Reinforcement Learning with Initialized Policy

Zihan Ding

April 10, 2019

Contents

Abstract

How can we apply reinforcement learning algorithms for robot control in most efficient way?

1 Background

Reinforcement learning (RL) is a topic aside from supervised learning and unsupervised learning, with strong correlations in applications like robotics control. Reinforcement learning can be illustrated typically using the Fig 1, which demonstrates the interaction learning process of RL agent with an environment.

The overall goal of RL is to maximize the expected total reward, which can be formalized in the following process. We first consider a series of states $(s_0, s_1, ..., s_t, ...)$ describing the environment changing through the interaction process of the agent and the environment, and we suppose that for each state transformation (s_t, s_{t+1}) there is a corresponding action a_t of the agent which leads to the transformation, no matter deterministically or non-deterministically (stochastically). There the state series are extended as $(s_0, a_0, s_1, a_1, ..., s_t, a_t, ...)$. And we also suppose that there is always a reward r_t corresponding with each state s_t. Then we have $(s_0, a_0, r_0, s_1, a_1, r_1, ..., s_t, a_t, r_t, ...)$. With an absorbing state, the sequence will be finite; and it will be infinite without an absorbing state. We can hence formalize the total reward as $R = \sum_{t=0}^{\infty or T} r_t$ for infinite/finite process. And the probability of the above total reward is the probability of the sequence of state transformation: $P(s_0, a_0, r_0, s_1, a_1, r_1, ..., s_t, a_t, r_t, ...)$. Therefore we want to maximize the expected total reward with the following relationship:

$$\max \mathbb{E}_{P(s_0,a_0,r_0,s_1,a_1,r_1,...,s_t,a_t,r_t,...)}[\sum_{t=0}^{\infty} r_t] \tag{1}$$

and the expectation is taken as the process of interaction is always stochastic. The maximization is taken over the parameterized policy π_θ which determines the sequential transformation probability. Now we try to specify and simplify above equation.

Assumption 1. Markov process: we take the state transformation as a markov process, and we have $P(s_{t+1}|s_t, s_{t-1}, ..., s_0) = P(s_{t+1}|s_t)$, and therefore:

$$P(s_0, a_0, r_0, s_1, a_1, r_1, ..., s_t, a_t, r_t, ...) = P(s_0, r_0)\Pi_{t=0}^{\infty}(P((s_{t+1}, r_{t+1})|(s_t, a_t, r_t))P(a_t|s_t)) \tag{2}$$

Assumption 2. Deterministic transformation process: we assume the transformation matrix described by $P((s_{t+1}, r_{t+1})|(s_t, a_t, r_t)$ is deterministic and therefore the probability equals to 1. So we have,

$$P(s_0, a_0, r_0, s_1, a_1, r_1, ..., s_t, a_t, r_t, ...) = P(s_0, r_0)\Pi_{t=0}^{\infty}P(a_t|s_t) \tag{3}$$

where the $P(s_0, r_0)$ is probability of different starting state.

We usually use logarithm in practice and denote $P(a_t|s_t)$ as the parameterized policy π_θ we want to learn. Therefore we have the gradients of optimization goal (the reward term has no relationship with policy parameters θ and neglect the constant term like probability of initial state):

$$\begin{aligned} g &= \nabla_\theta \mathbb{E}[\sum_{t=0}^{\infty} \Phi_t \log \pi_\theta(a_t|s_t)] \\ &= \mathbb{E}[\sum_{t=0}^{\infty} \Phi_t \nabla_\theta \log \pi_\theta(a_t|s_t)] \end{aligned} \tag{4}$$

where Φ_t is the term of reward/value/advantage describing how good current state is with respect to direct rewards from the environment. It have several different expressions as shown in [1]. Eq. (4) is called the policy gradient.

Deep reinforcement learning (DRL) [2] is gradient-based RL algorithms with deep neural networks as approximators, which composes the majority of present RL methods. RL can also be addressed with model-free methods like evolutionary methods [3].

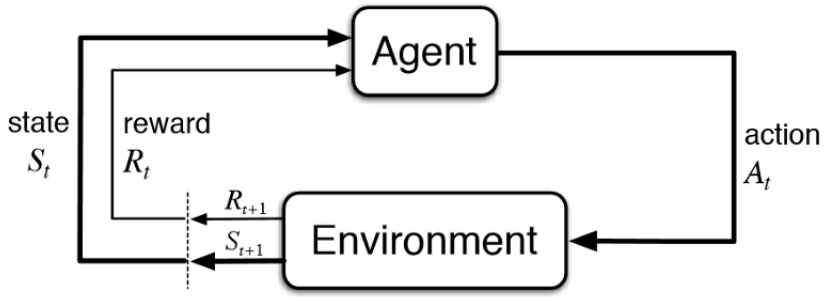

Figure 1: Reinforcement learning algorithms development.

