

Flow Matching Policy Gradients

Simple Online Reinforcement Learning with Flow Matching

Contents

[Flow Matching](#)

[On-Policy RL - Sample, Score, Reinforce](#)

[Flow Matching Policy Gradients](#)

[FPO in Action](#)

Flow models have become the go-to approach to model distributions in continuous space. They soak up data with a simple, scalable denoising objective and now represent the state-of-the art in generating images, videos, audio and, more recently, robot actions. However, they're still not widely used for learning from rewards with reinforcement learning.

To perform RL in continuous spaces, practitioners typically train far simpler Gaussian policies, which represent a single, ellipsoidal mode of the action distribution. Flow-based policies can capture complex, multimodal action distributions, but they are primarily trained in a supervised manner with behavior cloning (BC). We show that it's possible to train RL policies using flow matching, the framework behind modern diffusion and flow models, to benefit from its expressivity.

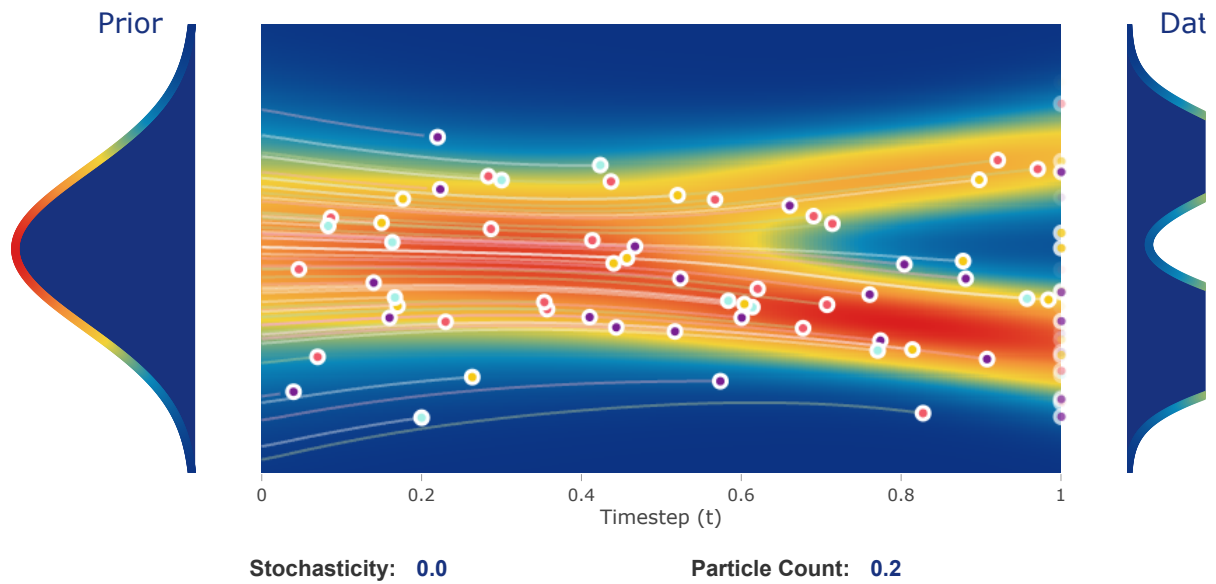
We introduce **Flow Policy Optimization** (FPO), a new algorithm to train RL policies with flow matching. It can train expressive flow policies from only rewards. We find its particularly useful to learn underconditioned policies, like humanoid locomotion with simple joystick commands. We felt limited by the expressiveness of Gaussian policies and thought diffusion could help. In this blog post, we'll explain how we connect flow matching and on-policy RL in a way that makes sense without an extensive RL background.

Flow Matching

Flow matching [\[1\]](#) optimizes a model to transform a simple distribution (e.g., the Gaussian distribution) into a complex one through a multi-step mapping called the marginal flow.

