Supplementary Material

A. Algorithms

In this section, we give more details about the algorithms proposed in this work. We first give a detailed version for the multi-objective policy gradient (Section 3.2), then we describe the nonlinear regression for fitting the parameters of the hyperbolic prediction model (Section 3.3), finally we analyze the prediction-guided optimization algorithm (Section 3.4) and show the time complexity of it.

A.1. Multi-Objective Policy Gradient

Given a policy $\pi_{\boldsymbol{\theta}}$ and a weight vector $\boldsymbol{\omega}(\sum_i \omega_i = 1)$, our reinforcement learning worker aims to use policy gradient to optimize for the weighted-sum reward $\mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega})$:

$$\mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega}) = \boldsymbol{\omega}^\top \mathbf{F}(\pi) = \sum_{i=1}^{m} \omega_i f_i(\pi) = \sum_{i=1}^{m} \omega_i J_i^\pi$$

The most straight forward way is converting the environment from returning a vector of rewards into a scalar weighted-sum reward, regarding it as a single-objective control problem, and solving it with any single-objective policy gradient algorithm. Most policy gradient algorithms simultaneously learn a value function $V(s)$ and a policy network $\pi(a|s)$. The value function receives the current state s and estimates the expected return (expected weighted-sum return here) under following the current policy and is used to lower the training variance.

However, with our evolutionary learning algorithm, a policy will be selected to be optimized with different weights during the whole learning process. It is inefficient to simply apply the above approach. With this naive approach, the value network trained with previous weights would be invalid for the new weight and would need to be trained from scratch.

In order to leverage the previous learning information of a policy when running reinforcement learning for it with a new weight, we improve the single-objective policy gradient algorithm by extending the value function to be vectorized, which shares a similar strategy as applied in multi-objective Q-learning (Yang et al., 2019).

Specifically, the vectorized value function $\boldsymbol{V}^\pi(s) : \mathcal{S} \to \mathbb{R}^m$ maps a state s to the vector of expected returns under following the current policy π. In this way, the value function is still valid when the optimization weight changes and can be directly used to train the policy for the new weight and quickly adapt its output to the new policy. The value function is updated in each iteration by the Bellman Equation:

$$\boldsymbol{V}^\pi(s_t) = \hat{\boldsymbol{V}}(s_t) = \sum_u \pi(u|s_t)(\boldsymbol{R}(s) + \gamma \boldsymbol{V}^\pi(s_{t+1})),$$

where $\hat{\boldsymbol{V}}(s_t)$ is the target value function, and s_{t+1} is reached from state s_t by action u.

In the implementation, the value function is updated by a squared-error loss $\|\boldsymbol{V}^\pi(s) - \hat{\boldsymbol{V}}(s)\|^2$.

To compute the policy gradient, we start from the policy gradient for each objective (we use the advantage version described in (Schulman et al., 2015)).

$$\nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}) = \mathbb{E}\left[\sum_{t=0}^{T} A_i^\pi(s_t, a_t) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right]$$

where $A_i^\pi(s_t, a_t)$ is the advantage function for i-th objective.

Then the policy gradient for $\mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega})$ is derived:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega}) &= \sum_{i=1}^{m} \omega_i \nabla_{\boldsymbol{\theta}} J_i(\boldsymbol{\theta}) \\ &= \sum_{i=1}^{m} \omega_i \mathbb{E}\left[\sum_{t=0}^{T} A_i^{\pi}(s_t, a_t)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right] \\ &= \mathbb{E}\left[\sum_{t=0}^{T} \boldsymbol{\omega}^T \boldsymbol{A}^{\pi}(s_t, a_t)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right] \\ &= \mathbb{E}\left[\sum_{t=0}^{T} A_{\boldsymbol{\omega}}^{\pi}(s_t, a_t)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t)\right]\end{aligned}$$

where $\boldsymbol{A}^{\pi}(s_t, a_t)$ is the vectorized advantage function. In our extension, the new advantage function $A_{\boldsymbol{\omega}}^{\pi}(s_t, a_t)$ is simply represented as a weighted-sum scalarization of the advantage functions for individual objectives.

Such value network and policy gradient extension can be easily applied to most policy gradient methods. In our implementation, we choose to adapt the Proximal Policy Optimization (PPO) (Schulman et al., 2017) into our multi-objective weighted-sum version, where the clipped surrogate objective is applied to update the policy parameters, and the Generalized Advantage Estimation (Schulman et al., 2015) is used to compute the advantage function and target values.

A.2. Nonlinear Regression for Prediction Model

In Section 3.3, we construct the following four-parameter hyperbolic model $\Delta_j^i(\omega_j)$ for each policy π_i and each objective f_j:

$$\Delta_j^i(\omega_j) = A \cdot \frac{e^{a(\omega_j - b)} - 1}{e^{a(\omega_j - b)} + 1} + c. \tag{6}$$

In each generation, for each policy π_i, the data $\{(\boldsymbol{\omega}, \Delta\mathbf{F})\} = \{(\boldsymbol{\omega}, \mathbf{F}(\pi') - \mathbf{F}(\pi))\}$ in the neighborhood of the policy π_i (*i.e.,* $\|\mathbf{F}(\pi) - \mathbf{F}(\pi_i)\| < \delta\|\mathbf{F}(\pi_i)\|$) is collected from the RL history record $\mathcal{R}$, and the following nonlinear least-square regression is applied to fit the parameters of the hyperbolic model:

$$\min_{\boldsymbol{\xi}} \sum_{k=1}^{n} \rho((\Delta_j^i(\boldsymbol{\omega}_j^i; \boldsymbol{\xi}) - \Delta\mathbf{F}_j^i)^2),$$

where $\boldsymbol{\xi} = \{A, a, b, c\}$ are the four parameters that need to be determined for each model, n is the size of the dataset and $\rho(z) = 2(\sqrt{1+z} - 1)$ is the soft-l_1 loss. For the threshold δ, we set it as 0.1 in all examples. In the rare cases that there is not enough data around the policy π_i (< 4 data points), we iteratively relax the threshold δ until enough data points are collected.

A.3. Time Complexity Analysis for Prediction-Guided Optimization

In Section 3.4, we present a prediction-guided optimization for task selection. As shown in Algorithm 3, given the current population $\mathcal{P}$, the prediction models $\{\boldsymbol{\Delta}^i\}$, and the current Pareto archive EP, we adopt a greedy algorithm to solve a knapsack problem to select the tasks that can best improve Pareto quality. Our greedy algorithm maintains a virtual policy set EP′ for the predicted Pareto archive. It then iteratively selects the task that best improves the Pareto metric of EP′ and then updates EP′ by inserting the predicted offspring policy of the selected task. We analyze the time complexity of this algorithm below.

