A Inferred Options Policy Gradient

In our work, we use IOPG [44], a recently introduced policy gradient method for learning options that treats options as latent variables during gradient calculation. This method allows for updating all options based on their responsibilities for a given action which results in better data efficiency. The IOPG gradient can be calculated from sampled trajectories:

$$\nabla J = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla \log \pi(\boldsymbol{a}_t|\boldsymbol{s}_t) A_t \right] \overbrace{\sum_{\omega^i} p(\omega_t^i|\boldsymbol{s}_{[0:t]}, \boldsymbol{a}_{[0:t-1]}) \pi^{\omega^i}(\boldsymbol{a}_t|\boldsymbol{s}_t)}, \tag{7}$$

where $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$ can be decomposed according to Equation 7. The probability $p(\omega_t^i|\boldsymbol{s}_{[0:t]}, \boldsymbol{a}_{[0:t-1]})$ represents probability that option ω^i was active at timestep t given past actions and states and can be computed recursively:

$$p(\omega_{t+1}^j|\boldsymbol{s}_{[0:t+1]}, \boldsymbol{a}_{[0:t]}) = \frac{\sum_{\omega^i} p(\omega_t^i|\boldsymbol{s}_{[0:t]}, \boldsymbol{a}_{[0:t-1]}) \pi^{\omega^i}(\boldsymbol{a}_t|\boldsymbol{s}_t) \tilde{\pi}^\Omega(\omega_{t+1}^j|\omega_t^i, \boldsymbol{s}_{t+1})}{\sum_{\omega^k} p(\omega_t^k|\boldsymbol{s}_{[0:t]}, \boldsymbol{a}_{[0:t-1]}) \pi^{\omega^k}(\boldsymbol{a}_t|\boldsymbol{s}_t)}, \tag{8}$$

with option transition probability $\tilde{\pi}^\Omega(\omega_{t+1}^j|\omega_t^i, \boldsymbol{s}_{t+1})$ given by:

$$\tilde{\pi}^\Omega(\omega_{t+1}^j|\omega_t^i, \boldsymbol{s}_{t+1}) = \xi^{\omega^i}(\boldsymbol{s}_{t+1}) \pi^\Omega(\omega^j|\boldsymbol{s}_{t+1}) + \mathbf{1}_{\omega^j=\omega^i} \left[1 - \xi^{\omega^i}(\boldsymbol{s}_{t+1}) \right]. \tag{9}$$

B Experiment details and hyperparameters

B.1 Adjustments to Taxi environment

We made a small adjustment to the original Taxi domain proposed by Dietterich [8]. Instead of starting with any state with random probability, the agent was always initialized in one of the corners. Our motivation for this modification was the fact that high-level policy parameters were not shared across states in tabular policy. Consequently, a problem could arise if the agent would be initialized in completely different part of the state space after the inner update (due to sampling). In this case, high-level policy would be adapted only in the part that was visited before the update and would likely not lead to improvement in unexplored part of the state space. Therefore, the post-adaptation performance would strongly depend on sampled initial states.

B.2 Hyperparameters and implementation details

Hyperparameter values for FAMP and MLSH which were used in our experiments are outlined in Tables 2 and 3. We used layers without biases for FAMP in the Taxi experiment and learned the inner learning rate α_{in} for all weights and biases separately in the Ant Maze experiment. Sigmoid non-linearity was used after a final layer for terminations whereas policy over options used softmax. In the Ant Maze experiment, sub-policies outputted mean of a multivariate normal distribution. This distribution used learned diagonal covariance matrix that was shared among all states. Furthermore, we used the Adam optimizer for the update of outer parameters during the training in both experiments. During the adaptation, the SGD optimizer with learning rate α_{in} was used.