The flow smoothly directs a particle x_t to the data distribution, so integrating a particle's position across time according to the flow yields a sample from the data distribution. Equivalently, sampling is the process of solving an ordinary differential equation (the flow), which we can do deterministically or with stochastic "churn" every step.

We can actually calculate the marginal flow *analytically*, which we do in real-time in the plot below. We added interactive control over the data distribution and sampling stochasticity, so try messing with it!

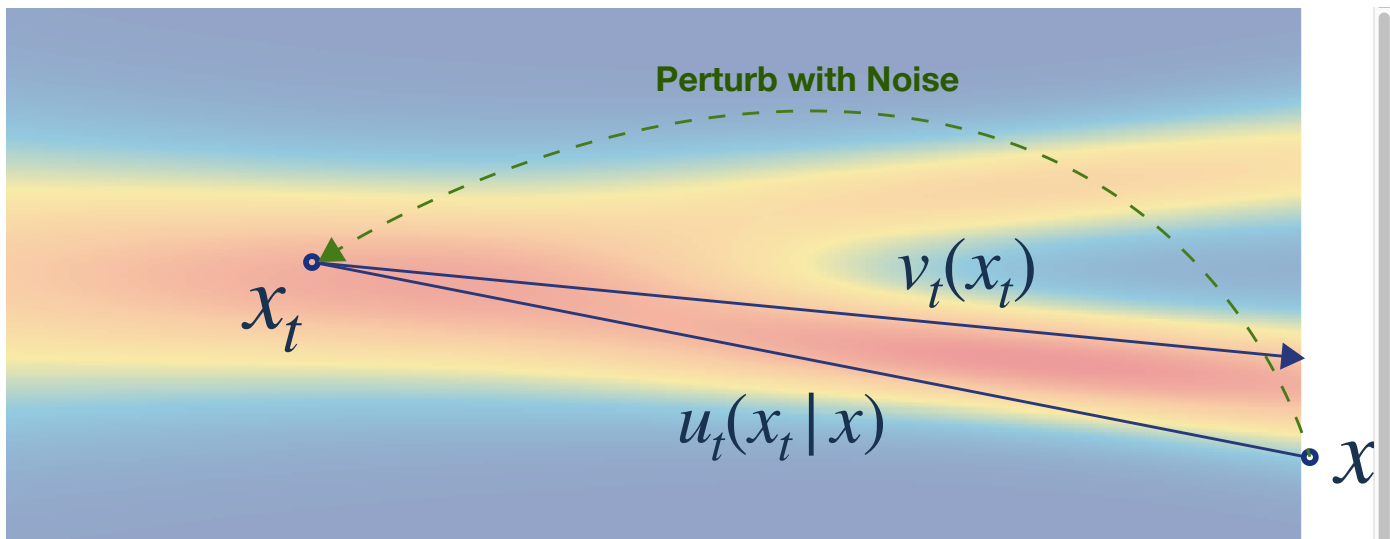


Drag the control points on the right to adjust the data mixture (position and weighting of components). Control stochasticity and particle rate with the sliders above.

Each particle above represent an x_t noisy latent that gets iteratively denoised as the time is integrated from zero to one. Drag the control points of the modes on the right to see how the underlying PDF and the particle trajectories change. Notice how the probability mass flows smoothly from the initial noise to form two distinct modes. The multi-step mapping is the magic that lets flow models transform a simple, tractable distribution into one of arbitrary complexity.

While it's possible to interactively compute this flow in 1D, it becomes intractable over large datasets in high dimensional space. Instead, we use flow matching, which compresses the marginal flow into a neural network through a simple reconstruction objective.

Flow matching perturbs a clean data sample with Gaussian noise then tasks the model with reconstructing the sample by predicting the velocity, which is the derivative of x_t 's position *w.r.t.* time. In expectation over a fixed dataset, this optimization recovers the marginal flow for any x_t . Integrating x_t 's position across time according to a well-trained model's velocity prediction will recover a sample from the data distribution.



Flow matching the velocity prediction $v_t(x_t)$ to the conditional flow $u_t(x_t|x)$.

