Appendix

A. MACAW Auxiliary Loss and Update Expressiveness

Finn and Levine (2018) lay out conditions under which the MAML update procedure is universal, in the sense that it can approximate any function $f(\mathbf{x}, \mathbf{y}, \mathbf{x}^*)$ arbitrarily well (given enough capacity), where $\mathbf{x}$ and $\mathbf{y}$ are the support set inputs and labels, respectively, and $\mathbf{x}^*$ is the test input. Universality in this sense is an attractive property because it implies that the update is expressive enough to approximate any update procedure; a method that does *not* possess the universality property might be limited in its asymptotic post-adaptation performance because it cannot express (or closely approximate) the true optimal update procedure. In order for the MAML update procedure to be universal, several requirements of the network architecture, hyperparameters, and loss function must be satisfied. Most of these are not method-specific in that they stipulate minimum network depth, activation functions, and non-zero learning rate for any neural network. However, the condition placed on the loss function require more careful treatment. The requirement is described in Definition 1.

Definition 1. *A loss function is 'universal' if the gradient of the loss with respect to the prediction(s) is an invertible function of the label(s) used to compute the loss.*

We note that Definition 1 is a necessary but not sufficient condition for an update procedure to be universal (see other conditions above and Finn and Levine (2018)). For the AWR loss function (copied below from Equation 1 with minor changes), the labels are the ground truth action $\mathbf{a}$ and the corresponding advantage $\mathcal{R}(\mathbf{s}, \mathbf{a}) - V_{\phi'_i}(\mathbf{s})$.

$$\mathcal{L}^{\text{AWR}}(\mathbf{s}, \mathbf{a}, \theta, \phi'_i) = \\ -\log \pi_\theta(\mathbf{a}|\mathbf{s}) \exp\left(\frac{1}{T}\left(\mathcal{R}(\mathbf{s}, \mathbf{a}) - V_{\phi'_i}(\mathbf{s})\right)\right) \quad (7)$$

For simplicity and without loss of generality (see Finn and Levine (2018), Sections 4 & 5), we will consider the loss for only a single sample, rather than averaged over a batch.

In the remainder of this section, we first state in Theorem 1 that the standard AWR policy loss function does not satisfy the condition for universality described in Definition 1. The proof is by a simple counterexample. Next, we state in Theorem 2 that the MACAW auxiliary loss does satisfy the universality condition, enabling a universal update procedure given the other generic universality conditions are satisfied (note that the MACAW value function loss satisfies the condition in Definition 1 because it uses L2 regression (Finn and Levine, 2018)).

A.1. Non-Universality of Standard AWR Policy Loss Function

Intuitively, the AWR gradient does not satisfy the invertibility condition because it does not distinguish between a small error in the predicted action that has a large corresponding advantage weight and a large error in the predicted action (in the same direction) that has a small corresponding advantage weight. The following theorem formalizes this statement.

Theorem 1. *The AWR loss function $\mathcal{L}^{AWR}$ is not universal according to Definition 1.*

The proof is by counterexample; we will show that there exist different sets of labels $\{\mathbf{a}_1, A_1(\mathbf{s}, \mathbf{a}_1)\}$ and $\{\mathbf{a}_2, A_2(\mathbf{s}, \mathbf{a}_1)\}$ that produce the same gradient for some output of the model. First, rewriting Equation 7 with $A(\mathbf{s}, \mathbf{a}) = \left(\mathcal{R}(\mathbf{s}, \mathbf{a}) - V_{\phi'_i}(\mathbf{s})\right)$, we have

$$\mathcal{L}^{\text{AWR}}(\mathbf{s}, \mathbf{a}, \theta) = -\log \pi_\theta(\mathbf{a}|\mathbf{s}) \exp\left(\frac{A(\mathbf{s}, \mathbf{a})}{T}\right)$$

Because our policy is parameterized as a Gaussian with fixed diagonal covariance $\sigma^2 I$, we can again rewrite this loss as

$$\mathcal{L}^{\text{AWR}}(\mathbf{s}, \mathbf{a}, \hat{\mathbf{a}}_\mu) = \\ \left(\log \frac{1}{(2\pi\sigma^2)^{\frac{k}{2}}} + \frac{||\mathbf{a} - \hat{\mathbf{a}}_\mu||^2}{2\sigma^2}\right) \exp\left(\frac{A(\mathbf{s}, \mathbf{a})}{T}\right) \quad (8)$$

where $\hat{\mathbf{a}}_\mu$ is the mean of the Gaussian output by the policy and $k = \dim(\mathbf{a})$. For the purpose of the simplicity of the counterexample, we assume the policy output $\hat{\mathbf{a}}_\mu$ is $\mathbf{0}$. The gradient of this loss with respect to the policy output is

$$\nabla_{\hat{\mathbf{a}}_\mu} \mathcal{L}^{\text{AWR}}(\mathbf{s}, \mathbf{a}, \mathbf{0}) = -\frac{1}{\sigma^2} \exp\left(\frac{A(\mathbf{s}, \mathbf{a})}{T}\right) \mathbf{a}$$

