Report of Paper Diffusion Policy Policy Optimization

XINYU DU

The report should include,

(1):A literature review of potential methods for solving the problem

(2):Propose your choice of methods and the reason for the choice

(3):Present the answers of the question in clear and understandable language.

(4):Testing plans and testing results for checking your implementation is correct and your results are valid.

1 PyTorch-to-TensorFlow Conversion Wrapper Library

This library bridges the gap between PyTorch's user-friendly interface and widespread adoption in academia with TensorFlow's exceptional deployment capabilities and performance. It provides seamless conversion tools to transform PyTorch models into TensorFlow-compatible formats, enabling researchers and developers to leverage the strengths of both frameworks. Whether you're prototyping in PyTorch or deploying at scale with TensorFlow, this library ensures a smooth and efficient transition.

I write a broad library to convert the common functions and common network modules from PyTorch to TensorFlow. I want my users to afford the minimal prices and pain to convert them, so I nearly keep the original function names. But everything inside the function is going to process TensorFlow.Tensor rather than torch.Tensor. The philosophy of this design is to satisfy 90% of conversion work by just changing the dot symbol "." to an underscore symbol "_". So actually for most functions, I implement a wrapper for TensorFlow to run the similar logic as the original PyTorch Code.

The Strength of this framework:

- Pretty fast to replicate any code written in PyTorch with TensorFlow agent
- Accurate and easy to test. We could test each single modules separately. If the test function of each function and module is complete, it guarantees the results are the same as the original PyTorch one.
- Easy to find accurate documentation. This is a wrapper to implement the PyTorch functions in TensorFlow. So we could directly search the documentation of the original PyTorch modules. The Pytest is conducted to guarantee the output of the wrapper function in TensorFlow is the same as the output of the original PyTorch functions in PyTorch when these two outputs are both transferred to a Numpy Array.
- Enjoy the merit of efficiency and deployment of TensorFlow.

The Weakness of this framework:

- Difficult for people to understand if they don't know one of TensorFlow and PyTorch.

Author's Contact Information: Xinyu Du, xydu@link.cuhk.edu.hk.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Table 1. Functions Table1.

PyTorch	TensorFlow	TestFile
torch.Tensor.permute()	torch_tensor_permute()	test_permute.py
torch.Tensor.item()	torch_tensor_item()	test_item.py
torch.gather()	torch_gather()	test_gather.py
torch.quantile()	torch_quantile()	test_quantile.py
torch.flatten()	torch_flatten()	test_flatten.py
torch.abs()	torch_abs()	test_abs.py
torch.square()	torch_square()	test_square.py
torch.min()	torch_min()	test_min.py
torch.max()	torch_max()	test_max.py
torch.mean()	torch_mean()	test_mean.py
torch.softmax()	torch_softmax()	test_softmax.py
torch.stack()	torch_stack()	test_stack.py
torch.multinomial()	torch_multinomial()	test_multinomial.py
torch.where()	torch_where()	test_where.py
torch.tensor()	torch_tensor()	test_stack.py
torch.clip()	torch_clip()	
torch.clamp()	torch_clamp()	
torch.log()	torch_log()	test_log.py
torch.Tensor.clamp_()	torch_tensor_clamp_()	
torch.zeros()	torch_zeros()	test_torch_zeros.py
torch.ones()	torch_ones()	test_torch_ones.py
torch.nanmean()	torch_nanmean()	test_nanmean.py
torch.prod()	torch_prod()	test_prod.py
torch.cat()	torch_cat()	test_cat.py
torch.hstack()	torch_hstack()	test_hstack.py
torch.linspace()	torch_linspace()	test_linspace.py
torch.argmax()	torch_argmax()	test_argmax.py
torch.Tensor.view()	torch_tensor_view()	test_view.py
torch.reshape()	torch_reshape()	test_reshape.py
torch.arange()	torch_arange()	test_arange.py
torch.logsumexp()	torch_logsumexp()	test_logsumexp.py
torch.nn.functional.mse_loss()	torch_mse_loss()	test_mse_loss.py
torch.unsqueeze()	torch_unsqueeze()	test_unsqueeze.py
torch.squeeze()	torch_squeeze()	test_squeeze.py
torch.full()	torch_full()	test_full.py
torch.sqrt()	torch_sqrt()	test_sqrt.py

The table 1, table 2, table3, table4 are simple documentations to introduce the correspondence of the modules and functions. With this correspondence, you could search the torch documentation for a detailed introduction of each functions and modules. The test files are inside the folder ./learning_code_tf/test/.

Table 2. Functions Table2.

PyTorch	TensorFlow	TestFile
torch.Tensor.float()	torch_tensor_float()	test_float.py
torch.Tensor.long()	torch_tensor_long()	test_long.py
torch.Tensor.expand()	torch_tensor_expand()	test_expand.py
torch.Tensor.expand_as()	torch_tensor_expand_as()	test_expand_as.py
torch.triu()	torch_triu()	test_triu.py
torch.round()	torch_round()	test_round.py
torch.meshgrid()	torch_meshgrid()	test_meshgrid.py
torch.sum()	torch_sum()	test_sum.py
torch.cumprod()	torch_cumprod()	test_cumprod.py
torch.randn()	torch_randn()	test_randn.py
torch.randperm()	torch_randperm()	test_randperm.py
torch.randn_like()	torch_randn_like()	test_randn_like.py
torch.zeros_like()	torch_zeros_like()	test_zeros_like.py
torch.full_like()	torch_full_like()	test_full_like.py
torch.from_numpy()	torch_from_numpy()	
torch.exp()	torch_exp()	test_exp.py
torch.flip()	torch_flip()	test_flip.py
torch.randint()	torch_randint()	test_randint.py
torch.Tensor.transpose()	torch_tensor_transpose()	test_transpose.py
torch.Tensor.clone()	torch_tensor_clone()	test_clone.py
torch.Tensor.masked_fill()	torch_tensor_masked_fill()	test_masked_fill.py
torch.nn.functional.pad()	nn_functional_pad()	test_nn_functional_pad.py
torch.repeat_interleave()	torch_repeat_interleave()	test_repeat_interleave.py
torch.Tensor.detach()	torch_tensor_detach()	test_detach.py
torch.std()	torch_std()	test_std.py
torch.tensor.repeat()	torch_tensor_repeat()	test_repeat.py
torch.unravel_index()	torch_unravel_index()	test_unravel_index.py
torch.register_buffer()	torch_register_buffer()	test_register_buffer.py
torch.split()	torch_split()	test_split.py
torch.rand()	torch_rand()	test_rand.py
torch.vmap()	torch_vmap()	
torch.func.stack_module_state()	torch_func_stack_module_state()	
torch.func.functional_call()	torch_func_functional_call()	test_func_functional_call.py
torch.nn.functional.grid_sample()	torch_nn_functional_grid_sample()	test_grid_sample.py
torch.nn.init.normal_()	torch_nn_init_normal_()	test_nn_init_normal.py
torch.nn.init.zeros_()	torch_nn_init_zeros_()	test_nn_init_zeros.py
torch.nn.init.ones_()	torch_nn_init_ones_()	test_nn_init_ones.py
torch.Tensor.requires_grad_()	torch_tensor_requires_grad_()	test_requires_grad.py