Let n be the number of tasks to be selected, m be the number of objectives in the problem, and K be the number of candidate weights sampled for each policy in the population. The algorithm will run for n iterations, and in each iteration, the predicted Pareto quality needs to be calculated for each task (policy-weight pair) by virtually inserting the predicted offspring objectives of the task into the virtual Pareto set EP′. For hypervolume metric, given a Pareto front EP′ of size N, our implementation computes the hypervolume in $\mathcal{O}(N)$ when $m = 2$ and $\mathcal{O}(N^{m-2} \log N)$ when $m > 2$ (Guerreiro et al., 2012), and computes the sparsity in $\mathcal{O}(mN \log N)$. This computation can be fully parallelized for each task. Virtually updating the N by the new

offspring can be done in $\mathcal{O}(N)$. Let p be the number of parallel threads, then the time complexity in each iteration to select the best task is $\mathcal{O}\left(K|\mathcal{P}|(N + mN\log N)/p\right)$ when $m = 2$ and $\mathcal{O}\left(K|\mathcal{P}|(N^{m-2}\log N + mN\log N)/p\right)$ when $m > 2$. Because there are n iterations and usually $N \gg m$, so the time complexity of our implementation is $\mathcal{O}\left(nK|\mathcal{P}|N\log N/p\right)$ when $m = 2$ and $\mathcal{O}\left(nK|\mathcal{P}|N^{m-2}\log N/p\right)$ when $m > 2$. Compared to the time spent in the reinforcement learning, the time cost in task selection optimization is negligible when the number of objectives $m = 2$ or $m = 3$. There also exists some efficient hypervolume estimation algorithms which can be used to further reduce the time complexity.

B. Performance Buffer Strategy

In this section, we introduce the performance buffer strategy for population update.

In the proposed evolutionary learning algorithm, We apply the performance buffer strategy (Schulz et al., 2018) in each generation to update the population of the policies. The performance buffer is a data structure to store the population and aims to maintain the performance and the diversity of the policies in the population. Here we assume all the objectives are non-negative. As illustrated in Figure 7, the performance buffer strategy evenly discretizes the performance space into P_{num} bins by angles. Then each policy π_i is inserted in to the corresponding buffer based on its objective $\boldsymbol{F}(\pi_i)$. Finally P_{size} policies in each buffer with the largest distance to the origin are selected into the population.

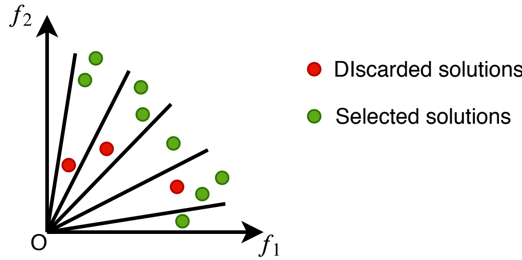

Figure 7. **Performance Buffer.** In 2-objective case, the performance buffer strategy split the performance space into bins and then the solutions with top P_{size} distance to the origin in each bin are selected into population. $P_{size} = 2$ in this example.

C. Benchmark Problems

In this section, we give more details about the proposed benchmark problems.

We designed six multi-objective control problems with continuous action space based on Mujoco gym environments. We keep the same state space $\mathcal{S}$ and action space $\mathcal{A}$ as used in Mujoco for most problems. We make several modifications to the physical parameters of some robots (e.g. mass, friction, actuator limit) for working better on the multiple objectives, which can be found in our provided code. The reward functions for each environment are illustrated as follows, where R_i means the reward for the i-th objective we care, and the reward function are designed so that the values are in similar scale.

C.1. HalfCheetah-v2

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{17}, \mathcal{A} \in \mathbb{R}^{6}$, and the environment runs for 500 steps.

The first objective is forward speed:

$$R_1 = \min(v_x, 4) + C$$

The second objective is energy efficiency:

$$R_2 = 4 - \sum_i a_i^2 + C$$

where $C = 1$ is the alive bonus, v_x is the speed in x direction, a_i is the action of each actuator.

Since both objectives in each step are smaller or equal to 5, the theoretical upper bound of the hyper volume (reference point at origin) in this problem is $(500 \times 5) \times (500 \times 5) = 6.25 \times 10^6$.

C.2. Hopper-v2

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{11}, \mathcal{A} \in \mathbb{R}^{3}$, and the environment runs for 500 steps.

The first objective is forward speed:

$$R_1 = 1.5v_x + C$$

The second objective is jumping height:

$$R_2 = 12(h - h_{init}) + C$$

where $C = 1 - 0.0002 \sum_i a_i^2$ is composed of alive bonus and energy efficiency, v_x is the speed in x direction, h is the current height, h_{init} is the initial height, a_i is the action of each actuator.

C.3. Swimmer-v2

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{8}, \mathcal{A} \in \mathbb{R}^{2}$, and the environment runs for 500 steps.

The first objective is forward speed:

$$R_1 = v_x$$

The second objective is energy efficiency:

$$R_2 = 0.3 - 0.15 \sum_i a_i^2$$

where v_x is the speed in x direction, a_i is the action of each actuator.

C.4. Ant-v2

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{27}, \mathcal{A} \in \mathbb{R}^{8}$, and the environment runs for 500 steps.

The first objective is x-axis speed:

$$R_1 = v_x + C$$

The second objective is y-axis speed:

$$R_2 = v_y + C$$

where $C = 1 - 0.5 \sum_i a_i^2$ is composed of alive bonus and energy efficiency, v_x is x-axis speed, v_y is y-axis speed, a_i is the action of each actuator.

C.5. Walker2d-v2

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{17}, \mathcal{A} \in \mathbb{R}^{6}$, and the environment runs for 500 steps.

The first objective is forward speed:

$$R_1 = v_x + C$$

The second objective is energy efficiency:

$$R_2 = 4 - \sum_i a_i^2 + C$$

where $C = 1$ is the alive bonus, v_x is the speed in x direction, a_i is the action of each actuator.

C.6. Humanoid-v2

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{376}, \mathcal{A} \in \mathbb{R}^{17}$, and the environment runs for 1000 steps.

The first objective is forward speed:

$$R_1 = 1.25v_x + C$$

The second objective is energy efficiency:

$$R_2 = 3 - 4 \sum_i a_i^2 + C$$

where $C = 3$ is the alive bonus, v_x is the speed in x direction, a_i is the action of each actuator.

C.7. Hopper-v3

Observation and action space dimensionality: $\mathcal{S} \in \mathbb{R}^{11}, \mathcal{A} \in \mathbb{R}^{3}$, and the environment runs for 500 steps.