To demonstrate that the gradient operator applied to this loss function is not invertible, we pick two distinct label values and show that they give the same gradient. We pick $\mathbf{a}_1 = [1, ..., 1]^T$, $A_1(\mathbf{s}, \mathbf{a}_1) = T$ and $\mathbf{a}_2 = [0.1, ..., 0.1]^T$, $A_2(\mathbf{s}, \mathbf{a}_2) = \log(10)T$. Inserting these values into Equation A.1, this gives gradients $g_1 = \frac{-e}{\sigma^2}[1, ..., 1]^T$ and $g_2 = \frac{-10e}{\sigma^2}[0.1, ..., 0.1]^T = \frac{-e}{\sigma^2}[1, ..., 1]^T = g_1$. Thus the gradient of the standard AWR loss does not possess sufficient information to recover the labels uniquely and using this loss for policy adaptation does not produce a universal policy update procedure. Next, we show how the auxiliary loss used in MACAW alleviates this problem.

A.2. Universality of the MACAW Policy Adaptation Loss Function

In this section, we show that by adding an additional term to the AWR loss function, we acquire a loss that satisfies

the condition stated in Definition 1, which we state in Theorem 2. Intuitively, the additional loss term allows the gradient to distinguish between the cases that were problematic for the AWR loss (large action error and small advantage weight vs small action error and large advantage weight).

Theorem 2. *The MACAW policy loss function $\mathcal{L}_\pi$ is universal according to Definition 1.*

The MACAW policy adaptation loss (given in Equation 3) is the sum of the AWR loss and an auxiliary advantage regression loss (the following is adapted from Equation 8):

$$\mathcal{L}_\pi(\mathbf{s}, \mathbf{a}, \hat{\mathbf{a}}_\mu, \hat{A}) = \left(\log \frac{1}{(2\pi\sigma^2)^{\frac{k}{2}}} + \frac{||\mathbf{a} - \hat{\mathbf{a}}_\mu||^2}{2\sigma^2}\right) \exp\left(\frac{A(\mathbf{s}, \mathbf{a})}{T}\right) + \lambda(A(\mathbf{s}, \mathbf{a}) - \hat{A})^2$$

where $\hat{A}$ is the predicted advantage output from the policy advantage head and λ is the advantage regression coefficient. The gradient of this loss with respect to the predicted advantage $\hat{A}$ is

$$g_{\text{ADV}} = \nabla_{\hat{A}} \mathcal{L}_\pi(\mathbf{s}, \mathbf{a}, \hat{\mathbf{a}}_\mu, \hat{A}) = 2\lambda(\hat{A} - A(\mathbf{s}, \mathbf{a})) \tag{9}$$

and the gradient of the loss with respect to $\hat{\mathbf{a}}_\mu$ is

$$\mathbf{g}_{\text{AWR}} = \nabla_{\hat{\mathbf{a}}_\mu} \mathcal{L}_\pi(\mathbf{s}, \mathbf{a}, \hat{\mathbf{a}}_\mu, \hat{A}) = \frac{1}{\sigma^2} \exp\left(\frac{A(\mathbf{s}, \mathbf{a})}{T}\right) (\hat{\mathbf{a}}_\mu - \mathbf{a}) \tag{10}$$

We write the combined gradient as $\mathbf{g} = \begin{bmatrix} g_{\text{ADV}} \\ \mathbf{g}_{\text{AWR}} \end{bmatrix}$. In order to provide a universal update procedure, we must be able to recover both the action label $\mathbf{a}$ and the advantage label $A(\mathbf{s}, \mathbf{a})$ from $\mathbf{g}$. First, because g_{ADV} is an invertible function of $A(\mathbf{s}, \mathbf{a})$, we can directly extract the advantage label by re-arranging Equation 9:

$$A(\mathbf{s}, \mathbf{a}) = \frac{g_{\text{ADV}} - 2\lambda\hat{A}}{-2\lambda}$$

Similarly, $\mathbf{g}_{\text{AWR}}$ is an invertible function of $\mathbf{a}$, so we can then extract the action label by re-arranging Equation 10:

$$\mathbf{a} = \frac{\mathbf{g}_{\text{AWR}} - \frac{1}{\sigma^2} \exp\left(\frac{A(\mathbf{s},\mathbf{a})}{T}\right) \hat{\mathbf{a}}_\mu}{-\frac{1}{\sigma^2} \exp\left(\frac{A(\mathbf{s},\mathbf{a})}{T}\right)} \tag{11}$$

Because we can compute $A(\mathbf{s}, \mathbf{a})$ from g_{ADV}, there are no unknowns in the RHS of Equation 11 and we can compute $\mathbf{a}$ (here, σ, λ, and T are known constants); it is thus the additional information provided by g_{ADV} that resolves the ambiguity that is problematic for the standard AWR policy loss gradient. We have now shown that both the action label and advantage label used in the MACAW policy adaptation loss are recoverable from its gradient, implying that the update procedure is universal under the conditions given by Finn and Levine (2018), which concludes the proof.