2 A literature review of potential methods for solving the problem.

Policy Architecture Used In All Experiments

MLP. For most of the experiments, we use a Multi-layer Perceptron (MLP) with two-layer residual connection as the policy head. For diffusion-based policies, we also use a small MLP encoder for the state input and another small MLP with sinusoidal positional encoding for the denoising timestep

Table 3. Modules, Optimizers, and Gradients Table.

PyTorch	TensorFlow	TestFile
torch.nn.Mish	nn_Mish	
torch.nn.ReLU	nn_ReLU	
torch.nn.Dropout	nn_Dropout	test_Dropout.py
torch.nn.Linear	nn_Linear	test_nn_Linear.py
torch.nn.Parameter	nn_Parameter	
torch.nn.Sequential	nn_Sequential	test_nn_Sequential.py
torch.nn.ModuleList	nn_ModuleList	test_ModuleList.py
torch.nn.Embedding	nn_Embedding	test_embedding.py
torch.nn.LayerNorm	nn_LayerNorm	test_LayerNorm.py
torch.nn.MultiheadAttention	nn_MultiheadAttention	test_MultiheadAttention.py
torch.nn.TransformerDecoderLayer	nn_TransformerDecoderLayer	
torch.nn.TransformerDecoder	torch_nn_TransformerDecoder	
torch.nn.TransformerEncoderLayer	nn_TransformerEncoderLayer	
torch.nn.TransformerEncoder	nn_TransformerEncoder	
torch.optim.Adam	torch_optim_Adam	test_Adam.py
torch.optim.AdamW	torch_optim_AdamW	test_AdamW.py

Table 4. Custom and Modified Functions Table.

PyTorch	TensorFlow
CosineAnnealingWarmupRestarts	tf_CosineAnnealingWarmupRestarts
torch.distributions.normal.Normal	Normal
torch.distributions.independent.Independent	Independent
torch.distributions.Categorical	Categorical
torch.distributions.MixtureSameFamily	MixtureSameFamily

input. Their output features are then concatenated before being fed into the MLP head. Diffusion Policy, proposed by Chi et al. (2024b)[3], does not use MLP as the diffusion architecture, but we find it delivers comparable (or even better) pre-training performance compared to UNet.

Transformer.[18] For comparing to other policy parameterizations in Section 5.3, we also consider Transformer as the policy architecture for the Gaussian and GMM baselines. We consider decoder only. No dropout is used. A learned positional embedding for the action chunk is the sequence into the decoder.

UNet.[15] For comparing to other policy parameterizations in Section 5.3, we also consider UNet (Ronneberger et al., 2015) as a possible architecture for DP. We follow the implementation from Chi et al. (2024b) that uses sinusoidal positional encoding for the denoising timestep input, except for using a larger MLP encoder for the observation input in each convolutional block. We find this modification helpful in more challenging tasks.

ViT.[7] For pixel-based experiments in Section 5.3 we use Vision-Transformer(ViT)-based image encoder introduced by Hu et al. (2023) before an MLP head. Proprioception input is appended to each channel of the image patches. We also follow (Hu et al., 2023) and use a learned spatial embedding for the ViT output to greatly reduce the number of features, which are then fed into the downstream MLP head.

Descriptions of Diffusion-based RL Algorithm Baselines

DRWR[13]: This is a customized reward-weighted regression (RWR) algorithm Peters and Schaal (2007) that fine-tunes a pre-trained DP with a supervised objective with higher weights on actions that lead to higher reward-to-go r.

The reward is scaled with β and the exponentiated weight is clipped at $w_{\max}$. The policy is updated with experiences collected with the current policy (no buffer for data from previous iteration) and a replay ratio of N_θ. No critic is learned.

$$\mathcal{L}_\theta = \mathbb{E}^{\bar{\pi}_\theta, \varepsilon_t}\left[\min\left(e^{\beta r_t}, w_{\max}\right)\left\|\varepsilon_t - \varepsilon_\theta\left(a_t^0, s_t, k\right)\right\|^2\right]$$

DAWR[11]: This is a customized advantage-weighted regression (AWR) algorithm Peng et al. (2019) that builds on DRWR but uses TD-bootstrapped Sutton and Barto (2018) advantage estimation instead of the higher-variance reward-to-go for better training stability and efficiency. DAWR (and DRWR) can be seen as approximately optimizing (4.2) with a Kullback-Leibler (KL) divergence constraint on the policy Peng et al. (2019); Black et al. (2023)[2].

The advantage is scaled with β and the exponentiated weight is clipped at $w_{\max}$. Unlike DRWR, we follow (Peng et al., 2019) and trains the actor in an off-policy manner: recent experiences are saved in a replay buffer $\mathcal{D}$, and the actor is updated with a replay ratio of N_θ.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}, \varepsilon_t}\left[\min\left(e^{\beta \hat{A}_\phi\left(s_t, a_t^0\right)}, w_{\max}\right)\left\|\varepsilon_t - \varepsilon_\theta\left(a_t^0, s_t, k\right)\right\|^2\right].$$

The critic is updated less frequently (we find diffusion models need many gradient updates to fit the actions) with a replay ratio of N_ϕ.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\left\|\hat{A}_\phi\left(s_t, a_t^0\right) - A\left(s_t, a_t^0\right)\right\|^2\right]$$

where A is calculated using TD(λ), with λ as λ_{DAWR} and the discount factor γ_{ENV} .

DIPO (Yang et al., 2023)[21] : This baseline applies "action gradient" that uses a learned state-action Q function to update the actions saved in the replay buffer, and then has DP fitting on them without weighting. Similar to DAWR, recent experiences are saved in a replay buffer $\mathcal{D}$. The actions ($k = 0$) in the buffer are updated for M_{DIPO} iterations with learning rate α_{DIPO} .

$$a_t^{m+1,k=0} = a_t^{m,k=0} + \alpha_{\text{DIPO}} \nabla_\phi \hat{Q}_\phi\left(s_t, a_t^{m,k=0}\right), m = 0, \ldots, M_{\text{DIPO}} - 1$$

The actor is then updated with a replay ratio of N_θ.