Geometrically, the marginal flow points to a *weighted-average* of the data where the weights are a function of the timestep and distance from x_t to each data point. You can see the particles follow the marginal flow exactly in the plot above when stochasticity is turned off. At a high level, flow matching learns to point the model's flow field, $v_t(x_t)$, to the data distribution.

Flow matching has statistical significance too. Instead of computing exact flow likelihoods (expensive and unstable), it optimizes a lower bound called the Evidence Lower Bound (ELBO) [2]. Increasing the ELBO pushes the model toward higher likelihoods without computing them directly. In the limit, the flow model will sample exactly from the probability distribution of the dataset. So if you've learned the flow function well, you've learned the underlying structure of the data.

TLDR: Flowing toward a data point increases its likelihood under the model.

On-Policy RL: Sample, Score, Reinforce

On-policy reinforcement learning follows a basic core loop: sample from your policy, score each action with rewards, then make high-reward actions more likely. Rinse and repeat.

This procedure climbs the policy gradient—the gradient of expected cumulative reward. Your model collects “experience” by sampling its learned distribution, sees which samples are most advantageous, and adjusts to perform similar actions more often.

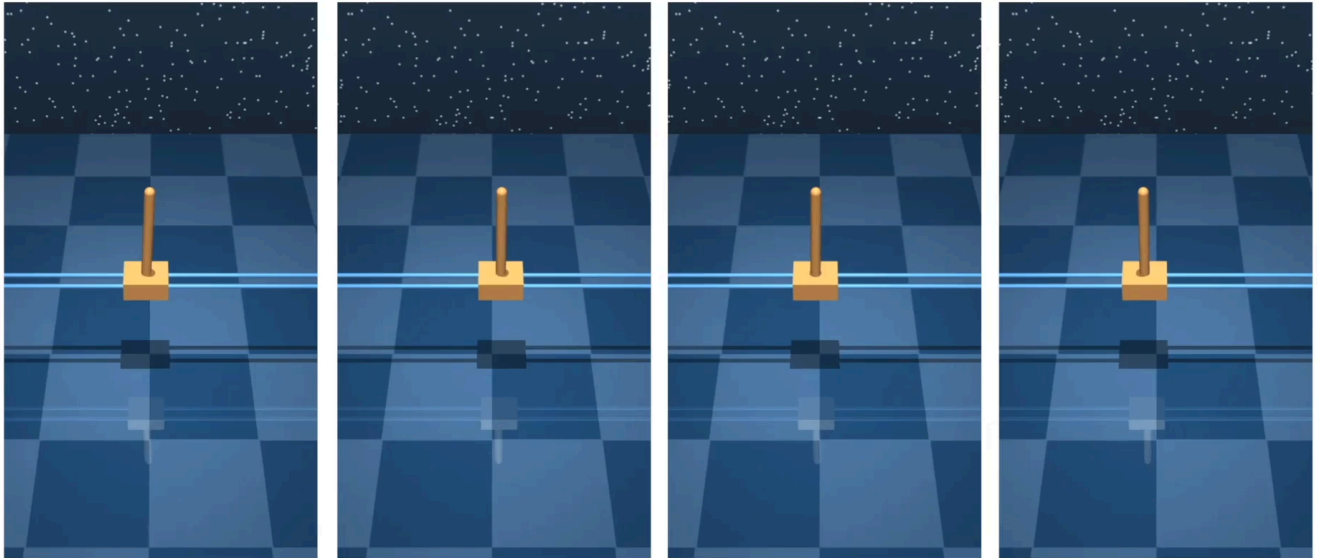
On-policy RL can be cast as search iteratively distilled into a model. The policy “happens upon” good behaviors through exploration, then reinforces them. Over time, it discovers the patterns in the random successes and develops reliable strategies. You can start from a pretrained model and continue training with RL to explore within a pruned prior distribution rather than at random. This is the dominant approach to upcycle LLMs for preference alignment [3] and mathematical reasoning [4].