B. Weight Transform Layers

Here, we describe in detail the 'weight transformation' layer that augments the expressiveness of the MAML update in MACAW. First, we start with the observation in past work (Finn et al., 2017b) that adding a 'bias transformation' to each layer improves the expressiveness of the MAML update. To understand the bias transform, we compare with a typical fully-connected layer, which has the forward pass

$$\mathbf{y} = \sigma\left(W\mathbf{x} + \mathbf{b}\right)$$

where $\mathbf{x}$ is the previous layer's activations, $\mathbf{b}$ is the bias vector, W is the weight matrix, and $\mathbf{y}$ is this layer's activations. For a bias transformation layer, the forward pass is

$$\mathbf{y} = \sigma\left(W\mathbf{x} + W^b\mathbf{z}\right)$$

where $\mathbf{z}$ and W^b are learnable parameters of the bias transformation. During adaptation, either only the vector $\mathbf{z}$ or both the vector $\mathbf{z}$ and the bias matrix W^b are adapted. The vector $W^b\mathbf{z}$ has the same dimensionality as the bias in the previous equation. This formulation does not increase the expressive power of the forward pass of the layer, but it does allow for a more expressive update of the 'bias vector' $W^b\mathbf{z}$ (in the case of dim($\mathbf{z}$) = dim($\mathbf{b}$) and $W^b = I$, we recover the standard fully-connected layer).

For a weight transformation layer (used in MACAW), we extend the idea of computing the bias from a latent vector to the weight matrix itself. We now present the forward pass for a weight transformation layer layer with d input and d output dimensions and latent dimension c. First, we compute $w = W^{\text{wt}}\mathbf{z}$, where $W^{\text{wt}} \in \mathbb{R}^{(d^2+d)\times c}$. The first d^2 components of w are reshaped into the $d \times d$ weight matrix of the layer W^*, and the last d components are used as the bias vector b^*. The forward pass is then the same as a regular fully-connected layer, but using the computed matrix and bias W^* and b^* instead of a fixed matrix and bias vector; that is $y = \sigma(W^*\mathbf{x} + \mathbf{b}^*)$. During adaptation, both the latent vector $\mathbf{z}$ and the transform matrix W^{wt} are adapted. We note that adapting $\mathbf{z}$ enables the post-adaptation weight matrix used in the forward pass, $W^{*'}$, to differ from the pre-adaptation weight matrix W^* by a matrix of rank up to the dimension of $\mathbf{z}$, whereas gradient descent with normal layers makes rank-1 updates to weight matrices. We hypothesize it is this added expressivity that makes the weight transform layer effective. A comparison of MACAW with and without weight transformation layers can be found in Figures 5-center and 7.

C. Additional Experiments and Ablations

C.1. Weight Transform Ablation Study

In addition to the results shown in Figure 5 (center), we include an ablation of the weight transform here for all tasks. Figure 6 shows these results. We find that across

environments, the weight transform plays a significant role in increasing training speed, stability, and even final performance. On the relatively simple cheetah direction benchmark, it does not affect the quality of the final meta-trained agent, but it does improve the speed and stability of training. On the other three (more difficult) tasks, we see a much more noticeable affect in terms of both training stability as well as final performance.

Additionally, we investigate the effect of the weight transform in a few-shot image classification setting. We use the 20-way 1-shot Omniglot digit classification setup (Lake et al., 2015), specifically the train/val split used by (Vinyals et al., 2016) as implemented by (Deleu et al., 2019). We compare three MLP models, all with 4 hidden layers:

1. An MLP **with weight transform layers** of 128 hidden units and a latent layer dimension of 32 (4,866,048 parameters; Weight Transform in Figure 7).
2. An MLP **without weight transform layers**, with 128 hidden units (152,596 parameters; No WT-Equal Width in Figure 7)
3. An MLP **without weight transform layers**, with 1150 hidden units (4,896,720 parameters; No WT-Equal Params in Figure 7)

We find that the model with weight transform layers shows the best combination of fast convergence and good asymptotic performance compared with baselines with regular fully-connected layers. *No WT-Equal Width* has the same number of hidden units as the weight transform model (128), which means the model has fewer parameters in total (because the weight transform layers include a larger weight matrix). The *No WT-Equal Params* baseline uses wider hidden layers to equalize the number of parameters in the entire model with the Weight Transform model. Somewhat surprisingly, the smaller baseline model (Equal Width) outperforms the larger baseline model (Equal Params).

When using MAML-style meta-learners, it is important to consider that adding parameters to the model affects the expressiveness of *both* the forward computation of the model and the updates computable with a finite number of steps of gradient descent.

Generally, increasing the number of parameters in the model should improve the model's ability to fit the training set (because the inner loop of MAML is more expressive), which we observe here. Increasing the expressiveness of the inner loop of MAML can also speed convergence, which we also observe in Figure 7. However, by simply adding neurons to a typical MLP, the post-adaptation model tends to overfit the training set more, as we see in Figure 7. On the other hand, adding parameters through weight transformation layers **increases expressiveness of the adaptation step by enabling weight updates with rank greater than 1 without changing the expressiveness of the forward computation of the model**.

C.2. Online Fine-Tuning for Out-of-Distribution Test Tasks

In some cases, it may be necessary or desirable to perform **online** fine-tuning after the initial offline adaptation step. This is the fully offline meta-RL problem with online fine-tuning described in Section 3, where an algorithm is given a small amount of initial adaptation data from the test task, just as in the fully offline setting, and then is able to interact with the environment to collect additional training data and perform on-policy updates. This ability to continually improve with additional training after the initial offline adaptation step is what makes a consistent meta-reinforcement learner advantageous.