2 Reinforcement Learning Algorithms

2.1 Discrete and Continuous Action Space

There are two types of action space for general RL problem: the continuous one and the discrete one. RL algorithms like Q-learning, Deep Q-Network can only work for discrete action space without further modifications. And algorithms like REINFORCE, Actor-Critic, DDPG, TRPO, PPO, etc can work for continuous action space, and sometimes work for discrete action space with a few proper modifications.

The discrete and continuous action space can transform into each other to some extent. A continuous action space problem can be sometimes be solved through discretization of the potential action values and applying a discrete RL algorithm; a discrete action space problem can also be handled with continuous RL algorithms if the output action values are discretized properly. However, there is always a cost for the transformation in different action spaces. If a discrete RL algorithm has to be used for a task with a continuous action space, the possible number of the discretized action values could be large to have a similar control effect as a continuous RL algorithm, which would be hard to learn in practice for the RL agent.

2.2 Deterministic and Stochastic Policy

There are two types of action choice in a RL policy, the deterministic and the stochastic. The deterministic policy is that the action values are directly determined by the outputs of neural networks. Algorithms including DQN and DDPG are deterministic policy. The stochastic policy is that the action values are sampled from a distribution parametrized by the direct outputs of neural networks, therefore the action values themselves are not directly determined by the outputs of neural networks. Algorithms including REINFORCE, Actor-Critic, TRPO and PPO are stochastic policy.

2.3 Present Algorithms

Fig. 2 shows a development of RL algorithms for recent years. We introduce some model-free RL algorithms as follows, including the REINFORCE, Actor-Critic, Q-learning, deep Q-network, deep deterministic policy gradient, trust region policy optimization, proximal policy optimization, etc.