When choosing hyperparameter values for our method, we used the values from MAML [14] as a starting point and experimented with different values of α_{out} and α_{in}. The value of α_{in} is particularly important for the algorithm because the value which is too low will not allow for a sufficiently fast change of the high-level policy. Such behavior was observed in the Taxi experiment, where we increased this learning rate to a final value 10. To account for a more complex neural network in our second experiment, we chose a lower initial value and meta-learned this parameter. In both

Table 2: FAMP hyperparameters

Hyperparameter	Taxi	Ant Maze
Meta-training epochs	2000	2500-2650
Trajs per update k	10	20
Env samples N	64	48
GAE λ	0.98	0.98
Loaded DiCE λ	0	0
Return discount	0.95	0.99
Baseline	Linear	Linear
LR inner α_{in}	10	10^{-3}
Learn α_{in}	No	Yes
LR outer α_{out}	10^{-2}	10^{-3}
Options $opts$	4 (default)	3
High-level policy NN	$(72, opts)$	$(29, 64, 64, 3)$
Sub-policy NN	$(72, 6)$	$(29, 64, 64, 8)$
Terminations NN	$(72, 1)$	$(29, 64, 64, 1)$
Non-linearities	N/A	Tanh

Table 3: MLSH hyperparameters

Hyperparameter	Taxi	Ant Maze
Meta-training epochs	50	60
Trajs per update	2	2
Warmup duration	20	20
Joint update duration	30	40
GAE λ	0.98	0.98
Return discount γ	0.95	0.99
PPO clip ϵ	0.2	0.2
PPO optim epochs	10	10
PPO batchsize	64	64
Baseline	NN	NN
LR sub-policies	3×10^{-4}	3×10^{-4}
LR policy over options	0.01	0.01
Options	4	3
High-level policyNN	$(72, 4)$	$(29, 64, 64, 3)$
Sub-policy NN	$(72, 6)$	$(29, 64, 64, 8)$
Non-linearities	N/A	Tanh
Option length	4 or 10	200

experiments, we used the value of 0 for the hyperparameter of Loaded DiCE to reduce the variance of the estimator as much as possible.

In order to reproduce MLSH as closely as possible, we used the code and the hyperparameters from the original paper [17] for the Ant Maze experiment. This included running the algorithm on 120 Intel Haswell E5-2630-v3 cores, separated into 10 groups. Meta-training on a cluster took 13 hours for Taxi and 82 hours for Ant Maze experiment. Although we could not perform quantitative comparison with the original experiment because performance in these tasks was not reported, we observed that trained sub-policies were qualitatively similar to the ones shown in the videos provided by Frans et al. [17]. Since multiple sub-routines in the computation of FAMP gradient can be parallelized, we also utilized same computational cluster for meta-training. The gradients from different environments were computed on different cores independently before averaging. We used 48 cores for Ant Maze and 64 cores for Taxi. The meta-training took 7 hours for Taxi and 196 hours for Ant Maze experiment.

The single-task and multi-task baselines used the same architecture and hyperparameters as FAMP except for learning rates. The multi-task baseline was trained by averaging gradients from all 48 training environments in every update. We used learning rate 0.01 for training and 1 during the adaptation. The single-task baseline was trained with the Adam optimizer. We set the learning rate to 0.3, which was the highest value that allowed for stable training. For PPO [43] and RL^2 [10] we used the implementation of SpinningUp [1] and Garage [18] respectively. In both implementations, we

Table 4: PPO hyperparameters

Hyperparameter	Ant Maze
Trajs per update	4
GAE λ	0.98
Return discount γ	0.99
PPO clip ϵ	0.2
PPO optim epochs	80
Baseline	NN
LR baseline	10^{-3}
LR policy	3×10^{-4}
Policy NN	$(29, 64, 64, 8)$
Non-linearities	ReLU

Table 5: RL^2 hyperparameters

Hyperparameter	Ant Maze
Meta-batch size	9
Episodes per task	4
GAE λ	0.98
Return discount γ	0.99
PPO clip ϵ	0.2
PPO optim epochs	10
PPO batchsize	32
Baseline	Linear
Policy type	GRU policy
Policy hidden dims	64

used default hyperparameter values from mujoco experiments except for the values outlined in Tables 4 and 5.