The first objective is forward speed:

$$R_1 = 1.5v_x + C$$

The second objective is jumping height:

$$R_2 = 12(h - h_{init}) + C$$

The third objective is energy efficiency:

$$R_3 = 4 - \sum_i a_i^2 + C$$

where $C = 1$ is the alive bonus, v_x is the speed in x direction, h is the current height, h_{init} is the initial height, a_i is the action of each actuator.

D. Experiment Setup Details

In this section, we give more details about the experiment setup, including the implementation details for baseline algorithms, and the parameters and infrastructure used for training.

D.1. Framework for Baselines

Our proposed algorithm computes a dense and high-quality set of Pareto policies in the first step. We implement five baseline algorithms to compare the quality of the computed Pareto set approximations as described in Section 4.2. To fairly compare the baseline algorithms to ours, we implemented the **RA**, **PFA**, **MOEA/D** and **RANDOM** in a common framework as our proposed algorithm. The framework is illustrated in Algorithm 4. All the baselines and our algorithm starts with a warm-up stage and enters an evolutionary stage afterwards. We also apply the same population strategy and external Pareto archive to the baseline algorithms. The only differences among algorithms are the different TaskSelection functions.

Algorithm 4 Experiment Framework

Input: #parallel tasks n, #warm-up iterations m_w, #task iterations m_t, #generations M.
Initialize population $\mathcal{P}$, external pareto archive EP, and RL history record $\mathcal{R}$.
▷ Warm-up Stage
Generate task set $\mathcal{T} = \{(\pi_i, \boldsymbol{\omega}_i)\}_{i=1}^n$ by random initial policies and evenly distributed weight vectors.
$\mathcal{P}' \leftarrow$ MOPG($\mathcal{T}, m_w, \mathcal{R}$)
Update $\mathcal{P}$ and EP with $\mathcal{P}'$.
▷ Evolutionary Stage
for $generation \leftarrow 1, 2, ..., M$ **do**
  $\mathcal{T} \leftarrow$TaskSelection($n, \mathcal{T}, \mathcal{P}', \mathcal{P}$)
  $\mathcal{P}' \leftarrow$ MOPG($\mathcal{T}, m_t, \mathcal{R}$)
  Update $\mathcal{P}$ and EP with $\mathcal{P}'$.
end for
Output: Approximated pareto set EP.

Prediction-Guided MORL The TaskSelection function of our method composes a prediction function fitting algorithm and a prediction-guided task selection algorithm as described in Algorithm 1 and Section 3.

RA For Radial Algorithm, its TaskSelection function is simply replacing the policies in previous tasks by the new offspring policies. Its pseudo code is in Algorithm 5.

PFA In TaskSelection, PFA still uses new offspring policies as RA but slightly changes the optimization weights with a fixed step length to try to cover all weights. (Algorithm 6)

Algorithm 5 RA Task Selection

Input: #tasks n, last task set $\mathcal{T}_0$, latest offspring population $\mathcal{P}'$, population $\mathcal{P}$.
Initialize task set $\mathcal{T}$.
for each task $(\pi_i, \boldsymbol{\omega}_i) \in \mathcal{T}_0$ **do**
  $\mathcal{T}_i \leftarrow (\mathcal{P}'(i), \boldsymbol{\omega}_i)$
  Append task $\mathcal{T}_i$ into $\mathcal{T}$.
end for
Output: Selected task set $\mathcal{T}$.

Algorithm 6 PFA Task Selection

Input: #tasks n, last task set $\mathcal{T}_0$, latest offspring population $\mathcal{P}'$, population $\mathcal{P}$.
Initialize task set $\mathcal{T}$.
Compute weight change $\Delta\boldsymbol{\omega}$ in each generation.
for each task $(\pi_i, \boldsymbol{\omega}_i) \in \mathcal{T}_0$ **do**
  $\mathcal{T}_i \leftarrow (\mathcal{P}'(i), \boldsymbol{\omega}_i + \Delta\boldsymbol{\omega})$
  Append task $\mathcal{T}_i$ into $\mathcal{T}$.
end for
Output: Selected task set $\mathcal{T}$.

MOEA/D MOEA/D decomposes the multi-objective problem into subproblems by a set of uniformly distributed weights and solves all of those subproblems in a collaborative way. In each generation, MOEA/D selects the best policy for each subproblem. Specifically, for a subproblem associated with weight vector $\boldsymbol{\omega}$, MOEA/D selects the policy π from the current population which maximize the weighted-sum scalarization function $\boldsymbol{\omega} \cdot \mathbf{F}(\pi)$. (Algorithm 7)

Algorithm 7 MOEA/D Task Selection

Input: #tasks n, last task set $\mathcal{T}_0$, population $\mathcal{P}$.
Initialize task set $\mathcal{T}$.
for each task $(\pi_i, \boldsymbol{\omega}_i) \in \mathcal{T}_0$ **do**
  Initialize selected policy $\pi^* \leftarrow$ None.
  for each policy $\pi_j \in \mathcal{P}$ **do**
    if $\boldsymbol{\omega}_i \cdot \mathbf{F}(\pi_j) > \boldsymbol{\omega}_i \cdot \mathbf{F}(\pi^*)$ **then**
      $\pi^* \leftarrow \pi_j$
    end if
  end for
  $\mathcal{T}_i \leftarrow (\pi^*, \boldsymbol{\omega}_i)$
  Append task $\mathcal{T}_i$ into $\mathcal{T}$.
end for
Output: Selected task set $\mathcal{T}$.

RANDOM RANDOM baseline randomly assigns n tasks to be optimized in next generation. (Algorithm 8)

Algorithm 8 RANDOM Task Selection

Input: #tasks n, population $\mathcal{P}$.
Initialize task set $\mathcal{T}$.
for $i \leftarrow 1, 2, ..., n$ **do**
  Initialize task $\mathcal{T}_i \leftarrow$ None
  Randomly select policy $\pi \in \mathcal{P}$ and weight $\boldsymbol{\omega}$.
  $\mathcal{T}_i \leftarrow (\pi, \boldsymbol{\omega})$
  Append task $\mathcal{T}_i$ into $\mathcal{T}$.
end for
Output: Selected task set $\mathcal{T}$.

D.2. Training Details

We run all our experiments on VM instances with 96 Intel Skylake vCPUs and 86.4G memory on Google Cloud Platform, and no GPU is required. For all baselines, we use a standard two-layer neural network policy. The input layer receives the state vector of the robot, and the dimension of the network output is as twice as the size of the action space which consists both the mean and the standard deviation of the action. Each of the two hidden layers has 64 units and activated by $\tanh$ function. For META, we train it with same amount of total environment steps as our algorithm.