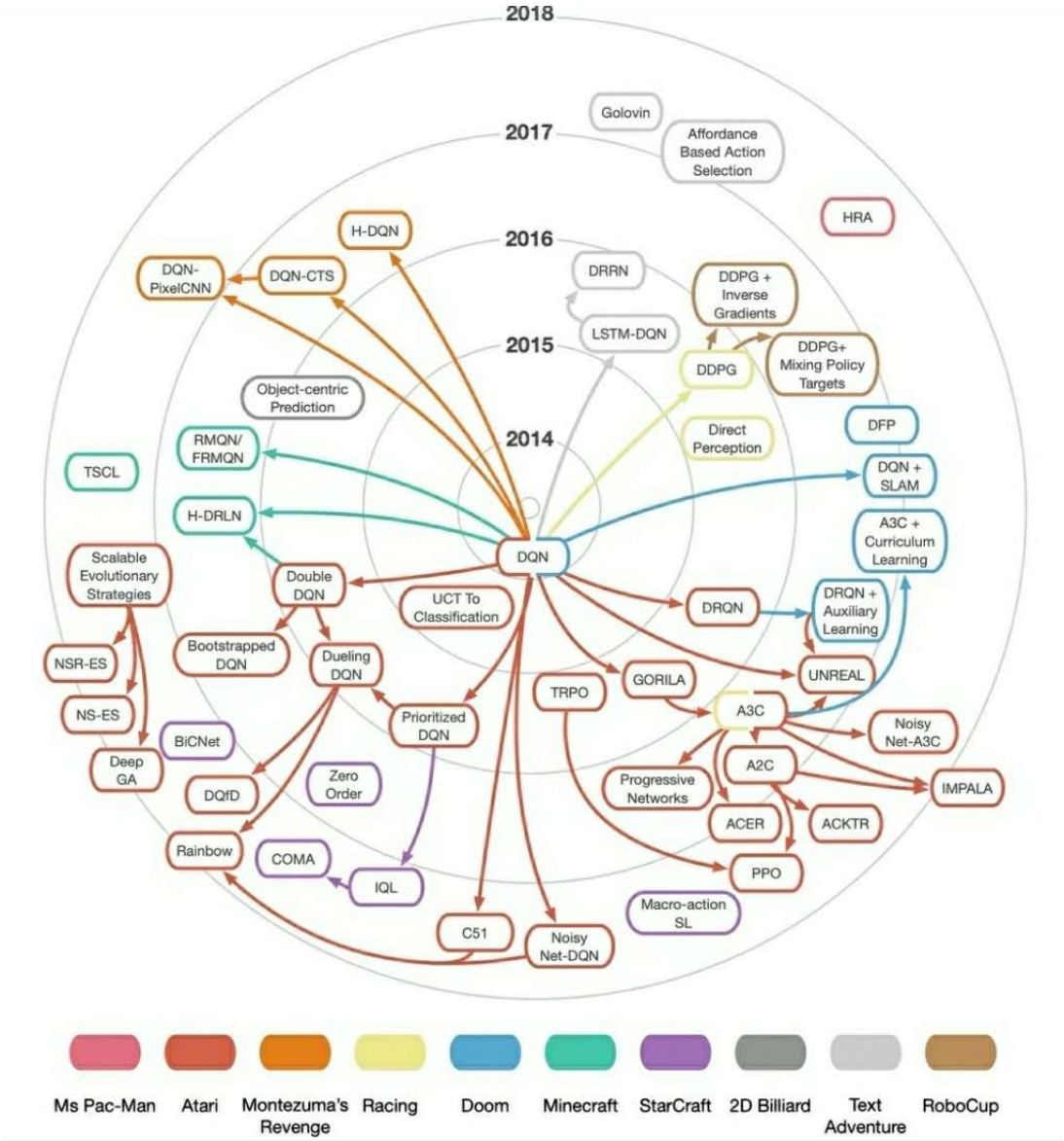

Figure 2: Reinforcement learning algorithms development.

REINFORCE

REINFORCE [4] is an algorithm using stochastic policy gradient method as in Eq.(4), where the $\Phi_t = Q^\pi(s_t, a_t)$ and it is estimated with $V^\pi(s_t) = \sum_{t'=t}^{\infty} r_{t'}$ in REINFORCE. The gradients for

updating the policy are:

$$g = \mathbb{E}[\sum_{t=0}^{\infty} \sum_{t'=t}^{\infty} r_{t'} \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{5}$$

REINFORCE algorithm applies stochastic policy and updates the policy in an on-policy way. Although the estimation of $Q^\pi(s_t, a_t)$ using $V^\pi(s_t)$ is unbiased, but it could have large variance as a Monte-Carlo approach. More advanced approach like generalized advantage estimation (GAE) [1] could be applied to alleviate this problem in REINFORCE algorithm.

Actor-Critic

The actor-critic architecture is another approach for estimating the $Q^\pi(s_t, a_t)$ in policy gradient methods. Different from $V^\pi(s_t)$ in REINFORCE, the actor-critic applies an additional component called the critic for approximating the action-value function: $Q^\omega(s, a) \approx Q^\pi(s, a)$. The estimation of the critic could be biased but with lower variance compared with the REINFORCE algorithm. The gradients for updating the policy in actor-critic algorithm are:

$$g = \mathbb{E}[\sum_{t=0}^{\infty} Q^\omega(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{6}$$

Q-learning

Q-learing is to optimize the policy with the action-value function $Q(s, a)$, via the Bellman optimality equation for optimal action-value function $Q^*(s, a)$:

$$Q^*(s_t, a_t) = \sum_{s_{t+1}} P^a_{s_t s_{t+1}} (r^{a_t}_{s_t s_{t+1}} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})) \tag{7}$$

where the $P^{a_t}_{s_t s_{t+1}}$ is the probability of state transformation from s_t to s_{t+1} with action a_t, and $r^{a_t}_{s_t s_{t+1}}$ is the reward corresponding to that. And the policy π is represented as:

$$\pi(s_t) = arg \max_{a_t} Q(s_t, a_t) \tag{8}$$

The optimal action-value function can be derived with value iteration methods. The update rule for Q-learning is:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) Q(s_t, a_t) + \alpha (r^{a_t}_{s_t s_{t+1}} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})) \tag{9}$$

where α, γ are the step-size for soft update and the discount factor respectively.

DQN

Deep Q-Network (DQN) [5][6] applies deep neural networks in Q-learning for action-value function estimation. In order to realize that, DQN formalizes a differentiable objective function and derives the gradients for updating the policy:

$$g = \nabla_\theta \mathbb{E}_{s_{t+1} \sim P(s_{t+1}|s_t, a_t)} [(Q_\theta(s_t, a_t) - Q^T_{\theta'}(s_t, a_t))^2] \tag{10}$$

where the target action-value function is represented by a target network with different parameters θ' for alleviating the problem of unstable target:

$$Q^T_{\theta'}(s_t, a_t) = r^{a_t}_{s_t s_{t+1}} + \gamma \max_{a_{t+1}} Q_{\theta'}(s_{t+1}, a_{t+1}) \tag{11}$$

Both the Q-learning and DQN work for discrete action space only. Their actions are all based on the argmax operation on the action-value function, instead of sampling from a stochastic policy.

DDPG

Deep deterministic policy gradient (DDPG) [7] is a combination of DQN and the actor-critic architecture with a deep deterministic policy (DPG) [8]. The deterministic policy is represented as $a_t = \mu_\theta(s_t) + \mathcal{N}_t$, which is different from the stochastic policy $\pi(a_t|s_t)$ in REINFORCE and actor-critic. The $\mathcal{N}_t$ is a noise term for exploration. And DDPG applies the two key points in DQN that could be effective: 1. the replay buffer and off-policy update; 2. the target networks.