Illustrative Example

We use the toy cartpole task from DMControl [5] for clear illustration. The goal is to move a cart along a rail to balance an attached pole vertically. Here's how this manifests as an RL loop:

1. Sample an action from your model's state-conditional distribution then simulate a step of physics. Do this back and forth in succession over a time horizon (rollouts).
2. Score each sequence with rewards for each timestep (“how vertical is the pole?”).
3. Train your model to boost the likelihood of actions that lead to high-reward sequences.
 - Repeat above until your model reliably balances the pole.

Sample and score rollouts:



1.2

0.2

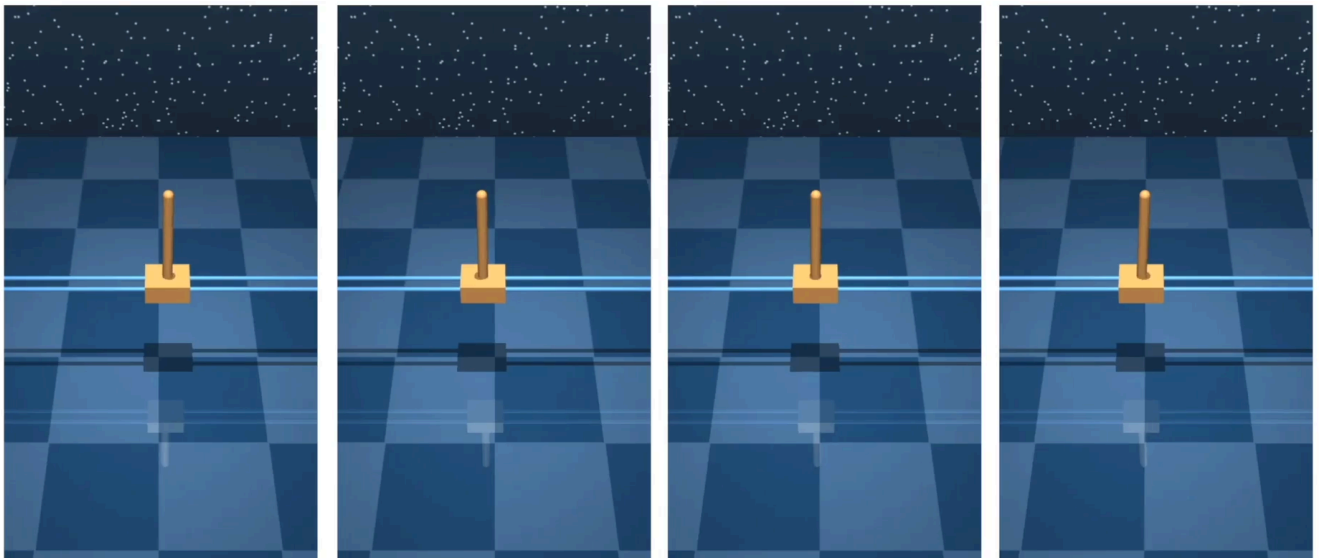
0.7

0.4

On-policy RL samples multiple rollouts of actions then scores them according to the reward. In this case, only one (leftmost) rollout successfully balances the pole across the whole time horizon.

Calculate each advantage and estimate the policy gradient:

From the rewards, we estimate advantages. These can be viewed as the reward over time (return) normalized *w.r.t.* the expected return. This expectation is what the critic learns in PPO [6] or computed as the average of a group's rewards in GRPO [7].



-0.6

-0.1

0.1

0.5

Advantages are lower-variance estimates of action "goodness" than rewards. There is a design space to estimating advantages, but one way to think of them is as normalized rewards.

Given the advantages, we train the model on each data point with a gradient update scaled by its corresponding advantage. So, if the advantage is negative, it will become less likely. Postive advantage, more likely.

Typically, the policy gradient is computed in discrete space or using Gaussian likelihoods. Flow Policy Optimization extends the policy gradient to flow models, which introduces some important details we discuss in the following sections.