This hybrid setting (offline training with additional online fine-tuning) is known to be extremely challenging in traditional reinforcement learning. These difficulties are clearly documented by recent work (Nair et al., 2020). In short, this setting is challenging in traditional RL because while offline pre-training might produce a policy that performs well, online fine-tuning often leads to a significant drop in initial performance, which can take a very long time to recover from (see (Nair et al., 2020)). In many cases, online fine-tuning can take a very long time to recover the performance of the offline-only policy, if it does so at all. In offline meta-RL, we have a similar challenge; an offline meta-RL algorithm must not only meta-train for good performance on a single batch of offline test data, but it must also learn a set of parameters that enables fine-tuning to make productive updates to its policy and/or value function without completely destroying the meta-learned knowledge about the task distribution.

In this section, we use a hybrid setting as described above to evaluate not only MACAW's consistency (its ability to continue to improve after an initial offline adaptation step), but its ability to continue to improve **even when the test task distribution differs from the train distribution**. Because significant distribution shift means that some train tasks are irrelevant, or even detrimental to test performance, this setting is very difficult. In order to make a meaningful comparison, we compare with an "Offline PEARL + fine-tuning" (Offline PEARL+FT) algorithm, which is also technically consistent (because it essentially performs the SAC algorithm on the test task after the initial task inference step). However, we hypothesize that MACAW will have an advantage over this Offline PEARL+FT algorithm because while both algorithms are consistent, MACAW explicitly trains for good fine-tunability with gradient descent, unlike task inference-based meta-RL algorithms.

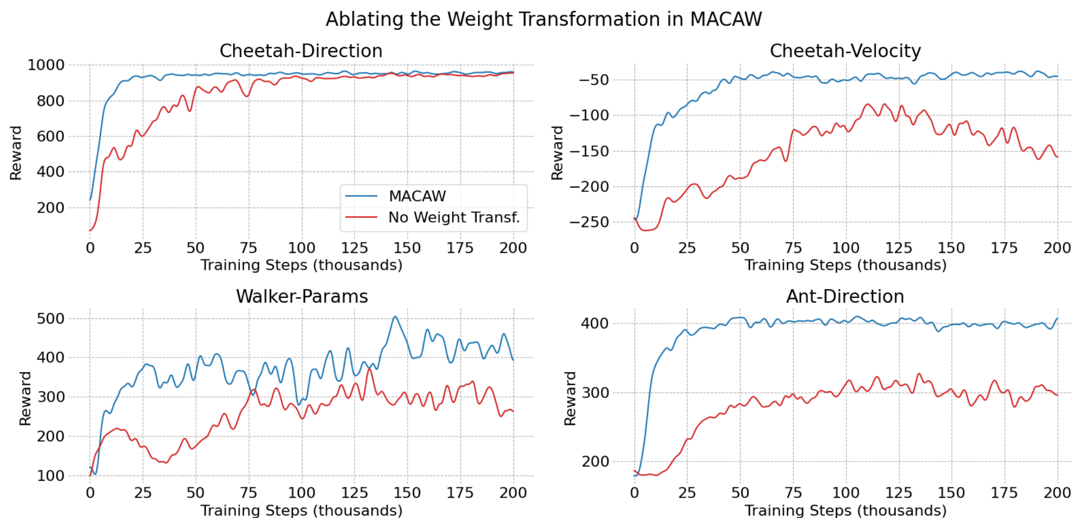

Figure 6: Ablating the weight transformation in MACAW on the MuJoCo benchmark environments. All networks have the same number of hidden units. Although MACAW is able to learn with regular fully-connected layers, the weight transformation significantly improves performance on all tasks that require adaptation to unseen tasks.

Additional Env. Steps	Offline PEARL+FT		MACAW	
	Reward	Improvement	Reward	Improvement
0	-553.4 (21.2)	–	-323.1 (42.9)	–
20k	-565.0 (4.7)	-11.5 (3.5)	-279.1 (16.8)	**44.0 (14.5)**
200k	-533.6 (19.8)	19.8 (3.7)	-272.0 (15.2)	**51.1 (12.7)**

Table 2: Absolute reward as well as improvement (in terms of reward) of Offline PEARL+FT and MACAW after 0, 20k, and 200k additional environment steps are gathered and used for online fine tuning. Standard errors of the mean over the 13 test tasks are reported in parentheses. Averages are taken over 10 rollouts of each policy. We find that MACAW achieves both **better out-of-distribution performance** before online training as well as **faster improvement** during online fine-tuning. Note that Offline PEARL+FT experiences an initial *drop* in average performance on the test task after 20k steps, compared with the performance of the policy conditioned only on the initial batch of offline data. A similar effect has been reported in recent work in offline RL (Nair et al., 2020).