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[\left\|\varepsilon_t - \varepsilon_\theta\left(a_t^{M_{\text{DIPO}},k=0}, s_t, k\right)\right\|^2\right].$$

The critic is trained to minimize the Bellman residual with a replay ratio of N_ϕ. Double Q-learning is also applied.

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|\left(R_t + \gamma_{\text{ENV}} \hat{Q}_\phi\left(s_{t+1}, \bar{\pi}_\theta\left(a_{t+1}^{k=0} \mid s_{t+1}\right)\right)\right) - \hat{Q}_\phi\left(s_t, a_t^{m=0,k=0}\right)\|^2\right]$$

IDQL (Hansen-Estruch et al., 2023) [6]: This baseline learns a state-action Q function and state V function to choose among the sampled actions from DP. DP fits on new samples without weighting. Again recent experiences are saved in a replay buffer $\mathcal{D}$. The state value function is updated to match the expected Q value with an expectile loss, with a replay ratio of N_ψ.

$$\mathcal{L}_\psi = \mathbb{E}^{\mathcal{D}}\left[\left|\tau_{\text{IDQL}} - \mathbb{1}\left(\hat{Q}_\phi\left(s_t, a_t^0\right) < \hat{V}_\psi^2\left(s_t\right)\right)\right|\right].$$

The value function is used to update the Q function with a replay ratio of N_ϕ.

$$\mathcal{L}_{\phi} = \mathbb{E}^{\mathcal{D}}\left[\| \left(R_t + \gamma_{\mathrm{ENV}} \hat{V}_{\psi}\left(s_{t+1}\right) - \hat{Q}_{\phi}\left(s_t, a_t^0\right) \|^2\right]$$

The actor fits all sampled experiences without weighting, with a replay ratio of N_θ.

$$\mathcal{L}_{\theta} = \mathbb{E}^{\mathcal{D}}\left[\left\|\varepsilon_t - \varepsilon_\theta\left(a_t^0, s_t, k\right)\right\|^2\right].$$

At inference time, M_{IDQL} actions are sampled from the actor. For training, Boltzmann exploration is applied based on the difference between Q value of the sampled actions and and the V value at the current state. For evaluation, the greedy action under Q is chosen.

DQL (Wang et al., 2022)[20]: This baseline learns a state-action Q function and backpropagates the gradient from the critic through the entire actor (with multiple denoising steps), akin to the usual Q-learning. Again recent experiences are saved in a replay buffer $\mathcal{D}$. The actor is then updated using both a supervised loss and the value loss with a replay ratio of N_θ.

$$\mathcal{L}_{\theta} = \mathbb{E}^{\mathcal{D}}\left[\left\|\varepsilon_t - \varepsilon_\theta\left(a_t^0, s_t, k\right)\right\|^2 - \alpha_{\mathrm{DQL}} \hat{Q}_{\phi}\left(s_t, \bar{\pi}_\theta\left(a_t^0 \mid s_t\right)\right)\right],$$

where α_{DQL} is a weighting coefficient. The critic is trained to minimize the Bellman residual with a replay ratio of N_ϕ. Double Q-learning is also applied.

$$\mathcal{L}_{\phi} = \mathbb{E}^{\mathcal{D}}\left[\| \left(R_t + \gamma_{\mathrm{ENV}} \hat{Q}_{\phi}\left(s_{t+1}, \bar{\pi}_\theta\left(a_{t+1}^0 \mid s_{t+1}\right)\right) - \hat{Q}_{\phi}\left(s_t, a_t^0\right) \|^2\right]$$

QSM (Psenka et al., 2023)[14]: This baselines learns a state-action Q function, and then updates the actor by aligning the score of the diffusion actor with the gradient of the Q function. Again recent experiences are saved in a replay buffer $\mathcal{D}$. The critic is trained to minimize the Bellman residual with a replay ratio of N_ϕ. Double Q-learning is also applied.

$$\mathcal{L}_{\phi} = \mathbb{E}^{\mathcal{D}}\left[\| \left(R_t + \gamma_{\mathrm{ENV}} \hat{Q}_{\phi}\left(s_{t+1}, \bar{\pi}_\theta\left(a_{t+1}^0 \mid s_{t+1}\right)\right) - \hat{Q}_{\phi}\left(s_t, a_t^0\right) \|^2\right].$$

The actor is updated as follows with a replay ratio of N_θ.

$$\mathcal{L}_{\theta} = \mathbb{E}^{\mathcal{D}}\left[\left\|\alpha_{\mathrm{QSM}} \nabla_a \hat{Q}_{\phi}\left(s_t, a_t\right) - \left(-\varepsilon_\theta\left(a_t^0, s_t, k\right)\right)\right\|^2\right]$$

where α_{QSM} scales the gradient. The negative sign before ε_θ is from taking the gradient of the mean μ in the denoising process. F. 4 DESCRIPTIONS OF RL FINE-TUNING ALGORITHM BASELINES IN SECTION 5.2

Descriptions of RL Fine-tuning Algorithm Baselines

In this subsection, we detail the baselines RLPD, Cal-QL, and IBRL. All policies π_θ are parameterized as unimodal Gaussian.

RLPD (Ball et al., 2023)[1]: This baseline is based on Soft Actor Critic (SAC, Haarnoja et al. (2018)[5]) - it learns an entropy-regularized state-action Q function, and then updates the actor by maximizing the Q function w.r.t. the action.

A replay buffer $\mathcal{D}$ is initialized with offline data, and online samples are added to $\mathcal{D}$. Each gradient update uses a batch of mixed 50/50 offline and online data. An ensemble of N_{critic} critics is used, and at each gradient step two critics are randomly chosen. The critics are trained to minimize the Bellman residual with replay ratio N_ϕ :

$$\mathcal{L}_{\phi} = \mathbb{E}^{\mathcal{D}}\left[\| \left(R_t + \gamma_{\mathrm{ENV}} \hat{Q}_{\phi'}\left(s_{t+1}, \pi_\theta\left(a_{t+1} \mid s_{t+1}\right)\right) - \hat{Q}_{\phi}\left(s_t, a_t\right) \|^2\right].$$

The target critic parameter ϕ' is updated with delay. The actor minimizes the following loss with a replay ratio of 1 :

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[-\hat{Q}_\phi\left(s_t, a_t\right) + \alpha_{\text{ent}} \log \pi_\theta\left(a_t \mid s_t\right)\right]$$

where α_{ent} is the entropy coefficient (automatically tuned as in SAC starting at 1).