Flow Matching Policy Gradients

To reiterate, the goal of on-policy RL is simple: increase the likelihood of high-reward actions. Meanwhile, flow matching naturally increases likelihoods by redirecting probability flow toward training samples. This makes our objective clear—**redirect the flow toward high reward actions**.

In the limit of perfect optimization, flow matching assigns probabilities according to the frequency of samples in your training set. Since we're using RL, that "training set" is dynamically generated from the model each epoch.

Advantages make the connection between synthetic data generation and on-policy RL explicit. In RL, we calculate the advantage of each sampled action, a quantity that indicates how much better it was than expected. These advantages are centered around zero to reduce variance: positive for better-than-expected actions, negative for worse. Advantages then become a *loss weighting* in the policy gradient. As a simple example, if an action is very advantageous, the model encounters a scaled-up loss on it and learns to boost it aggressively.

Advantage-Weighting

Policy Gradient

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\pi_{\theta} \sim (\tau, T)} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_t | \mathbf{s}_t) \mathbf{A}^{\pi_{\theta}}(\mathbf{s}_t, \mathbf{a}_t) \right]$$

Supervised Learning Gradient

The policy gradient resembles a standard log-likelihood supervised learning gradient on synthetic samples with the loss scaled by the reward or advantage (both are valid).

Zero-mean advantages are fine for RL in discrete spaces because a negative advantage simply pushes down the logit of a suboptimal action, and the softmax ensures that the resulting action probabilities remain valid and non-negative. Flow matching, however, learns probability flows to sample from a training data distribution. These are nonnegative by construction, so negative loss weights break this clean interpretation.

There's a simple solution: make the advantages nonnegative. Shifting advantages by a constant doesn't change the policy gradient. In fact, this is the mathematical property that lets us use advantages in the first place. Here's how we can understand non-negative advantages in the flow matching framework:

Path from \mathbf{x}_t to \mathbf{x}

$$u_t(x_t) = \sum_x \mathbf{u}_t(\mathbf{x}_t | \mathbf{x}) \frac{p_t(x_t | x)}{p_t(x_t)} \mathbf{q}(\mathbf{x})$$

Marginal Flow
Data Frequency

The marginal flow is a linear combination of the (conditional) flow to each data point. The weighting of each path scales with probability of drawing the data point from the dataset, $q(x)$.

Advantages manifest as loss-weighting, which can be intuitively expressed in the marginal flow framework. The marginal flow is the weighted average of the paths (the u_t 's) from the current noisy particle, x_t , to each data point x . The paths are also weighed by $q(x)$, the probability of drawing x from your training set. This is typically a constant $\frac{1}{N}$ for a dataset of size N , assuming every data point is unique. Loss weights are equivalent to altering the frequency of the data points in your training set. If the loss for a data point is scaled by a factor of 2, its equivalent to that data point showing up twice in the train set.

Flow Policy Optimization

Now, we can get a complete picture of our algorithm that connects flow matching and reinforcement learning: Flow Policy Optimization. FPO follows a three-step loop:

1. Generate actions from your flow model using your choice of sampler
2. Score them with rewards and compute advantages

3. Flow match (add noise and reconstruct) on the actions with an advantage-weighted loss

This procedure boosts the likelihood of actions that achieve high reward while preserving the desirable properties of flow models—multimodality, expressivity and the improved exploration that stems from them. Since FPO uses flow matching as its fundamental primitive, FPO-trained policies inherit the body of techniques developed for flow and diffusion models. These include guidance [8] [9] for conditioning and Mean Flows [10] for efficient sampling.