We use the same shared hyper parameters for all the baselines (except META). The shared parameters include:

- $\boldsymbol{n}$: the number of reinforcement learning tasks in each generation.
- **env_steps**: the number of environment steps in each reinforcement learning thread (i.e. the total number of environment steps equals $n \times$ env_steps).
- $\boldsymbol{m_w}$: the number of reinforcement learning iterations in the warm-up stage.
- $\boldsymbol{m_t}$: the number of reinforcement learning iterations in each generation in the evolutionary stage.
- $\boldsymbol{P_{num}}$: the number of performance buffers.
- $\boldsymbol{P_{size}}$: the size of each performance buffer.
- **PPO parameters**: all other hyper parameters in the PPO are same across all baselines and all problems.

Our algorithm only contains two unique hyper parameters $\boldsymbol{K}$ and $\boldsymbol{\alpha}$, where $\boldsymbol{K}$ is the number of sampled weights for each policy to discretize the mixed-integer programming problem, and $\boldsymbol{\alpha}$ is the weight of sparsity metric in the mixture metric (Section 3.4). All the hyper parameters used in our experiments are reported in the Table 2. Our PPO algorithm is implemented based on the codebase (Kostrikov, 2018), and the PPO parameters are reported in the Table 3.

Table 2. Hyper Parameters

Example	n	env_steps	m_w	m_t	P_{num}	P_{size}	K	α
HalfCheetah-v2	6	5×10^6	80	20	100	2	7	-1
Hopper-v2	6	8×10^6	200	40	100	2	7	-1
Swimmer-v2	6	2×10^6	40	10	100	2	7	-1
Ant-v2	6	8×10^6	200	40	100	2	7	-1
Walker2d-v2	6	5×10^6	80	20	100	2	7	-1
Humanoid-v2	6	2×10^7	200	40	100	2	7	-1
Hopper-v3	15	8×10^6	200	40	210	2	7	-10^6

Table 3. PPO Parameters

parameter name	value
timesteps per actorbatch	2048
num processes	4
lr	3×10^{-4}
gamma	0.995
gae lambda	0.95
num mini batch	32
ppo epoch	10
entropy coef	0
value loss coef	0.5

E. Additional Results

In this section, we give the full results for the experiments on all benchmark problems, including the training curves, Pareto set approximation results, and Pareto analysis results.

E.1. Pareto Quality Comparison

We first demonstrate the full plot of the learning curves of the hypervolume metric and the sparsity metric on all benchmark problems in Figure 8. Then we plot the Pareto fronts discovered by each algorithm on each benchmark problem in Figure 9 and Figure 10. The results show that our prediction-guided algorithm finds Pareto solution sets with higher hypervolume than the baseline algorithms on most problems. Most importantly, our algorithm is able to discover a significantly denser Pareto front than any exsiting algorithms.

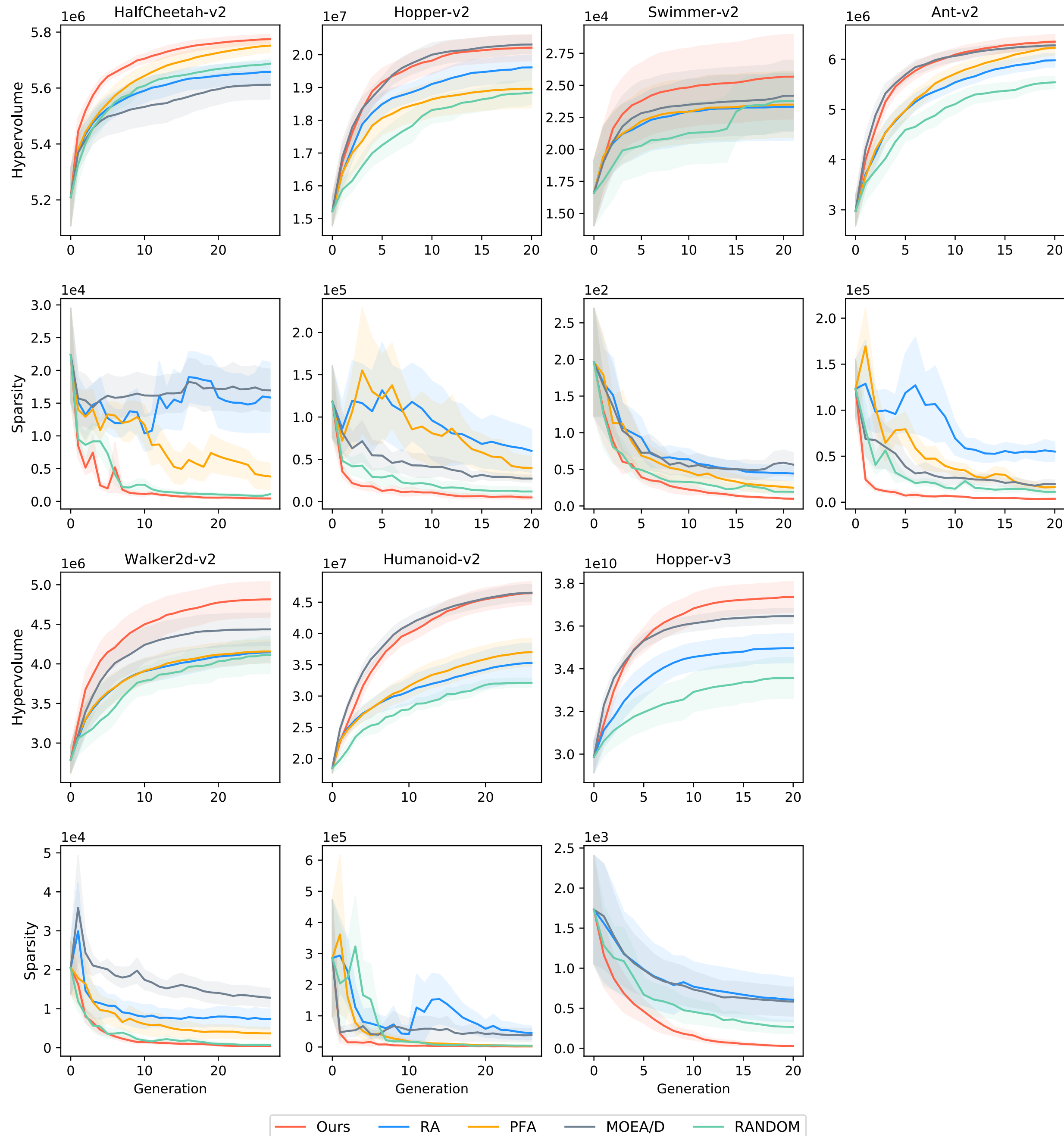

Figure 8. **The learning curves of hypervolume and sparsity metrics of different algorithms on benchmark problems.** The x-axis is the generation, the y-axis is the metric and the shadow area is the standard deviation. We do not plot the learning curve of META because it can be measured only during the final adaptation stage. For Hopper-v3, we do not run PFA as the sequence of the weights in three dimensional space is undefined.