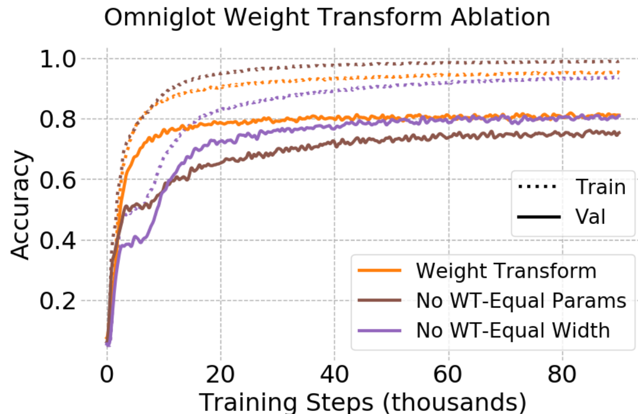

Figure 7: Faster convergence provided by the weight transform layer (orange) on Omniglot 20-way 1-shot image classification (Lake et al., 2015).

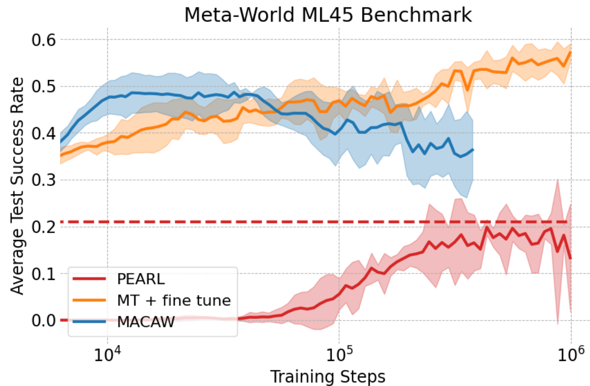

Figure 8: Average success rates of MACAW, PEARL, and MT + fine-tuning (with 20 fine-tuning steps) on the 5 test tasks the Meta-World ML45 suite of continuous control tasks. Dashed line shows final PEARL average success rate after 10m training steps.

The training procedure for Offline PEARL+FT is the same at the regular Offline PEARL training procedure. However, at test time, after receiving an initial small batch of offline data for task inference, we alternative between performing rollouts of the task-conditioned policy to collect additional data from the test task and perform gradient descent on the PEARL policy and value function objectives with this off-policy data. Similarly, for MACAW test time involves first using the small batch of *offline* test task data to take an initial gradient step on the value and policy loss functions (Eqns 2 and 3), then alternating between rolling out the adapted policy and taking more steps of gradient descent on the MACAW losses.

The specific experimental setup is as follows. We partition the individual tasks in the Cheetah-Vel problem such that training tasks correspond to target velocities in the range [0,2] and test tasks correspond to target velocities in the range [2,3]. After meta-training, for each test task, we provide the algorithm with a small batch of offline data for adaptation just as in the fully offline setting. However, we allow the algorithm to then collect and train on up to 200k additional interactions from the environment. Both algorithms alternate between sampling a single trajectory (200 environment interactions) and performing 100 steps of gradient descent on the aggregate buffer of data for the test task, which contains both the initial offline batch of data as well as all online data collected so far. We evaluate both algorithms on their performance after 20k and 200k additional interactions with the environment. The results of this experiment are reported in Table 2. We observe that MACAW achieves both **higher absolute reward** on the OOD test tasks as well as **faster relative improvement** over the offline-only adapted policy compared to the Offline PEARL+FT baseline.

C.3. Metaworld ML45 Benchmark

As an additional experiment, we test the training and generalization capabilities of MACAW on a much broader distribution of tasks, and where test tasks differ significantly from training tasks (e.g. picking up an object as opposed to opening a window or hammering a nail). Recently, Yu et al. (2019) proposed the Meta-World (Yu et al., 2019) suite of continuous control benchmark environments as a more realistic distribution of tasks for multi-task and meta-learning algorithms. This benchmark includes 45 meta-training tasks and 5 meta-testing tasks. The results of this experiment are summarized in Figure 8.

We find that all methods are able to make meaningful progress on the test tasks, with gradient-based methods (MACAW and MT + fine tune) learning much more quickly than PEARL. MACAW does achieve a quite high level of performance quite early on in training; however, it begins to overfit with further training. In the regime where periodic online evaluations are available for the purpose of early stopping, we could avoid this issue, in which case MACAW would slightly underperform the multi-task learning baseline. A possible reason for some inconsistency between the performance of each algorithm on Meta-World and the results reported in Figure 3 is the difficult scaling of the rewards in the current version of the Meta-World benchmark. Rewards can vary by 5 orders of magnitude, from negative values to values on the order of 100,000. This has been documented to adversely impact training performance even in single-task RL and increase hyperparameter sensitivity (see `https://github.com/rlworkgroup/metaworld/issues/226`). Because of the problems stemming from the current reward functions in Meta-World, the maintainers of the benchmark are updating them for the next version of the benchmark, which has not been released

as of January 2021.

D. Experimental Set-Up and Data Collection

D.1. Overview of Problem Settings

The problems of interest include:

1. **Half-Cheetah Direction** Train a simple cheetah to run in one of two direction: forward and backward. Thus, there are no held-out test tasks for this problem, making it more 'proof of concept' than benchmark.