Cal-QL (Nakamoto et al., 2024)[9]: This baseline trains the policy μ and the action-value function Q^μ in an offline phase and then an online phase. During the offline phase only offline data is sampled for gradient update, while during the online phase mixed 50/50 offline and online data are sampled. The critic is trained to minimize the following loss (Bellman residual and calibrated Q-learning):

$$\begin{aligned}\mathcal{L}_\phi = & \mathbb{E}^{\mathcal{D}}\left[\left\|\left(R_t + \gamma_{\text{ENV}} \hat{Q}_{\phi'}\left(s_{t+1}, \pi_\theta\left(a_{t+1} \mid s_{t+1}\right)\right)\right) - \hat{Q}_\phi\left(s_t, a_t\right)\right\|^2\right] \\ & + \beta_{\text{cql}}\left(\mathbb{E}^{\mathcal{D}}\left[\max\left(Q_\phi\left(s_t, a_t\right), V\left(s_t\right)\right)\right] - \mathbb{E}^{\mathcal{D}}\left[Q_\phi\left(s_t, a_t\right)\right]\right),\end{aligned}$$

where β_{cql} is a weighting coefficient between Bellman residual and calibration Q -learning and $V\left(s_t\right)$ is estimated using Monte-Carlo returns. The target critic parameter ϕ' is updated with delay. The actor minimizes the following loss:

$$\mathcal{L}_\theta = \mathbb{E}^{\mathcal{D}}\left[-\hat{Q}_\phi\left(s_t, a_t\right) + \alpha_{\text{ent}} \log \pi_\theta\left(a_t \mid s_t\right)\right]$$

where α_{ent} is the entropy coefficient (automatically tuned as in SAC starting at 1).

IBRL (Hu et al., 2023)[7]: This baseline first pre-trains a policy μ_ψ using behavior cloning, and for fine-tuning it trains a RL policy π_θ initialized as μ_ψ. During fine-tuning recent experiences are saved in a replay buffer $\mathcal{D}$. An ensemble of N_{critic} critics is used, and at each gradient step two critics are randomly chosen. The critics are trained to minimize the Bellman residual with replay ratio N_ϕ :

$$\mathcal{L}_\phi = \mathbb{E}^{\mathcal{D}}\left[\|\left(R_t + \gamma_{\text{ENV}}^{a' \in \left\{a^{IL}, a^{RL}\right\}} \max_{\phi'}\left(s_{t+1}, a'\right) - \hat{Q}_\phi\left(s_t, a_t\right)\|^2\right]$$

where $a^{IL} = \mu_\psi\left(s_{t+1}\right)$ (no noise) and $a^{RL} \sim \pi_{\theta'}\left(s_{t+1}\right)$, and $\pi_{\theta'}$ is the target actor. The target critic parameter ϕ' is updated with delay. The actor minimizes the following loss with a replay ratio of 1 :

$$\mathcal{L}_\theta = -\mathbb{E}^{\mathcal{D}}\left[\hat{Q}_\phi\left(s_t, a_t\right)\right].$$

The target actor parameter θ' is also updated with delay.

Other Related Works

Some established offline-to-online RL methods like: Wang, Shenzhi, et al. "Train once, get a family: State-adaptive balances for offline-to-online reinforcement learning." [19].

Furthermore, Zhang, Haichao, Wei Xu, and Haonan Yu propose "Policy Expansion for Bridging Offline-to-Online Reinforcement Learning."[22]

3 Propose your choice of methods and the reason for the choice.

I will try to apply Soft Actor Critic(SAC) algorithms to the Diffusion Policy Markov Decision Process.

4 Present the answers of the question in clear and understandable language.

4.1 Part1_Q1: Replicate results with Tensorflow Agent. Test the code with PYTEST.

4.1.1 The replication of this paper in TensorFlow including the main algorithm and the following papers(Baselines):

4.1.2 Transformed Files(means finished):* The below are the files that are changed when we transfer from Tensorflow framework to Pytorch framework:

- agent
  ./agent/dataset/sequence.py(*)
  ./agent/dataset/d3il_dataset/base_dataset.py(*)
  ./agent/dataset/d3il_dataset/aligning_dataset.py(*)
  ./agent/dataset/d3il_dataset/avoiding_dataset.py(*)
  ./agent/dataset/d3il_dataset/pushing_dataset.py(*)
  ./agent/dataset/d3il_dataset/sorting_dataset.py(*)
  ./agent/dataset/d3il_dataset/stacking_dataset.py(*)
  ./agent/eval/eval_agent.py(*)
  ./agent/eval/eval_diffusion_agent.py(*)
  ./agent/eval/eval_diffusion_img_agent.py(*)
  ./agent/eval/eval_gaussian_agent.py(*)
  ./agent/eval/eval_gaussian_img_agent.py(*)
  ./agent/finetune/train_agent.py(*)
  ./agent/finetune/train_awr_diffusion_agent.py(*)
  ./agent/finetune/train_calql_agent.py(*)
  ./agent/finetune/train_dipo_diffusion_agent.py(*)
  ./agent/finetune/train_dql_diffusion_agent.py(*)
  ./agent/finetune/train_ibrl_agent.py(*)
  ./agent/finetune/train_idql_diffusion_agent.py(*)
  ./agent/finetune/train_ppo_agent.py(*)
  ./agent/finetune/train_ppo_diffusion_agent.py(*)
  ./agent/finetune/train_ppo_diffusion_img_agent.py(*)
  ./agent/finetune/train_ppo_exact_diffusion_agent.py(*)
  ./agent/finetune/train_ppo_gaussian_agent.py(*)
  ./agent/finetune/train_ppo_gaussian_img_agent.py(*)
  ./agent/finetune/train_qsm_diffusion_agent.py(*)
  ./agent/finetune/train_rlpd_agent.py(*)
  ./agent/finetune/train_rwr_diffusion_agent.py(*)
  ./agent/finetune/train_sac_agent.py(*)
  ./agent/pretrain/train_agent.py(*)
- model
  ./model/common/critic.py(*)
  ./model/common/gaussian.py(*)
  ./model/common/gmm.py(*)
  ./model/common/mlp_gaussian.py(*)
  ./model/common/mlp_gmm.py(*)
  ./model/common/mlp.py(*)
  ./model/common/modules.py(*)
  ./model/common/transformer.py(*)
  ./model/common/vit.py(*)
  ./model/diffusion/diffusion_awr.py(*)
  ./model/diffusion/diffusion_dipo.py(*)
  ./model/diffusion/diffusion_dql.py(*)