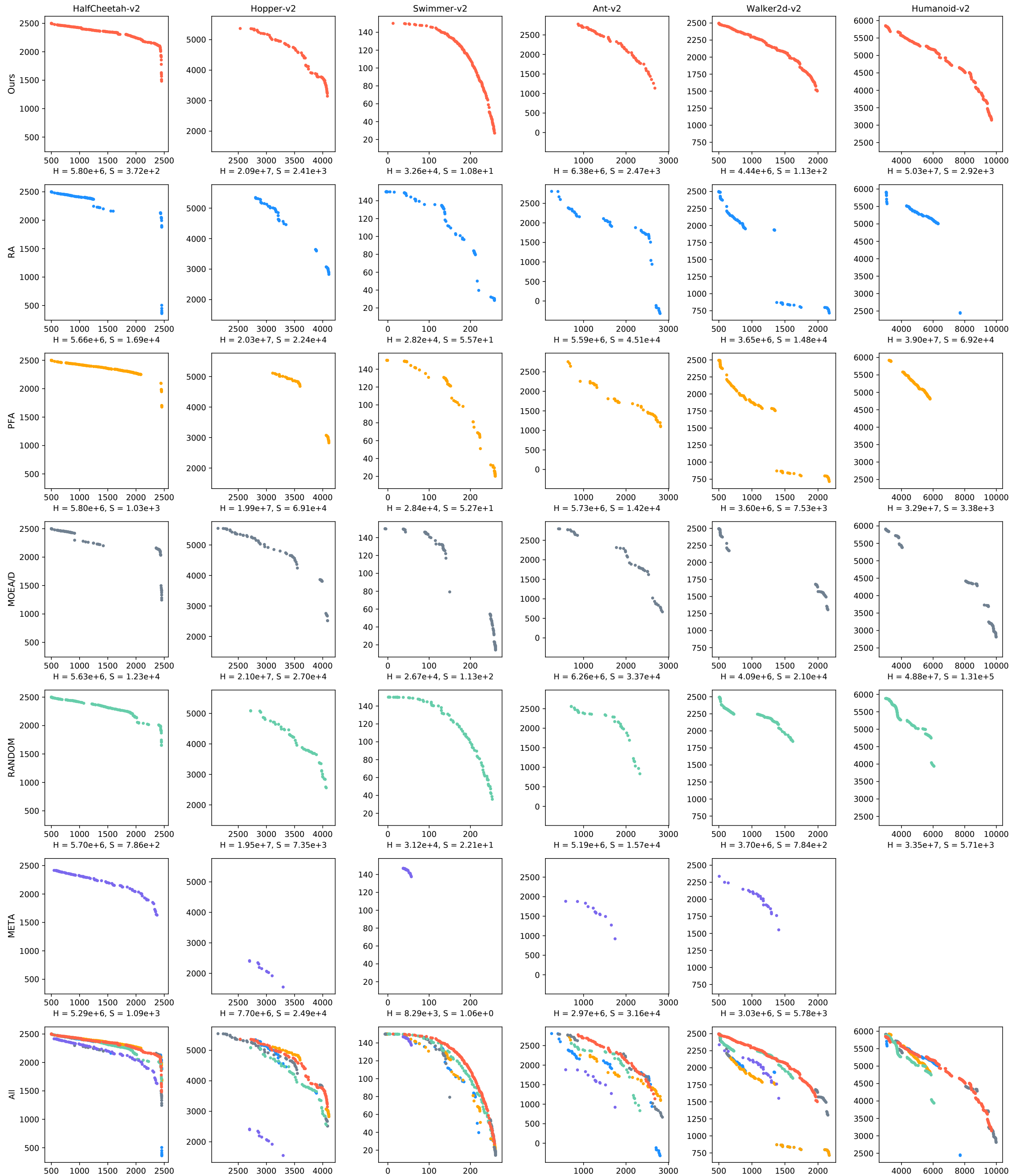

Figure 9. **The Pareto front approximation comparison for all 2-objective benchmark problems.** For each problem, we show the result for each algorithm with the same random seed. The Pareto of META on Humanoid-v2 problem is not plotted, since in our experiments, META is not able to generate a Pareto front in the first quadrant.

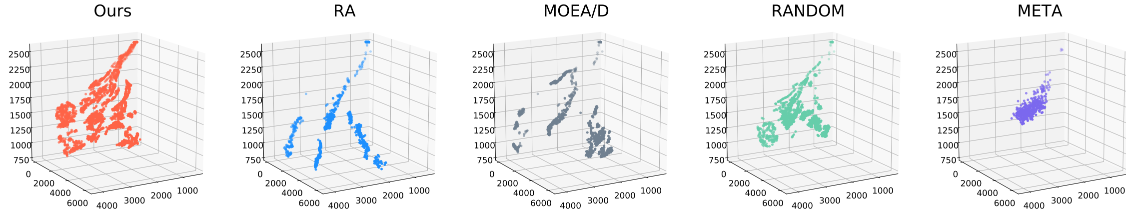

Figure 10. **The Pareto front approximation comparison for Hopper-v3.**

E.2. Pareto Analysis Results

We conduct Pareto analysis on the computed Pareto approximation to find the different policy families for all benchmark problems and construct continuous Pareto representation for each family. The full Pareto analysis results are shown in Figure 12. For two-objective problems, we construct the continuous Pareto front by linearly interpolating the consecutive policies from a same family. For three-objective problem, we represent the continuous Pareto front for each family by a triangle surface mesh. The results demonstrate that the Pareto solutions discovered by our evolutionary learning algorithm can be effectively represented by several different policy families, each occupying a continuous manifold in the parameter space and being responsible for a segment/patch on the Pareto front in the performance space.