2. **Half-Cheetah Velocity** Train a cheetah to run at a desired velocity, which fully parameterizes each task. For our main experiment, values of the task parameters are sampled from a uniform interval of 40 velocities in the range [0, 3]. A subset of 5 target velocities is sampled randomly for evaluation. For task sparsity ablation experiments, we create successively sparser sets of training tasks by repeatedly removing half of the training tasks, rounding down the number of tasks to be removed.

3. **Ant-2D Direction** Train a simulated ant with 8 articulated joints to run in a random 2D direction. For our experiments, we sample 50 random directions uniformly, holding out 5 for testing.

4. **Walker-2D Params** Train a simulated agent to move forward, where different tasks correspond to different randomized dynamics parameters rather than reward functions. For our experiments, we sample 50 random sets of dynamics parameters, holding out 5 for testing.

5. **Meta-World ML45** Train a simulated Sawyer robot to complete 45 different robotics manipulation tasks (for training). 5 additional tasks are included for testing, making 50 tasks in total. Tasks include opening a window, hammering a nail, pulling a lever, picking & placing an object. See Yu et al. (2019) for more information. Our experiments use a continuous space randomization for each task setup, unlike the experiments in (Yu et al., 2019), which sample from a fixed number of task states. This creates a much more challenging environment, as seen in the success rate curves above.

For the first 4 MuJoCo domains, each trajectory is 200 time steps (as in Rakelly et al. (2019)); for Meta-World, trajectories are 150 time steps long.

D.2. Data Collection

We adapt each task to the offline setting by restricting the data sampling procedure to sample data only from a fixed offline buffer of data. For each task, we train a separate policy from scratch, using Soft Actor-Critic (Haarnoja et al., 2018) for all tasks except Cheetah-Velocity, for which we use TD3 (Fujimoto et al., 2018) as it proved more stable across the various Cheetah-Velocity tasks. We save complete replay buffers from the entire lifetime of training for each task, which includes 5M steps for Meta-World, 2.5M steps for Cheetah-Velocity, 2.5M steps for Cheetah-Dir, 2M steps for Ant-Direction, and 1M steps for Walker-Params. We use these buffers of trajectories, one per task for each problem, to sample data in both the inner and outer loop of the algorithm during training. See Figures 9 and 10 for the learning curves of the offline policies for each train and test task.

D.3. Ablation Experiments

For the data quality experiment, we compare the post-adaptation performance when MACAW is trained with 3 different sampling regimes for the Cheetah-Vel problem setting. Bad, medium, and good data quality mean that adaptation data (during both training and evaluation) is drawn from the first, middle, and last 500 trajectories from the offline replay buffers. For the task quantity experiment, we order the tasks by the target velocity in ascending order, giving equally spaced tasks with target velocities $g_0 = 0.075$, $g_2 = 0.15$, ..., $g_{39} = 3.0$. For the 20 task experiment, we use g_i with even i for training and odd i for testing. For the 10 task experiment, we move every other train task to the test set (e.g. tasks $i = 2, 6, 10, ...$). For the 5 task experiment, we move every other remaining train task to the test set (e.g. tasks $i = 4, 12, 20, ...$), and for the 3 task experiment, we again move every other task to the test set, so that the train set only contains tasks 0, 16, and 32. Task selection was performed this way to ensure that even in sparse task environments, the train tasks provide coverage of most of the task space.

D.4. Online Fine-tuning Experiments

For these experiments, we evaluate each model after a total of 50 or 100 additional trajectories have been collected (with a trajectory length of 200 time steps, this corresponds to 10k or 20k total environment steps). After performing the initial offline adaptation step, we initialize a buffer of online data D_{online} with the offline data D_{test} (256 experience tuples). We then alternate between collecting a trajectory of data with the current policy, adding the trajectory to D_{online}, and performing 100 gradient steps of training using the entire contents of D_{online}. To avoid overfitting to a small buffer, we skip training until 5 online trajectories have been added to D_{online}. We never remove experiences from the online replay buffer during online training. For MACAW, we use learning rates and batch sizes equal to the outer loop learning rates and batch sizes given in Table 5; for PEARL, we use the learning rates and hyperparameters in Table 3.