C Additional Plots

C.1 Meta-training plots

In Figure 5 we provide meta-training plots from both experiments. We ran all algorithms until they stopped improving. The plot of FAMP shows average post-adaptation performance from the final episode. Due to the update schedule of MLSH which staggers some cores (for more details see [17]), reported performance is averaged over cores which may be in different adaptation phases. It is therefore not directly comparable to FAMP. However, it still provides a good indication about the final performance. Similarly, meta-training plot of RL^2 does not show the return achieved in the final episode but instead shows average return obtained over all 4 episodes. This is the objective optimized by RL^2 algorithm.

In Ant Maze experiment the variance of MLSH increases towards the end of meta-training. This can be explained by higher difference between achieved returns in some environments. As we can see from Figure 8, MLSH achieves much lower return in environments 1 and 2 when compared to other environments. Thus, depending on whether these environments are sampled less or more, the average performance over all environments can fluctuate. We can also notice that FAMP requires more environment interactions than other methods to achieve it's peak performance. This is not surprising, since both MLSH and RL^2 use PPO that uses same samples to make multiple gradient steps while FAMP relies on REINFORCE-like updates. Development of more efficient version of FAMP can thus be an interesting direction for future research.

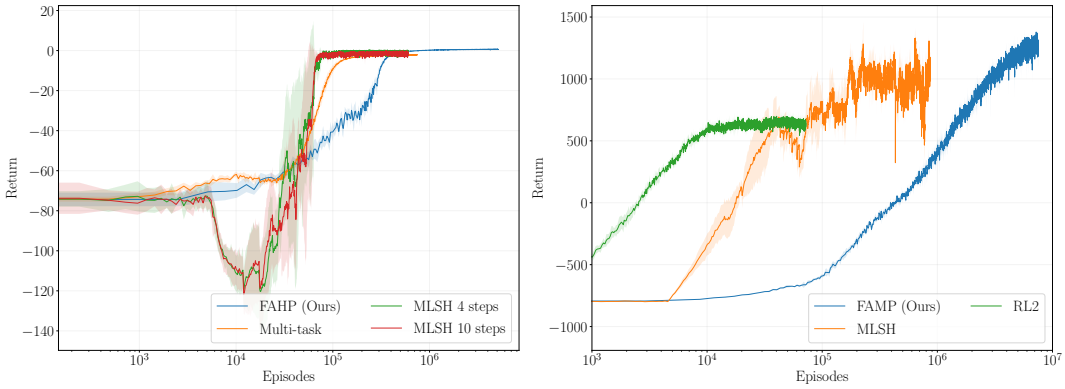

Figure 5: Meta-training plots. *Left:* Taxi experiment. *Right:* Ant Maze experiment.