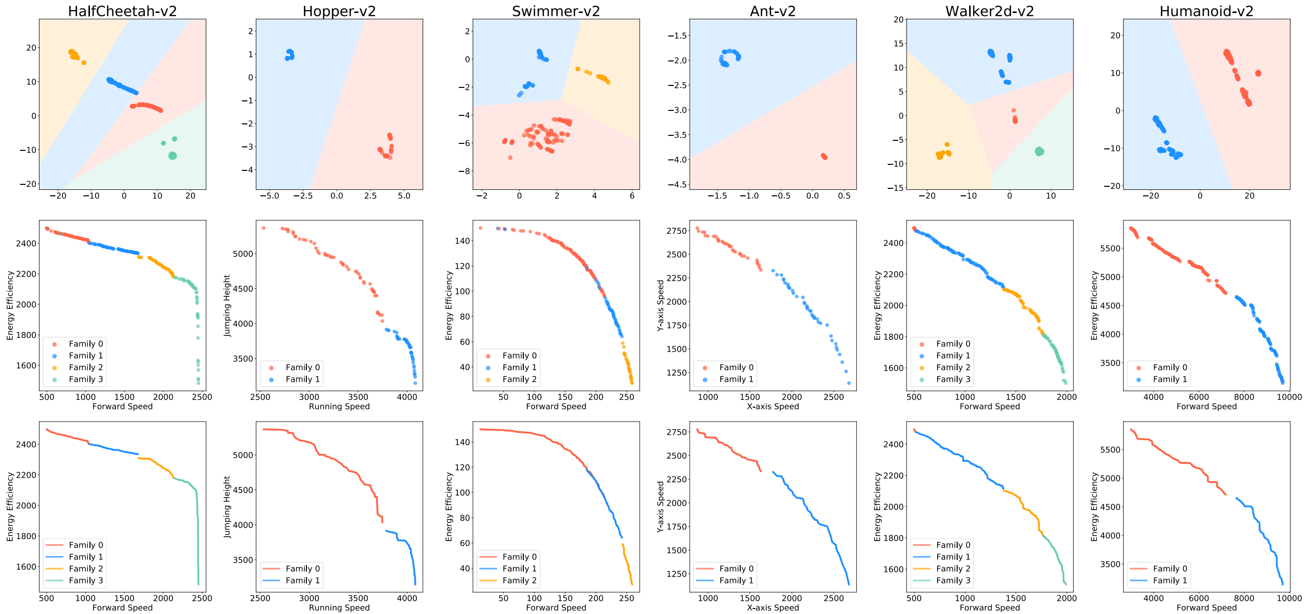

Figure 11. **Pareto analysis results for 2-objective benchmark problems.** The first row is the family identification in the parameter space by t-SNE and k-means. The second row is the corresponding objectives of those families in the performance space. The third row is the constructed continuous Pareto front approximation.

To validate the accuracy of the constructed continuous Pareto representation, we sample points on the continuous Pareto front for each family, and evaluate the relative error between the desired objectives and the objectives of the interpolated policy. For three-objective problem (Hopper-v3), we build a triangle mesh from the Pareto policies and then sample the testing points on the surfaces. The errors are reported in the first row of Table 4. The results show that the objectives of the interpolated policy is close enough to the desired objectives on the constructed Pareto front, which means by intra-family interpolation we can potentially get infinite number of policies on the Pareto front. We further test the necessity of the multi-family representation for the Pareto front. We sample points on the boundary of different families in performance space, and interpolate the policies from the different families and evaluate the relative error between the desired objectives and the objectives of the interpolated policy. The errors are reported in the second row of Table 4. The results show that it is impossible to get a policy with the desired objectives by interpolating the policies from different families, which further

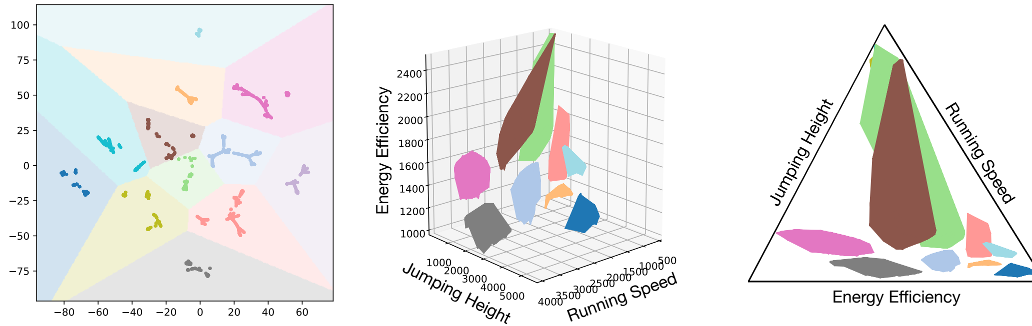

Figure 12. **Pareto analysis results for Hopper-v3.** (left) The family identification in the parameter space by t-SNE and k-means. (middle) The constructed continuous Pareto front approximation in the performance space. (right) Embedding the continuous Pareto front approximation in barycentric coordinates for better visualization.

validate that the different families are disjoint in the parameter space. The illustration of the tests is shown in Figure 13.

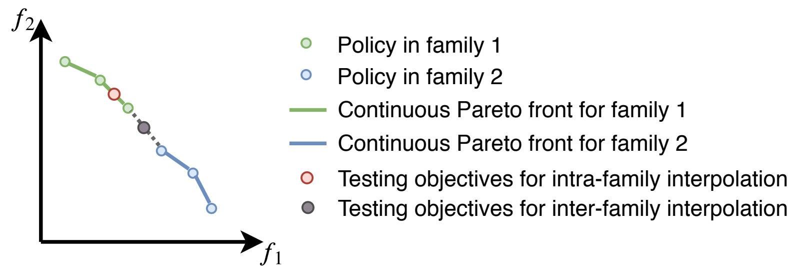

Figure 13. **Illustration for continuous Pareto front accuracy evaluation.** We sample testing objectives on the continuous Pareto front to test intra-family interpolation and on the boundary between families to test inter-family interpolation.

Table 4. **Intra-family and Inter-family interpolation errors.** We evaluate the relative error for intra-family and inter-family interpolation respectively. For two objective cases, we sample 1000 testing objectives for intra-family interpolation and 100 testing objectives for inter-family interpolation. For the three objective case the number of samples are 20000 and 5000 for intra-family and inter-family respectively. The average errors are reported below.

Example	HalfCheetah-v2	Hopper-v2	Swimmer-v2	Ant-v2	Walker2d-v2	Hopper-v3
Intra-Family	0.39%	0.85%	0.47%	3.94%	0.52%	0.87%
Inter-Family	6.71%	88.62%	2.81%	67.84%	7.95%	17.34%

E.3. Increase Number of Tasks

We further test the effectiveness of our proposed algorithm for larger number of tasks n. We increase the number of tasks so that the interval between evenly distributed weights decreases from 0.2 to 0.1 for two-objective examples and from 0.25 to 0.2 for Hopper-v3. In other words, for all two-objective examples, we change n from 6 to 11, and for Hopper-v3, we change n from 15 to 21. We run each algorithm on each problem for six times and report the average hypervolume and sparsity metrics in Table 5. The results show that the quality of the computed Pareto set approximation improves when the number of parallel tasks increases for all algorithms, and our algorithm consistently outperforms baselines in both cases.