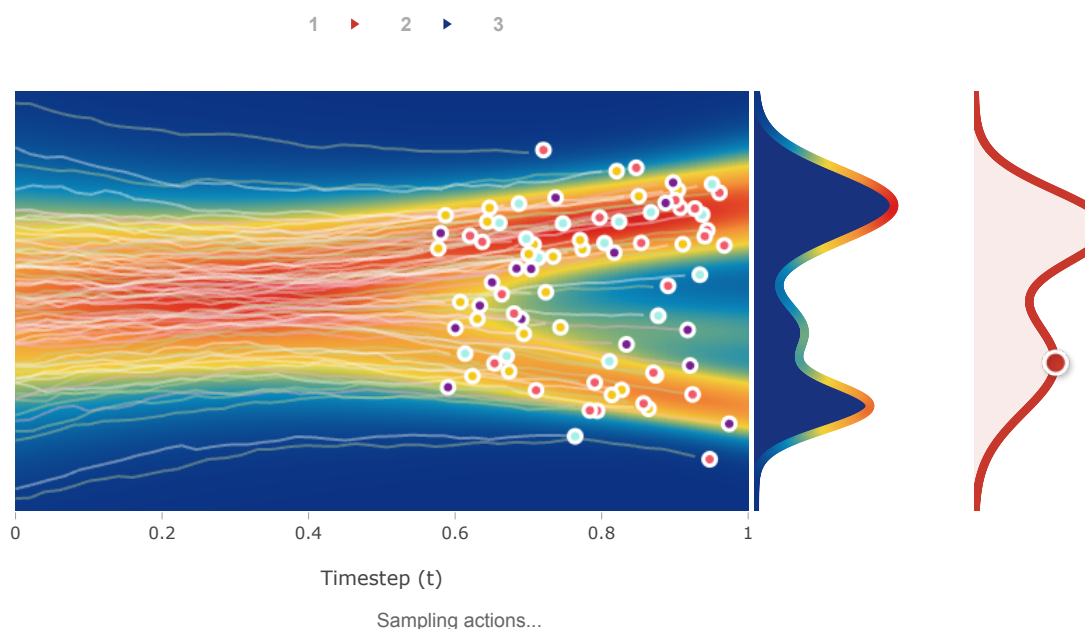
We visualize the three-step inner loop in the following interactive plot. We recommend viewing this on desktop. The red trace curve on the right determines the reward for different actions along the y-axis. It's controllable, drag the control points around to shape the reward function! The plot shows how FPO optimizes a flow-based policy to maximize the specified reward. It follows the three following stages that line up with label above the plot:

First, sample actions from the flow-based policy. At the first iteration, this will be whatever the model is initialized to (or two arbitrary modes in the plot below).

Second, for each sampled data point, multiply its influence by the reward. We do a k-means approximation of the resulting distribution for illustration and display it in the blue trace between the heatmap and red reward trace.

Third, redirect the flow according to this advantage-weighted distribution. In a real model, this happens by optimizing the FPO ratio just like how standard PPO optimizes its likelihood ratio.

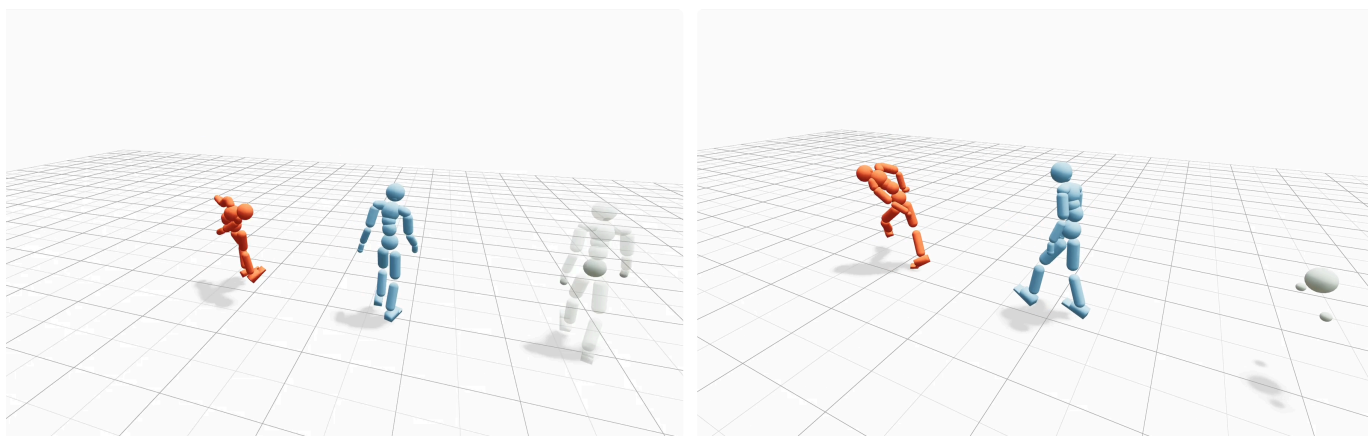
This represents *one epoch* of Flow Policy Optimization. The flow has been updated to sample higher-reward actions and we can repeat to continue climbing the policy gradient. The plot does this automatically, and you can reset it with the amber color reload button.