./model/diffusion/diffusion_idql.py(*)
./model/diffusion/diffusion_ppo_exact.py(*)
./model/diffusion/diffusion_ppo.py(*)
./model/diffusion/diffusion_qsm.py(*)
./model/diffusion/diffusion_rwr.py(*)
./model/diffusion/diffusion_vpg.py(*)
./model/diffusion/diffusion.py(*)
./model/diffusion/eta.py(*)
./model/diffusion/exact_likelihood.py
./model/diffusion/mlp_diffusion.py(*)
./model/diffusion/modules.py(*)
./model/diffusion/sampling.py(*)
./model/diffusion/sde_lib.py(*)
./model/diffusion/unet.py(*)
./model/rl/gaussian_awr.py(*)
./model/rl/gaussian_calql.py(*)
./model/rl/gaussian_ibrl.py
./model/rl/gaussian_ppo.py(*)
./model/rl/gaussian_rlpd.py
./model/rl/gaussian_rwr.py(*)
./model/rl/gaussian_sac.py(*)
./model/rl/gaussian_vpg.py(*)
./model/rl/gmm_ppo.py(*)
./model/rl/gmm_vpg.py(*)

- env and util
  ./env/gym_utils/furniture_normalizer.py
  ./env/gym_utils/wrapper/furniture.py
  ./util/scheduler.py(*)

4.1.3 *Compare the replicated results with the given results in the paper.*

4.1.4 Technical Details. **The different random number generator and initialization** The random number generators adopted are different for TensorFlow and PyTorch. Historically, PyTorch had only two pseudorandom number generator implementations: Mersenne Twister for CPU and Nvidia's cuRAND Philox for CUDA https://pytorch.org/blog/torchcsprng-release-blog/.

I replicate the Kaiming initialization in the TensorFlow as the default one used in the original paper with PyTorch.

Serialize get_config() and from_config()

PyTorch and TensorFlow

Tensors in PyTorch is NCHW(batch_dim * channel_dim * height_dim * width_dim).

TensorFlow uses NHWC(batch_dim * height_dim * width_dim * channel_dim).

4.2 Q2

From your experience of replication of their code, what are the parts that are unclear in the paper?

The optimizer is not detailed in the paper, which is [8].

Inside the files of diffusion_ppo.py, gaussian_ppo.py: The PPO diffusions should refer to Eqn. 2 of [17] . It gives a reward for maximizing probability of teacher policy's action with current policy.A ctions are chosen along trajectory induced by current policy.

Inside the diffusion.py file: DDIM parameters In DDIM paper [16]. alpha is alpha_cumprod in DDPM [10].

Inside the unet.py file: FiLM modulation [12] predicts per-channel scale and bias

Inside the reward_scaling.py file: The Reward Scaling: To balance actor and critic losses, the rewards are divided through by the standard deviation of a rolling discounted sum of the rewards (without subtracting and re-adding the mean) . Reference: [4]

4.3 Q3

Can you please write a note explaining in more details the algorithm and the parts that are missing in the paper?

The algorithm description is further given in the updated version of the paper as Algorithm1.

Markov Decision Process. Markov Decision Process (MDP) [1] $\mathcal{M}_{\text{Env}} := (\mathcal{S}, \mathcal{A}, P_0, P, R)$ with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution P_0, transition probabilities P, and reward R.

At each timestep t, the agent (e.g., robot) observes the state $s_t \in \mathcal{S}$, takes an action $a_t \sim \pi(a_t \mid s_t) \in \mathcal{A}$, transitions to the next state s_{t+1} according to $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$ while receiving the reward $R(s_t, a_t)$[2]. Fixing the MDP $\mathcal{M}_{\text{ENV}}$, we let $\mathbb{E}^{\pi}$ (resp. $\mathbb{P}^{\pi}$) denote the expectation (resp. probability distribution) over trajectories, $\tau = (s_0, a_0, \ldots, s_T, a_T)$ with length $T+1$, with initial state distribution $s_0 \sim P_0$ and transition operator P. We aim to train a policy to optimize the cumulative reward, discounted by a function $\gamma(\cdot)$, such that the agent receives:

$$\mathcal{J}(\pi_\theta) = \mathbb{E}^{\pi_\theta, P_0}\left[\sum_{t \geq 0} \gamma(t) R(s_t, a_t)\right].$$

Policy optimization. The policy gradient method (e.g., REINFORCE [85]) allows for improving policy performance by approximating the gradient of this objective w.r.t. the policy parameters:

$$\nabla_\theta \mathcal{J}(\pi_\theta) = \mathbb{E}^{\pi_\theta, P_0}\left[\sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t) r_t(s_t, a_t)\right], \quad r_t(s_t, a_t) := \sum_{\tau \geq t} \gamma(\tau) R(s_\tau, a_\tau),$$

```
Algorithm 1: DPPO
1  Pre-train diffusion policy $\pi_\theta$ with offline dataset $\mathcal{D}_{\text{off}}$ using BC loss $\mathcal{L}_{\text{BC}}(\theta)$ Eq. (C.1).
2  Initialize value function $V_\phi$.
3  for iteration = 1, 2, ... do
4      Initialize rollout buffer $\mathcal{D}_{\text{itr}}$ .
5      $\pi_{\theta_{\text{old}}} = \pi_\theta$.
6      for environment = 1, 2, ..., N in parallel do
7          Initialize state $\bar{s}_{\bar{t}(0,K)} = \left(s_0, a_0^K\right)$ in $\mathcal{M}_{\text{DP}}$.
8          for environment step t = 1, ..., T, denoising step k = K - 1, ..., 0 do
9              Sample the next denoised action $\bar{a}_{\bar{t}(t,k)} = a_t^k \sim \pi_{\theta_{\text{old}}}$.
10             if $k = 0$ then
11                 then Run $a_t^0$ in the environment and observe $\bar{R}_{\bar{t}(t,0)}$ and $\bar{s}_{\bar{t}(t+1,K)}$.
12             else
13                 Set $\bar{R}_{\bar{t}(t,k)} = 0$ and $\bar{s}_{\bar{t}(t,k-1)} = \left(s_t, a_t^k\right)$.
14             Add $\left(k, \bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}, \bar{R}_{\bar{t}(t,k)}\right)$ to $\mathcal{D}_{\text{itr}}$.
15     Compute advantage estimates $A^{\pi_{\theta_{\text{old}}}}\left(s_{\bar{t}(t,k=0)}, a_{\bar{t}(t,k=0)}\right)$ for $\mathcal{D}_{\text{itr}}$ using GAE Eq. (C.2).
16     for update = 1, 2,..., num_update    ▷ Based on replay ratio $N_\theta$ do
17         for minibatch = 1, 2, ..., B do
18             Sample $\left(k, \bar{s}_{\bar{t}(t,k)}, \bar{a}_{\bar{t}(t,k)}, \bar{R}_{\bar{t}(t,k)}\right)$ and $A^{\pi_{\theta_{\text{d}}}}\left(s_{\bar{t}(t,k)}, a_{\bar{t}(t,k)}\right)$ from $\mathcal{D}_{\text{itr}}$ .
19             Compute denoising-discounted advantage $\hat{A}_{\bar{t}(t,k)} = \gamma_{\text{DENOISE}}^k A^{\pi_{\theta_{\text{dat}}}}\left(s_{\bar{t}(t,0)}, a_{\bar{t}(t,0)}\right)$.
20             Optimize $\pi_\theta$ using policy gradient loss $\mathcal{L}_\theta$ Eq. (C.3).
21             Optimize $V_\phi$ using value loss $\mathcal{L}_\phi$ Eq. (C.4).
22     return converged policy $\pi_\theta$.
```