Table 5. **Evaluation of our algorithm and baseline algorithms on the proposed benchmark problems for more parallel tasks.** $n = 11$ for all two-objective problems and $n = 21$ for Hopper-v3. We run all algorithms on each problem for 6 runs and report the average Hypervolume (Hv) and Sparsity (Sp) metrics. Bold number is the best in each row.

EXAMPLE	METRIC	OURS	RA	PFA	MOEA/D	RANDOM	META
HALFCHEETAH-V2	HV ($\times 10^6$)	**5.81**	5.76	5.77	5.74	5.77	5.27
	SP ($\times 10^3$)	**0.29**	8.63	2.97	7.74	0.92	1.22
HOPPER-V2	HV ($\times 10^7$)	**2.09**	2.00	2.00	2.08	1.95	1.23
	SP ($\times 10^4$)	1.58	4.61	6.75	4.30	**1.13**	1.73
SWIMMER-V2	HV ($\times 10^4$)	**3.03**	2.65	2.65	2.82	2.87	1.58
	SP ($\times 10^1$)	**1.42**	3.92	3.06	2.31	2.99	5.58
ANT-V2	HV ($\times 10^6$)	**6.54**	6.44	6.44	6.42	5.68	4.17
	SP ($\times 10^4$)	**0.43**	2.78	2.33	1.33	0.81	1.58
WALKER2D-V2	HV ($\times 10^6$)	**4.95**	4.38	4.40	4.76	4.30	2.08
	SP ($\times 10^4$)	**0.02**	0.35	0.21	0.29	0.02	0.72
HOPPER-V3	HV ($\times 10^{10}$)	**3.83**	3.63	-	3.60	3.39	2.39
	SP ($\times 10^3$)	**0.02**	0.25	-	0.18	0.09	3.44

E.4. Parameter Study for t-SNE

Dimensionality reduction methods usually contain some hyperparameters that affect the embedding outcomes. However, in our experiments, the Pareto analysis results (i.e., the family clustering results) are not influenced by different parameters of the t-SNE in most cases. In this section, we take the Walker2d-v2 problem as an example, and conduct a parameter study to show how the Pareto analysis results are affected by the t-SNE parameters. Our default t-SNE parameters are reported in Table 6. We change each parameter within a reasonable range as shown in the last column of Table 6, and apply the Pareto analysis to identify the families as described in Section 3.5. The results are demonstrated in Figures 14, 15, and 16. The results show that although the visualization changes dramatically when we change the parameters, the family clustering results remain consistent.

Table 6. t-SNE Parameters

parameter name	default value	range
perplexity	50	$[40, 80]$
learning_rate	50	$[10, 200]$
n_iter	2000	$[500, 5000]$

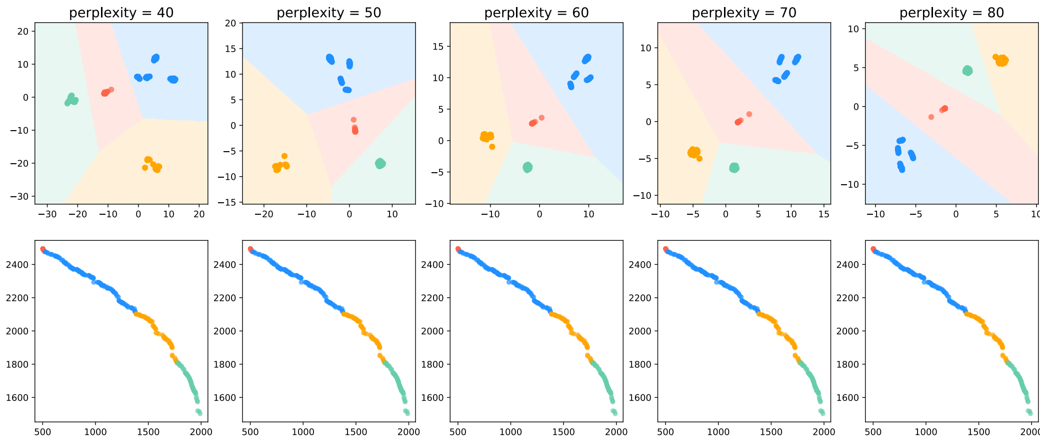

Figure 14. **t-SNE parameter study for perplexity.** We change the perplexity parameter from 40 to 80 and conduct the Pareto analysis to identify families for Walker2d-v2 problem. The first row is the clustered families in the embedding space, and the second row is in the performance space.

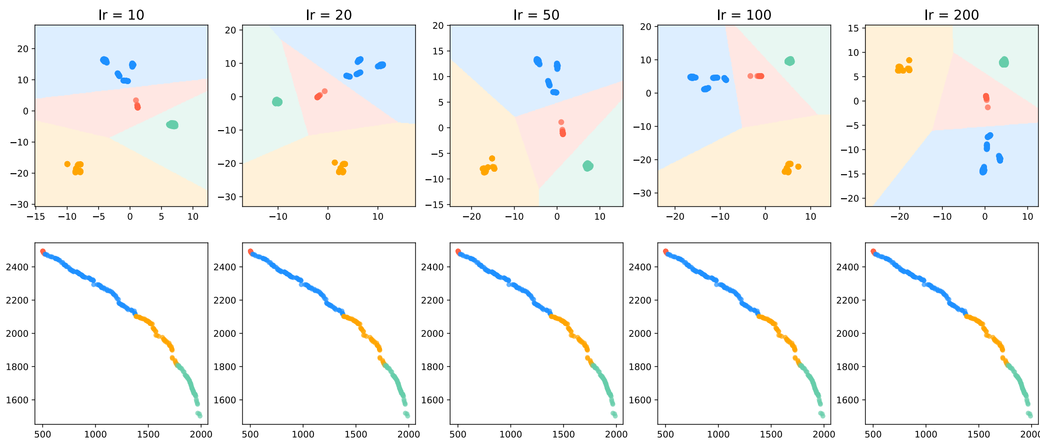

Figure 15. **t-SNE parameter study for learning rate.** We change the learning rate from 10 to 200 and conduct the Pareto analysis to identify families for Walker2d-v2 problem. The first row is the clustered families in the embedding space, and the second row is in the performance space.