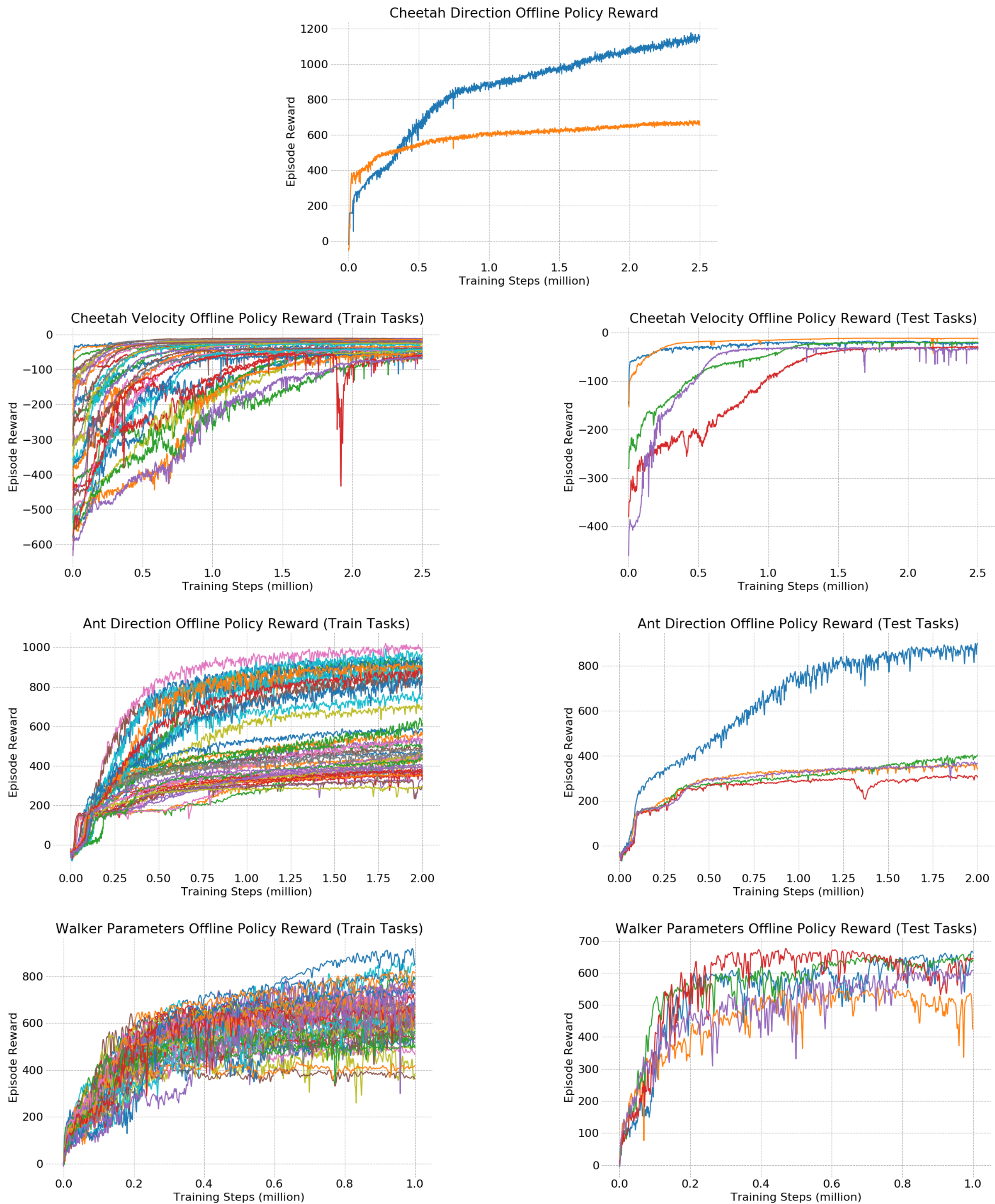

Figure 9: Learning curves for offline policies for the 4 different MuJoCo environments used in the experimental evaluations. Each curve corresponds to a policy trained on a unique task. Various levels of smoothing are applied for the purpose of easier visualization.