This is a pretty realistic analytical simulation of the FPO loop. It's missing one major component though, which is the trust region constraint [11]. This helps the optimization remain on-policy after multiple gradient steps per epoch. We encourage you to check out the paper to see how we implement this mechanism and for a more mathematical explanation of the algorithm.

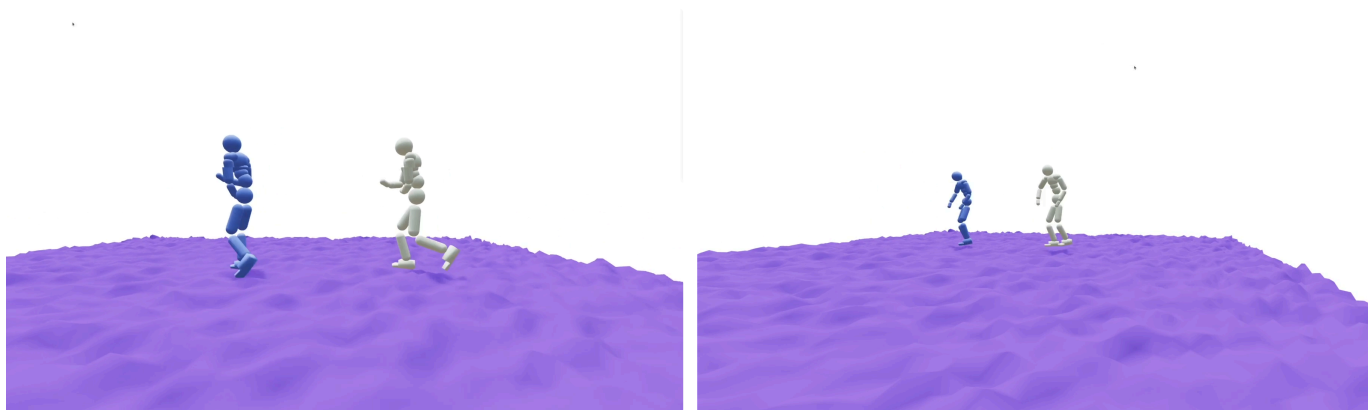
FPO In Action

We include a few video examples of FPO working on a range of control tasks. These demonstrate FPO's advantage over Gaussian policies for under-conditioned humanoid control. With only root-level commands, FPO successfully trains walking policies from scratch, while standard Gaussian policies fail to discover viable behaviors:



