+Count Bonus** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Classifier reward scale | $[\mathbf{0.25}, 0.5, 1]$ |
| Weight Decay $\lambda$ | $[\mathbf{0}, 5 \times 10^{-3}]$ |
| **DDL** | |
| $N_d$ | $[\mathbf{2}, 4]$ |
| Training frequency (every $n$ steps) | $[16, \mathbf{64}]$ |

*Table 4.* Hyperparameters we tuned for the Zigzag Maze task. Bolded values are what we use for the final runs in Section 6.

#### A.6.3. SPIRAL MAZE HYPERPARAMETERS

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [(512, 512), **(2048, 2048)**] |
| $\lambda_{dist}$ | $[\mathbf{0.5}, 1]$ |
| $k_{query}$ | **1** |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Weight Decay $\lambda$ | $[0, \mathbf{5 \times 10^{-3}}]$ |
| **VICE+Count Bonus** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Classifier reward scale | $[\mathbf{0.25}, 0.5, 1]$ |
| Weight Decay $\lambda$ | $[\mathbf{0}, 5 \times 10^{-3}]$ |
| **DDL** | |
| $N_d$ | $[\mathbf{2}, 4]$ |
| Training frequency (every $n$ steps) | $[16, \mathbf{64}]$ |

*Table 5.* Hyperparameters we tuned for the Spiral Maze task. Bolded values are what we use for the final runs in Section 6.

### A.6.4. Ant Locomotion Hyperparameters

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [(512, 512), **(2048, 2048)**] |
| $\lambda_{dist}$ | $[0.5, \mathbf{1}, 1.5, 2]$ |
| $k_{query}$ | **1** |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Weight Decay $\lambda$ | $[0, \mathbf{5 \times 10^{-3}}]$ |
| **VICE+Count Bonus** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Classifier reward scale | $[\mathbf{0.25}, 0.5, 1]$ |
| Weight Decay $\lambda$ | $\mathbf{5 \times 10^{-3}}$ |
| **DDL** | |
| $N_d$ | $[2, \mathbf{4}]$ |
| Training frequency (every $n$ steps) | $[\mathbf{16}, 64]$ |

*Table 6.* Hyperparameters we tuned for the Ant Locomotion task. Bolded values are what we use for the final runs in Section 6.

### A.6.5. Sawyer Push Hyperparameters

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [(512, 512), **(2048, 2048)**] |
| $\lambda_{dist}$ | $[0.2, \mathbf{0.6}, 1]$ |
| $k_{query}$ | **1** |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, 2, \mathbf{10}]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Weight Decay $\lambda$ | $[\mathbf{0}, 5 \times 10^{-3}]$ |
| **VICE + RND** | |
| $n_{\text{VICE}}$ | $[1, 2, \mathbf{10}]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| RND reward scale | $[\mathbf{1}, 5, 10]$ |
| **DDL** | |
| $N_d$ | $[\mathbf{4}, 10]$ |
| Training frequency (every $n$ steps) | $[\mathbf{16}, 64]$ |

*Table 7.* Hyperparameters we tuned for the Sawyer Push task. Bolded values are what we use for the final runs in Section 6.

### A.6.6. Sawyer Pick-and-Place Hyperparameters

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [**(512, 512)**, (2048, 2048)] |
| $\lambda_{dist}$ | $[0.2, \mathbf{0.6}, 1]$ |
| $k_{query}$ | **1** |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| Weight Decay $\lambda$ | $[\mathbf{0}, 5 \times 10^{-3}]$ |
| **VICE + RND** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[0, \mathbf{1}]$ |
| RND reward scale | $[\mathbf{1}, 5, 10]$ |
| **DDL** | |
| $N_d$ | $[4, \mathbf{10}]$ |
| Training frequency (every $n$ steps) | $[\mathbf{16}, 64]$ |

*Table 8.* Hyperparameters we tuned for the Sawyer Pick-and-Place task. Bolded values are what we use for the final runs in Section 6.

### A.6.7. Sawyer Door Opening Hyperparameters

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [**(512, 512)**, (2048, 2048)] |
| $\lambda_{dist}$ | $[0.05, 0.1, \mathbf{0.25}]$ |
| $k_{query}$ | [1, **2**] |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{5}, 10]$ |
| Mixup $\alpha$ | $[\mathbf{0}, 1]$ |
| Weight Decay $\lambda$ | $[\mathbf{0}, 5 \times 10^{-3}]$ |
| **VICE + RND** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{5}, 10]$ |
| Mixup $\alpha$ | $[\mathbf{0}, 1]$ |
| RND reward scale | $[1, \mathbf{5}, 10]$ |
| **DDL** | |
| $N_d$ | $[4, \mathbf{10}]$ |
| Training frequency (every $n$ steps) | $[\mathbf{16}, 64]$ |

*Table 9.* Hyperparameters we tuned for the Sawyer Door Opening task. Bolded values are what we use for the final runs in Section 6.

### A.6.8. DEXTEROUS HAND REPOSITIONING HYPERPARAMETERS

| **MURAL** | |
|---|---|
| Classifier Hidden Layers | [(512, 512), **(2048, 2048)**] |
| $\lambda_{dist}$ | $[0.2, \mathbf{0.5}, 1]$ |
| $k_{query}$ | **1** |
| **VICE** | |
| $n_{\text{VICE}}$ | $[1, 2, \mathbf{10}]$ |
| Mixup $\alpha$ | $[\mathbf{0}, 1]$ |
| Weight Decay $\lambda$ | $[0, \mathbf{5 \times 10^{-3}}]$ |
| **VICE + RND** | |
| $n_{\text{VICE}}$ | $[1, \mathbf{2}, 10]$ |
| Mixup $\alpha$ | $[\mathbf{0}, 1]$ |
| RND reward scale | $[\mathbf{1}, 5, 10]$ |
| **DDL** | |
| $N_d$ | $[\mathbf{4}, 10]$ |
| Training frequency (every $n$ steps) | $[\mathbf{16}, 64]$ |

*Table 10.* Hyperparameters we tuned for the Dexterous Hand Repositioning task. Bolded values are what we use for the final runs in Section 6.