Table 5. Notations Table.

Notations	Descriptions
t	timestamps
$\mathcal{M}_{\text{Env}}$	MDP
$s \in \mathcal{S}$	states
$a \in \mathcal{A}$	actions
P_0	initial state distribution
P	transition probabilities
R	reward

where r_t is the discounted cumulative future reward from time t (more generally, r_t can be replaced by a Q-function estimator [76]), γ is the discount factor that depends on the time-step, and $\nabla_\theta \log \pi_\theta\left(a_t \mid s_t\right)$ denotes the gradient of the logarithm of the likelihood of $a_t \mid s_t$. To reduce the variance of the gradient estimation, a state-value function $\hat{V}^{\pi_\theta}\left(s_t\right)$ can be learned to approximate $\mathbb{E}\left[r_t\right]$. The estimated advantage function $\hat{A}^{\pi_\theta}\left(s_t, a_t\right) := r_t\left(s_t, a_t\right) - \hat{V}^{\pi_\theta}\left(s_t\right)$ substitutes $r_t\left(s_t, a_t\right)$.
Diffusion models. A denoising diffusion probabilistic model (DDPM) [49, 29, 71] represents a continuousvalued data distribution $p(\cdot) = p\left(x^0\right)$ as the reverse denoising process of a forward noising process $q\left(x^k \mid x^{k-1}\right)$ that iteratively adds Gaussian noise to the data. The reverse process is parameterized by a neural network

[1] More generally, we can view our environment as a Partially Observed Markov Decision Process (POMDP) where the agent's actions depend on observations o of the states s (e.g., action from pixels). Our implementation applies in this setting, but we omit additional observations from the formalism to avoid notional clutter.

[2] For simplicity, we overload $R(\cdot, \cdot)$ to denote both the random variable reward and its distribution.

$\varepsilon_\theta\left(x_k, k\right)$, predicting the added noise ε that converts x_0 to x_k [29]. Sampling starts with a random sample $x^K \sim \mathcal{N}(0, \mathrm{I})$ and iteratively generates the denoised sample:

$$x^{k-1} \sim p_\theta\left(x^{k-1} \mid x^k\right):=\mathcal{N}\left(x^{k-1} ; \mu_k\left(x^k, \varepsilon_\theta\left(x^k, k\right)\right), \sigma_k^2 \mathrm{I}\right).$$

Above, $\mu_k(\cdot)$ is a fixed function, independent of θ, that maps x^k and predicted ε_θ to the next mean, and σ_k^2 is a variance term that abides by a fixed schedule from $k=1, \ldots, K$. We refer the reader to Chan [10] for an in-depth survey.

Diffusion models as policies. Diffusion Policy (DP; see [15]) is a policy π_θ parameterized by a DDPM which takes in s as a conditioning argument, and parameterizes $p_\theta\left(a^{k-1} \mid a^k, s\right)$ as in (3.3). DPs can be trained via behavior cloning by fitting the conditional noise prediction $\varepsilon_\theta\left(a^k, s, k\right)$ to predict added noise. Notice that unlike more standard policy parameterizations such as unimodal Gaussian policies, DPs do not maintain an explicit likelihood of $p_\theta\left(a^0 \mid s\right)$. In this work, we adopt the common practice of training DPs to predict an action chunk - a sequence of actions a few time steps (denoted T_a) into the future - to promote temporal consistency. For fair comparison, our non-diffusion baselines use the same chunk size.

As observed in [16, 6] and [59], a denoising process can be represented as a multi-step MDP in which policy likelihoods can be obtained directly. We extend this formalism by embedding the Diffusion MDP into the environmental MDP, obtaining a larger "Diffusion Policy MDP" denoted $\mathcal{M}_{\mathrm{DP}}$, visualized in Fig. 3. Below, we use the notation δ to denote a Dirac distribution and $\otimes$ to denote a product distribution.

Recall the environment MDP $\mathcal{M}_{\text{Env}} := \left(\mathcal{S}, \mathcal{A}, P_0, P, R\right)$ in Section 3. The Diffusion MDP $\mathcal{M}_{\mathrm{DP}}$ uses indices $\bar{t}(t, k)=t K+(K-k-1)$ corresponding to (t, k), which increases in t but (to keep the indexing conventions of diffusion) decreases lexicographically with $K-1 \geq k \geq 0$. We write states, actions and rewards as,

$$\bar{s}_{\bar{t}(t, k)}=\left(s_t, a_t^{k+1}\right), \quad \bar{a}_{\bar{t}(t, k)}=a_t^k, \quad \bar{R}_{\bar{t}(t, k)}\left(\bar{s}_{\bar{t}(t, k)}, \bar{a}_{\bar{t}(t, k)}\right)=\begin{cases}0 & k>0 \\ R\left(s_t, a_t^0\right) & k=0\end{cases},$$

where the bar-action at $\bar{t}(t, k)$ is the action a_t^k after one denoising step. Reward is only given at times corresponding to when a_t^0 is taken. The initial state distribution is $\bar{P}^0=P_0 \otimes \mathcal{N}(0, \mathbf{I})$, corresponding to $s_0 \sim P_0$ is the initial distribution from the environmental MDP and $a_0^K \sim \mathcal{N}(0, \mathbf{I})$ independently. Finally, the transitions are

$$\bar{P}\left(\bar{s}_{\bar{t}+1} \mid \bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right)=\begin{cases}\left(s_t, a_t^k\right) \sim \delta_{\left(s_t, a_t^k\right)} & \bar{t}=\bar{t}(t, k), k>0 \\ \left(s_{t+1}, a_{t+1}^K\right) \sim P\left(s_{t+1} \mid s_t, a_t^0\right) \otimes \mathcal{N}(0, \mathbf{I}) & \bar{t}=\bar{t}(t, k), k=0\end{cases}.$$