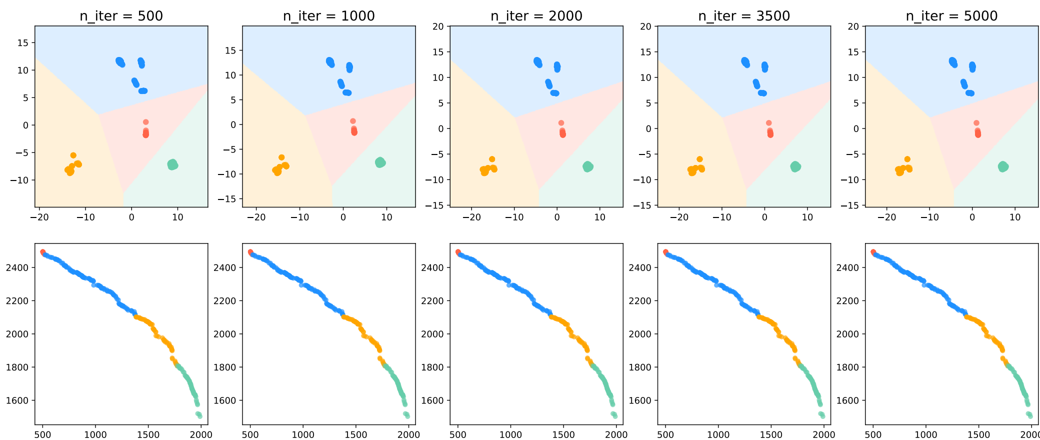

Figure 16. **t-SNE parameter study for n_iter.** We change the n_iter parameter from 500 to 5000 and conduct the Pareto analysis to identify families for Walker2d-v2 problem. The first row is the clustered families in the embedding space, and the second row is in the performance space.

E.5. Dimensionality Methods Comparison

For the purpose of dimensionality reduction, there are also other available methods (e.g., LLE, PCA, Isomap). We choose t-SNE due to its much clearer distinguishing effect. In this section, we conduct a comparison among these different dimensionality reduction methods. For each dimensionality method, we embed the policy parameters into a two dimensional space and use the same k value to cluster the families by k-means. Figure 17 shows the comparison on the Walker2d-v2 problem, and Figure 18 shows the comparison on the Ant-v2 problem. As we can see, even though t-SNE, Isomap, and LLE give identical family clustering results, t-SNE produces the visualization that most clearly demonstrates that the policies from the same family are clustered together and are far from the policies from other families.

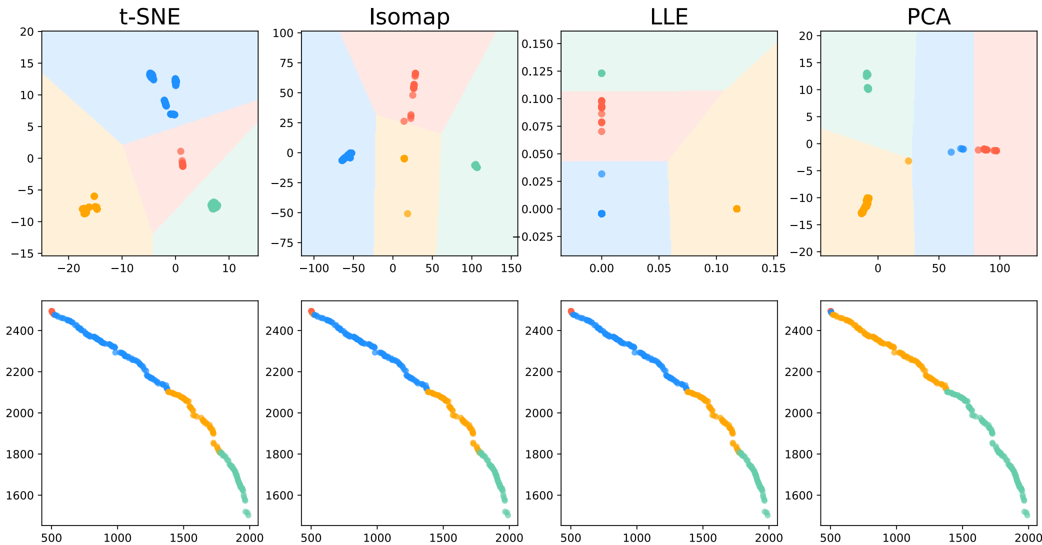

Figure 17. **Comparison of dimensionality reduction methods on Walker2d-v2 problem.** We use t-SNE, Isomap, LLE and PCA to embed the same computed Pareto set and conduct k-means to cluster the families. The first row is the clustered families in the embedding space, and the second row is in the performance space.

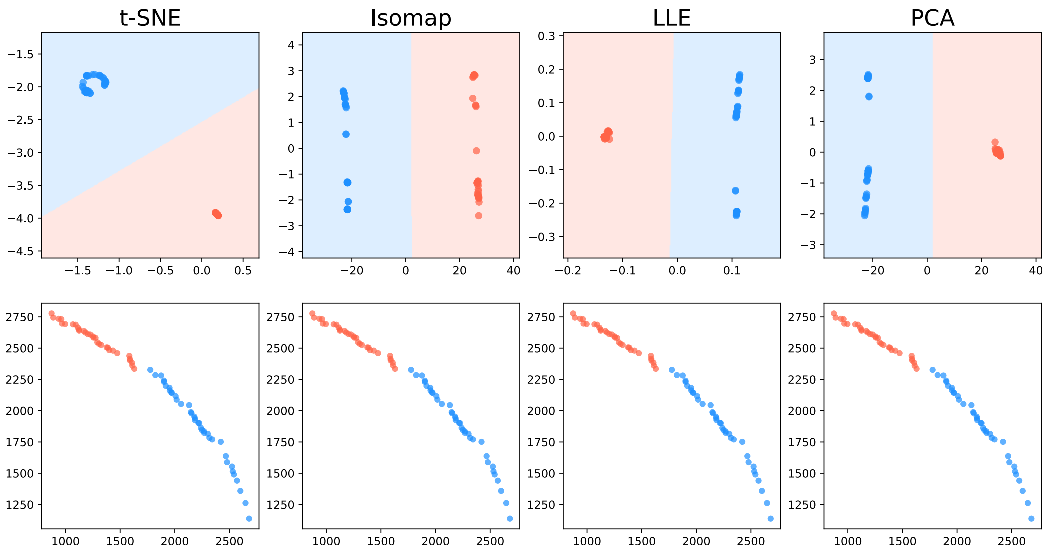

Figure 18. **Comparison of dimensionality reduction methods on Ant-v2 problem.** We use t-SNE, Isomap, LLE and PCA to embed the same computed Pareto set and conduct k-means to cluster the families. The first row is the clustered families in the embedding space, and the second row is in the performance space.