C.2 Taxi - Trained Options

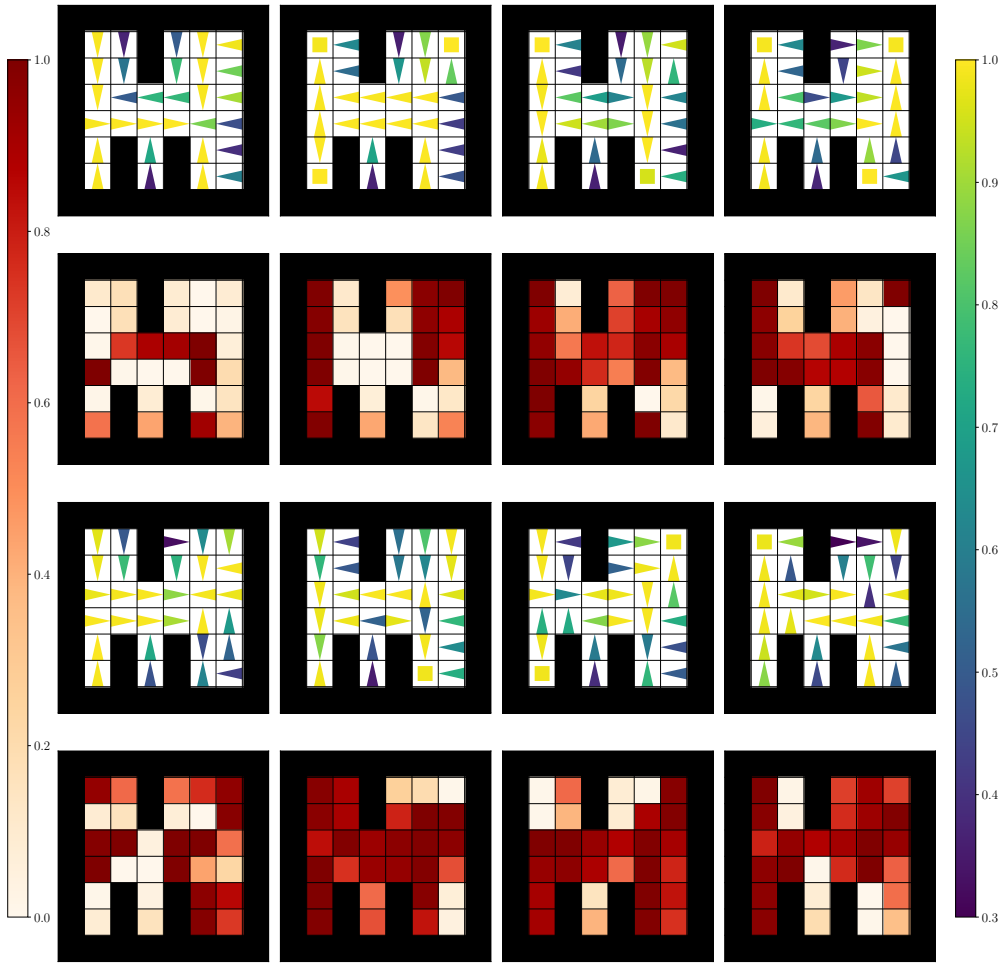

Figure 6: Example trained sub-policies and terminations on taxi tasks. Each column shows a different option. Two rows at the top show sub-policies and terminations in states without passenger and two bottom rows show states with passenger. Sub-policy plots show actions with the highest probability where we represent directional actions with arrows and pick-up/drop-off action with a square. A termination plot of each option shows the probability of termination in each state.

C.3 Performance plots

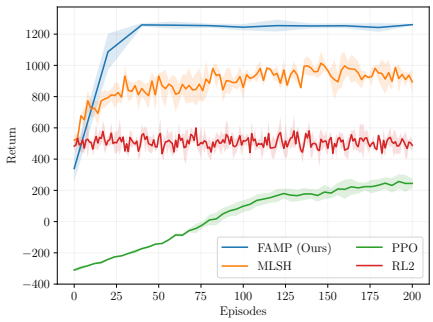

Figure 7: Average performance of algorithms on ant maze environments tasks *with resets*. Plot shows mean and standard deviation over 3 seeds