That is, the transition moves the denoised action a_t^k at step $\bar{t}(t, k)$ into the next state when $k>0$, or otherwise progresses the environment MDP dynamics with $k=0$. The pure noise a_t^K is considered part of the environment when transitioning at $k=0$. In light of (3.3), the policy in $\mathcal{M}_{\mathrm{DP}}$ takes the form

$$\bar{\pi}_\theta\left(\bar{a}_{\bar{t}(t, k)} \mid \bar{s}_{\bar{t}(t, k)}\right)=\pi_\theta\left(a_t^k \mid a_t^{k+1}, s_t\right)=\mathcal{N}\left(a_t^k ; \mu\left(a_t^{k+1}, \varepsilon_\theta\left(a_t^{k+1}, k+1, s_t\right)\right), \sigma_{k+1}^2 \mathrm{I}\right).$$

Fortunately, (4.1) is a Gaussian likelihood, which can be evaluated analytically and is amenable to the policy gradient updates (see also [59] for an alternative derivation):

$$\nabla_\theta \overline{\mathcal{J}}\left(\bar{\pi}_\theta\right)=\mathbb{E}^{\bar{\pi}_\theta, \bar{P}, \bar{P}^0}\left[\sum_{\bar{t} \geq 0} \nabla_\theta \log \bar{\pi}_\theta\left(\bar{a}_{\bar{t}} \mid \bar{s}_{\bar{t}}\right) \bar{r}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right)\right], \quad \bar{r}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right):=\sum_{\tau \geq \bar{t}} \gamma(\tau) \bar{R}\left(\bar{s}_\tau, \bar{a}_\tau\right).$$

Evaluating the above involves sampling through the denoising process, which is the usual "forward pass" that samples actions in Diffusion Policy; as noted above, the inital state can be sampled from the enviroment via $\bar{P}^0=P_0 \otimes \mathcal{N}(0, \mathbf{I})$, where P_0 is from the environment MDP. 4.2 Instantiating DPPO with Proximal Policy Optimization

We apply Proximal Policy Optimization (PPO) [70, 18, 32, 1], a popular improvement of the vanilla policy gradient update.

Definition 4.1 (Generalized PPO). Consider a general MDP. Given an advantage estimator $\hat{A}(s, a)$, the PPO update is given by [70] the sample approximation to where ε, the clipping ratio, controls the maximum magnitude of policy change from the previous policy. We instantiate PPO in our diffusion MDP with $(s, a, t) \longleftarrow (\bar{s}, \bar{a}, \bar{t})$. Our advantage estimator takes a specific form that respects the two-level nature of the MDP: let $\gamma_{\mathrm{ENV}} \in (0,1)$ be the environment discount and $\gamma_{\mathrm{DENOISE}} \in (0,1)$ be the denoising discount. Consider the environment-discounted return:

$$\bar{r}\left(\bar{s}_{\bar{t}}, \bar{a}_{\bar{t}}\right):=\sum_{t' \geq t} \gamma_{\mathrm{ENV}}^t \bar{r}\left(\bar{s}_{\bar{t}(t', 0)}, \bar{a}_{\bar{t}(t', 0)}\right), \quad \bar{t}=\bar{t}(t, k),$$

4.4 Q4

Have you found any mistakes or errors?

(1).The learning rate scheduler seems to be adopted in a way slightly different from the original one.

(2).In line 249 , the definition of γ_{ENV}^t seems to be incorrect? It should be $\gamma_{\mathrm{ENV}}^{t'-t}$.

4.5 Q5

If you were the reviewer of this paper, would you accept or reject this paper for a major conference?

Answer: If I was the reviewer of this paper. I will give it an acceptance if the overall writing is improved to highlight the motivations behind their ideas and their own contributions of this paper. I tend to have more tolerance to new topics and concepts. This paper is relatively novel in their topics(diffusion + reinforcement learning) and solid in their experiments. That's the main reason I tend to give it an acceptance.

4.6 Q6

Could you please write a note commenting the pros and cons of this paper?

Pros

- **Experiments**: The empirical results clearly demonstrate the effectiveness of the proposed framework on real-world datasets.
  - **P1-1:** The authors report performance improvements over baselines in long-horizon and sparse reward tasks.
  - **P1-2:** Their method also leverages environment parallelization, enabling faster training compared to off-policy RL methods. This is a notable strength, as off-policy RL techniques typically do not take full advantage of parallelization, limiting the use of high-throughput simulators.
  - **P1-3:** The paper includes comprehensive comparisons with other RL fine-tuning methods, including two novel baselines of the authors' own design, and thorough ablation studies.

  - **P1-4:** The experiments, including comparisons with diffusion policies and other fine-tuning RL policies, provide valuable insights and contribute significantly to the research field.
  - **P1-5:** DPPO consistently outperforms benchmarks, particularly in tasks involving pixel-based observations and long-horizon rollouts.
  - **P1-6:** DPPO shows impressive zero-shot transfer performance from simulation to real-world robotics tasks, with a minimal sim-to-real performance gap.
  - **P1-7:** The paper provides extensive findings on fine-tuning diffusion policies via PPO, covering topics like denoising steps, network architecture, GAE variants, and the impact of expert data. These results are highly valuable for the community.
  - **P1-8:** The authors also offer an empirical explanation for why their method outperforms alternatives, focusing on structural exploration.
- **Framework Design**:
  - **P2-1:** The paper introduces a novel dual-layer MDP framework, using PPO to fine-tune diffusion policies, which enhances performance across a variety of tasks.
  - **P2-2:** Several best practices for DPPO are proposed, such as fine-tuning only the last few denoising steps and replacing some denoising steps with DDIM sampling, improving both efficiency and performance.
  - **P2-3:** Fine-tuning diffusion models with RL methods is a valuable and original contribution. The experimental results presented in this work are insightful and ground-breaking.
- **New Topics**:
  - **P3-1:** The impressive results and the fact that this paper is among the first to leverage high-throughput simulators for diffusion policy fine-tuning is a key contribution to both the robotics and machine learning communities.