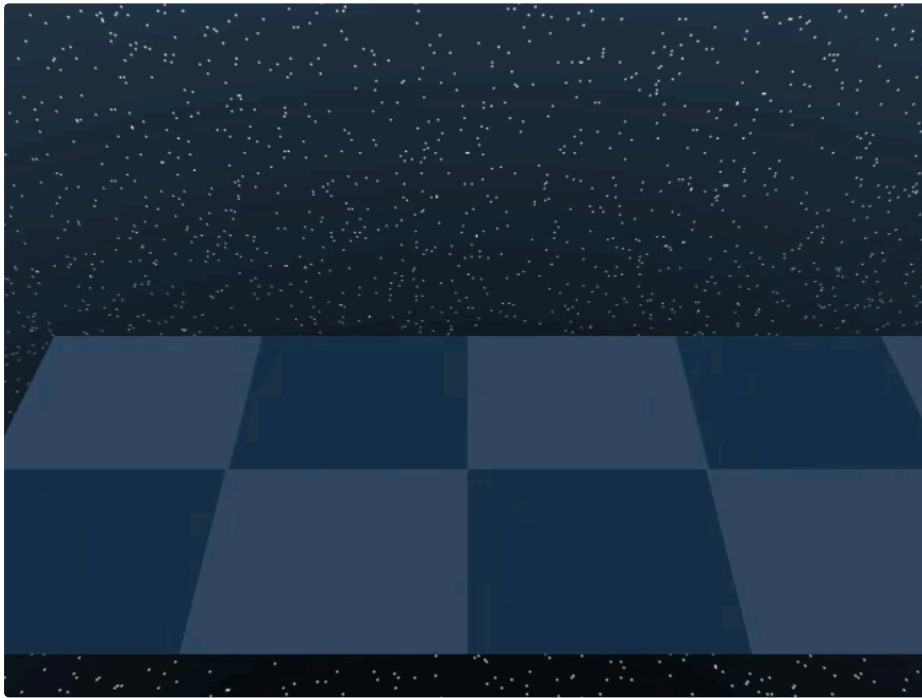
We compare Gaussian policies (orange) with FPO-trained policies (blue) when trained with sparse conditioning (gray).

Policies trained with FPO are robust to rough terrains for DeepMimic [12] -style motion tracking. We show a couple of examples:



Trained with terrain randomization, FPO walks stably across unseen procedurally generated rough ground.

It's not an RL paper without half cheetah! We compare quantitatively across DeepMind Control tasks to Gaussian policies and denoising MDPs in the main paper.



We show rollouts from our policy trained for the DeepMind Control task, CheetahRun, using FPO.

References

1. **Flow Matching for Generative Modeling** [\[PDF\]](#)
Lipman, Y., Chen, R.T.Q., Ben-Hamu, H., Nickel, M. and Le, M., 2023.
2. **Understanding Diffusion Objectives as the ELBO with Simple Data Augmentation** [\[PDF\]](#)
Kingma, D.P. and Gao, R., 2023.
3. **Training language models to follow instructions with human feedback** [\[PDF\]](#)
Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J. and Lowe, R., 2022.
4. **DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning** [\[PDF\]](#)
DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z.F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, B., Wang, B., Wu, B., Feng, B., Lu, C. and ..., 2025.
5. **DeepMind Control Suite** [\[PDF\]](#)
Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D.d.L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T. and Riedmiller, M., 2018.
6. **Proximal Policy Optimization Algorithms** [\[PDF\]](#)
Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017.
7. **DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models** [\[PDF\]](#)
Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y.K., Wu, Y. and Guo, D., 2024.
8. **Classifier-Free Diffusion Guidance** [\[PDF\]](#)
Ho, J. and Salimans, T., 2022.
9. **Diffusion Models Beat GANs on Image Synthesis** [\[PDF\]](#)
Dhariwal, P. and Nichol, A., 2021.
10. **Mean Flows for One-step Generative Modeling** [\[PDF\]](#)
Geng, Z., Deng, M., Bai, X., Kolter, J.Z. and He, K., 2025.
11. **Trust Region Policy Optimization** [\[PDF\]](#)
Schulman, J., Levine, S., Moritz, P., Jordan, M.I. and Abbeel, P., 2017.
12. **DeepMimic: example-guided deep reinforcement learning of physics-based character skills** [\[link\]](#)
Peng, X.B., Abbeel, P., Levine, S. and van de Panne, M., 2018. ACM Transactions on Graphics, Vol 37(4), pp. 1–14. Association for Computing Machinery (ACM). DOI: 10.1145/3197517.3201311