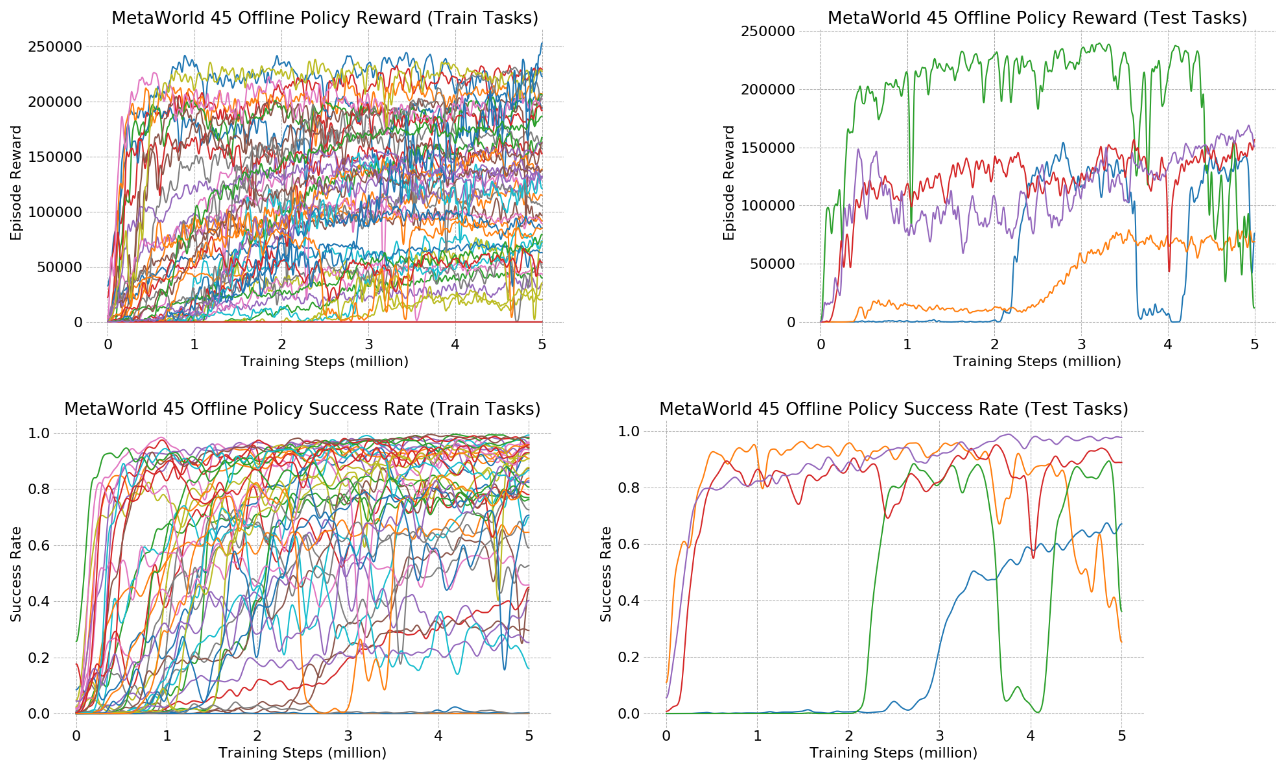

Figure 10: Learning curves and success rates for all tasks in the MetaWorld 45 benchmark. Each curve corresponds to a policy trained on a unique task. Various levels of smoothing are applied for the purpose of plotting.

E. Implementation Details and Hyperparameters

Peng et al. (2019) note several strategies used to increase the stability of their advantage-weighted regression implementation. We normalize the advantage logits in the policy update step to have zero mean and unit standard deviation, as in Peng et al. (2019). Advantage weight logits are also clipped to avoid exploding gradients and numerical overflow. To train the value function, we use simple least squares regression onto Monte Carlo returns, rather than TD(λ). Finally, our policy is parameterized by a single Gaussian with fixed variance of 0.04; our policy network thus predicts only the mean of the Gaussian distribution.

In addition to using weight transformation layers instead of regular fully-connected layers, we also learn learning rates for each layer of our network by gradient descent. To speed up training, we compute our loss using a 'task minibatch' of 5 tasks at each step of optimization, rather than using all of the training tasks. Finally, specific to the RL setting, we sample experiences in contiguous chunks from the replay buffers during train-time adaptation and uniformly (non-contiguously) from the replay buffers for outer-loop updates and test-time adaptation. For outer loop updates, we sample data selectively towards the end of the replay buffers. We use the Higher framework (Grefenstette et al., 2019) for computing higher-order derivatives for meta-learning.

E.1. Hyperparameters

Tables 3, 4, and 5 describe the hyperparameters used for each algorithm in our empirical evaluations. We performed some manual tuning of hyperparameters for all algorithms, but found that the performance was not significantly affected for environments other than Meta-World, likely due to the difficult reward scaling in the current release of Meta-World.

Parameter	Standard Configuration	Meta-World
Optimizer	Adam	–
Meta batch size	4-10	16
Batch size	256	–
Embedding batch size	100-256	750
KL penalty	0.1	–
Hidden layers	3	–
Neurons per hidden layer	300	512
Latent space size	5	8
Policy learning rate	3e-4	–
Value function learning rate	3e-4	–
Context embedding learning rate	3e-4	–
Q-Function learning rate	3e-4	–
Reward scale	5.0	–
Recurrent	False	–

Table 3: Hyperparameters used for the PEARL experiments. For the MuJoCo tasks, we generally used the same parameters as reported in (Rakelly et al., 2019), with some minor modifications. The different parameters used for the MetaWorld ML45 environment are reported above.

Parameter	Standard Configuration	Meta-World
Optimizer	Adam	–
Value learning rate	1e-4*	1e-6
Policy learning rate	1e-4	–
Value fine-tuning learning rate	1e-4	1e-6
Policy fine-tuning learning rate	1e-3	–
Train outer loop batch size	256	–
Fine-tuning batch size	256	–
Number of hidden layers	3	–
Neurons per hidden layer	100	300
Task batch size	5	–
Max advantage clip	20	–

Table 4: Hyperparameters used for the multi-task learning + fine tuning baseline. *For the Walker environment, the value learning rate was 1e-5 for stability.

Parameter	**Standard Configuration**	**Meta-World**
Optimizer	Adam	–
Auxiliary advantage loss coefficient	1e-2	1e-3
Outer value learning rate	1e-5	1e-6
Outer policy learning rate	1e-4	–
Inner policy learning rate	1e-3 (learned)	1e-2 (learned)
Inner value learning rate	1e-3 (learned)	1e-4 (learned)
Train outer loop batch size	256	–
Train adaptation batch size	256	256
Eval adaptation batch size	256	–
Number of adaptation steps	1	–
Learning rate for learnable learning rate	1e-3	–
Number of hidden layers	3	–
Neurons per hidden layer	100	300
Task batch size	5	10
Max advantage clip	20	–
AWR policy temperature	1	–

Table 5: Hyperparameters used for MACAW. The Standard Configuration is used for all experiments and all environments except for Meta-World (due to the extreme difference in magnitude of rewards in Meta-World, which has typical rewards 100-1000x larger than in the other tasks). For the Meta-World configuration, only parameters that differ from the standard configuration are listed.