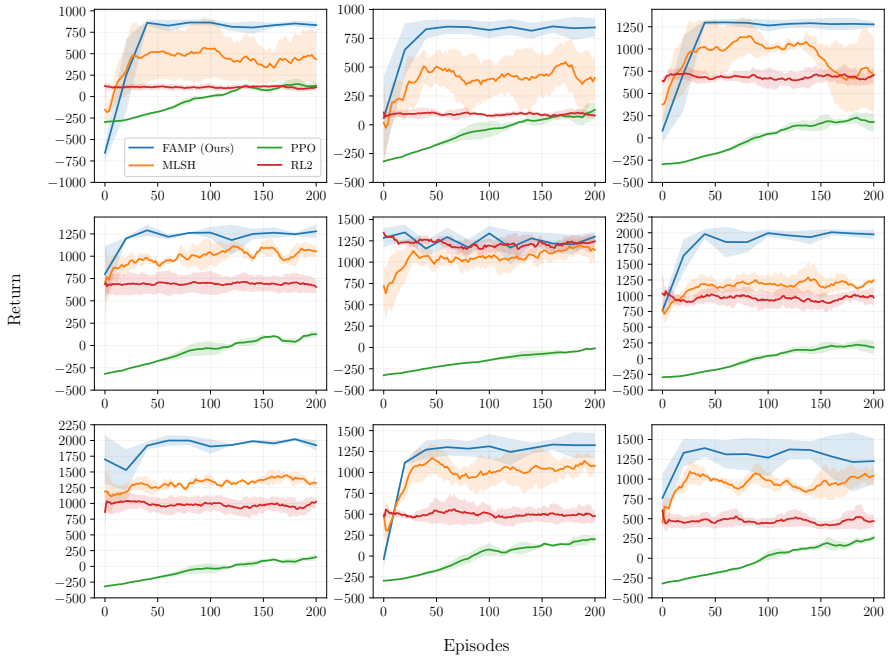

Figure 8: Performance of algorithms in each test environment of the Ant Maze experiment. Environments are ordered from left to right and top to bottom (see Figure 3). Plots show mean and standard deviation over 3 seeds. To make plots less erratic we show average performance over 20 episodes.

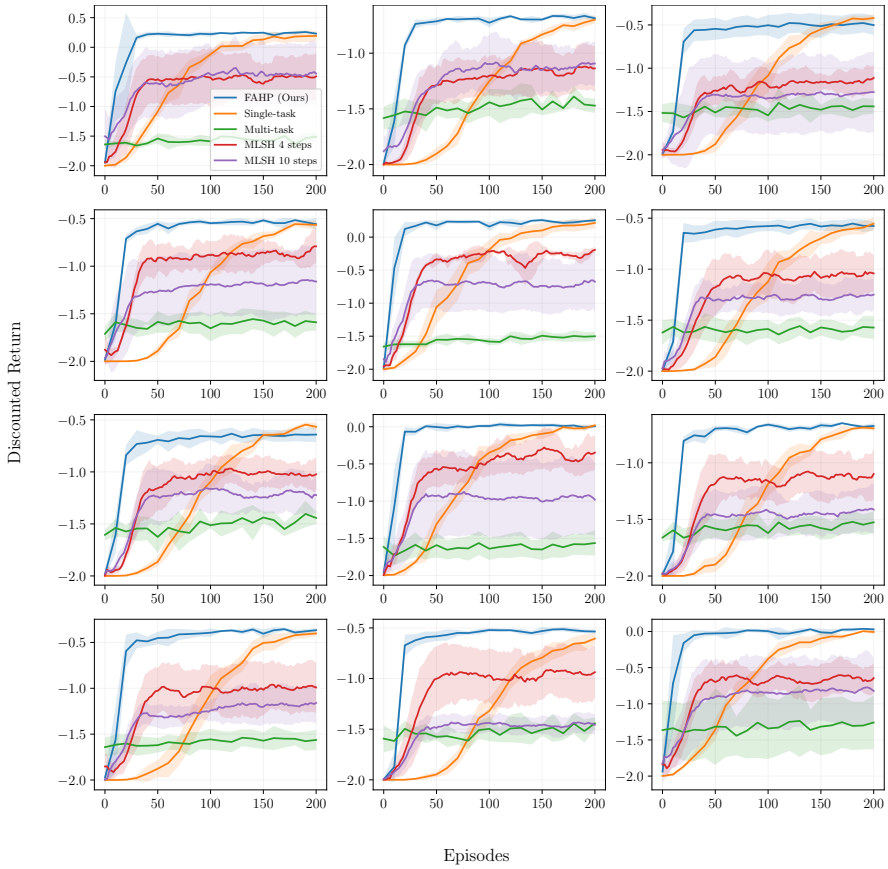

Figure 9: Performance of algorithms in each test environment of the Taxi experiment. Plot shows mean and standard deviation over 5 seeds.