Cons

- **Writing**: The writing lacks clarity and structure, making the paper difficult to follow. The authors should provide a more explicit list of contributions.
  - **C1-1:** The main sections (pages 1-10) are poorly structured, with the results presented without a clear focus, making it hard to understand the contributions and experimental outcomes.
- **Motivation**: The motivation behind the approach is not adequately explained. Readers expect a clear understanding of the intuition behind the idea, which is somewhat lacking in this paper.
  - **C2-1:** While the typical goal of fine-tuning is to improve sample efficiency and reduce online interactions, DPPO does not significantly improve sample efficiency, as its convergence speed is comparable to other online RL methods.
- **Baselines**: Some sections, such as the initial part of Section 6 and Figure 8, are redundant and unnecessary.
  - **C3-1:** Previous diffusion policy papers have already shown that diffusion models outperform GMM and Gaussian policies in capturing multi-modal distributions, making this result no longer novel.
  - **C3-2:** The experiments only compare DPPO against other diffusion-based methods and do not include comparisons with state-of-the-art Offline2Online or Sim2Real algorithms, such as Uni-O4 and O3F, missing an important opportunity for benchmarking.
  - **C3-3:** Including comparisons with established offline-to-online methods would improve the evaluation and strengthen the findings.
- **Novelty**: While the technical contribution is solid, the novelty is relatively low.
  - **C4-1:** The approach follows a well-established pattern of "applying diffusion policy to task X and fine-tuning with method Y," which has been explored in the robotics learning community.

– **C4-2:** The paper integrates a denoising process as a multi-step MDP into the environmental MDP, which is a concept already proposed in previous works. The PPO adaptation is also not particularly innovative, with similar ideas explored in GAE.
– **C4-3:** Compared to QSM, DPPO does not introduce groundbreaking concepts but instead offers empirical adjustments like fine-tuning certain denoising steps and replacing value estimation with advantage estimation.

5 Testing plans and testing results for checking your implementation is correct and your results are valid.

Testing Plans: I plan to test each function modules one by one. The test files are inside the folder "./learning_code_tf/test/" The correspondence is listed in the table 1, table 2, table3 and table4 above. I will further write more test cases and write test cases for the missing parts right now.

References

[1] Philip J. Ball, Laura Smith, Ilya Kostrikov, and Sergey Levine. 2023. Efficient Online Reinforcement Learning with Offline Data. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 1577–1594. https://proceedings.mlr.press/v202/ball23a.html
[2] Kevin Black, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. 2024. Training Diffusion Models with Reinforcement Learning. arXiv:2305.13301 [cs.LG] https://arxiv.org/abs/2305.13301
[3] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. 2024. Diffusion Policy: Visuomotor Policy Learning via Action Diffusion. arXiv:2303.04137 [cs.RO] https://arxiv.org/abs/2303.04137
[4] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. 2020. Implementation Matters in Deep Policy Gradients: A Case Study on PPO and TRPO. arXiv:2005.12729 [cs.LG] https://arxiv.org/abs/2005.12729
[5] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv:1801.01290 [cs.LG] https://arxiv.org/abs/1801.01290
[6] Philippe Hansen-Estruch, Ilya Kostrikov, Michael Janner, Jakub Grudzien Kuba, and Sergey Levine. 2023. IDQL: Implicit Q-Learning as an Actor-Critic Method with Diffusion Policies. arXiv:2304.10573 [cs.LG] https://arxiv.org/abs/2304.10573
[7] Hengyuan Hu, Suvir Mirchandani, and Dorsa Sadigh. 2024. Imitation Bootstrapped Reinforcement Learning. arXiv:2311.02198 [cs.LG] https://arxiv.org/abs/2311.02198
[8] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Skq89Scxx
[9] Mitsuhiko Nakamoto, Yuexiang Zhai, Anikait Singh, Max Sobol Mark, Yi Ma, Chelsea Finn, Aviral Kumar, and Sergey Levine. 2023. Cal-QL: Calibrated Offline RL Pre-Training for Efficient Online Fine-Tuning. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=GcEIvidYSw
[10] Alex Nichol and Prafulla Dhariwal. 2021. Improved Denoising Diffusion Probabilistic Models. arXiv:2102.09672 [cs.LG] https://arxiv.org/abs/2102.09672
[11] Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. 2019. Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning. arXiv:1910.00177 [cs.LG] https://arxiv.org/abs/1910.00177
[12] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. 2017. FiLM: Visual Reasoning with a General Conditioning Layer. arXiv:1709.07871 [cs.CV] https://arxiv.org/abs/1709.07871
[13] Jan Peters and Stefan Schaal. 2007. Reinforcement learning by reward-weighted regression for operational space control. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:11551208
[14] Michael Psenka, Alejandro Escontrela, Pieter Abbeel, and Yi Ma. 2024. Learning a Diffusion Model Policy from Rewards via Q-Score Matching. arXiv:2312.11752 [cs.LG] https://arxiv.org/abs/2312.11752
[15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. arXiv:1505.04597 [cs.CV] https://arxiv.org/abs/1505.04597
[16] Jiaming Song, Chenlin Meng, and Stefano Ermon. 2022. Denoising Diffusion Implicit Models. arXiv:2010.02502 [cs.LG] https://arxiv.org/abs/2010.02502
[17] Marcel Torne, Anthony Simeonov, Zechu Li, April Chan, Tao Chen, Abhishek Gupta, and Pulkit Agrawal. 2024. Reconciling Reality through Simulation: A Real-to-Sim-to-Real Approach for Robust Manipulation. arXiv:2403.03949 [cs.RO]

https://arxiv.org/abs/2403.03949

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[19] Shenzhi Wang, Qisen Yang, Jiawei Gao, Matthieu Gaetan Lin, Hao Chen, Liwei Wu, Ning Jia, Shiji Song, and Gao Huang. 2023. Train Once, Get a Family: State-Adaptive Balances for Offline-to-Online Reinforcement Learning. arXiv:2310.17966 [cs.LG] https://arxiv.org/abs/2310.17966

[20] Zhendong Wang, Jonathan J Hunt, and Mingyuan Zhou. 2023. Diffusion Policies as an Expressive Policy Class for Offline Reinforcement Learning. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=AHvFDPi-FA

[21] Long Yang, Zhixiong Huang, Fenghao Lei, Yucun Zhong, Yiming Yang, Cong Fang, Shiting Wen, Binbin Zhou, and Zhouchen Lin. 2023. Policy Representation via Diffusion Probability Model for Reinforcement Learning. arXiv:2305.13122 [cs.LG] https://arxiv.org/abs/2305.13122

[22] Haichao Zhang, We Xu, and Haonan Yu. 2023. Policy Expansion for Bridging Offline-to-Online Reinforcement Learning. arXiv:2302.00935 [cs.AI] https://arxiv.org/abs/2302.00935