The update rule for the critic in DDPG is the same as Eq.(10) in DQN. For the actor policy update in DDPG, it needs to replace the stochastic log-probability policy $\log \pi_\theta(s_t)$ in Eq.(4) with the deterministic policy $\mu_\theta(s_t)$ of probability equivalent to 1 and parameterize the action-value function $Q(s, a)$ in Eq.(4) with θ as $Q_\theta(s, a)$. By applying the chain rule, we can derive the update gradients for the actor network in DDPG:

$$g \approx \frac{1}{N} \sum_i \nabla_a Q_\theta(s_i, a) \nabla_\theta(\mu_\theta(s_i)) \tag{12}$$

where N is number of samples used for the update.

TRPO

Trust region policy optimization (TRPO) [9] applies a surrogate loss for the policy update and makes the process of policy training a constrained optimization problem to achieve monotonically improvement in policy performance. The update rule for TRPO is:

$$\begin{aligned} &\text{maximize}_\theta \mathbb{E}_{s\sim\rho_{\theta_k}, a\sim\pi_{\theta_k}}[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A_{\theta_k}(s, a)] \\ &\mathbb{E}_{s\sim\rho_{\theta_k}}[D_{KL}(\pi_{\theta_k}(\cdot|s)||\pi_\theta(\cdot|s))] \leqslant \delta \end{aligned} \tag{13}$$

where ρ_θ is the state-visitation frequencies and the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. TRPO has a more precise control about the policy update, but suffers from high computational cost because of the inverse of the Hessian matrix in the update steps.

PPO

Proximal Policy Optimization (PPO) [10] is a simplified version of TRPO algorithm for accelerating the computation process. It applies a clip optimization objective to replace the KL-divergence penalty in TRPO, which is more computationally efficient:

$$L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{14}$$

In practice, PPO algorithm usually shows more stable improvement in learning process compared with DDPG algorithm, but sometimes may learn slower.

2.4 Environment

In this work, we apply reinforcement learning algorithms on the *Reacher* task. As shown in Fig.1, the goal of *Reacher* task is to let the end position of 'reacher' agent to be as close as possible to the positions with highest rewards. The *Reacher* task is a simplified environment very close to robotic arm control in reality, and this is the reason why we choose this environment in the research. Experiments in the following sections are all based on the *Reacher* task, with slight modifications for different purposes.

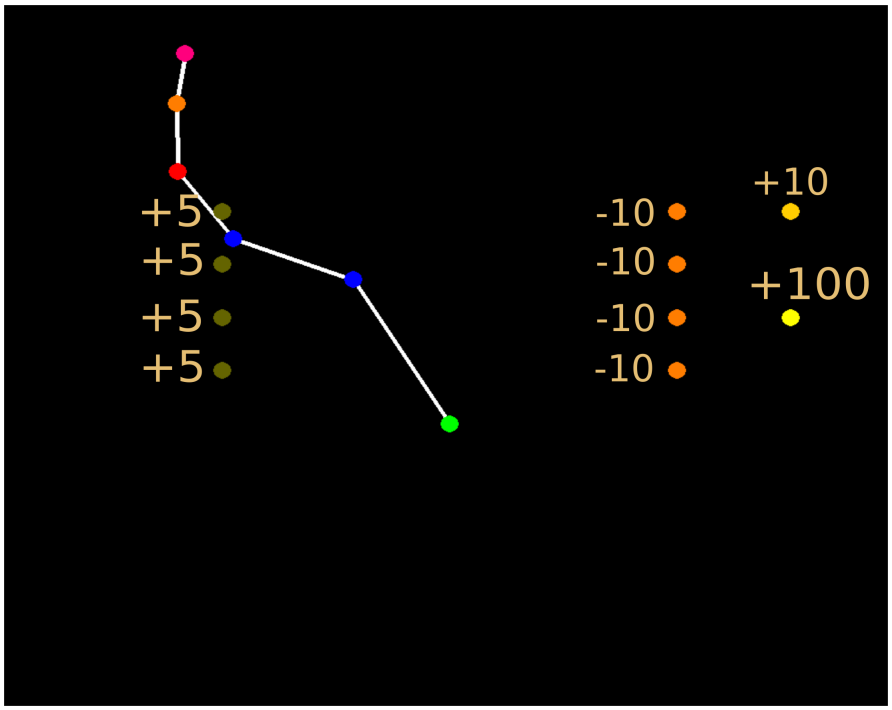

Figure 3: *Reacher* task with 5 joints and several reward/penalty positions

2.4.1 Task Specification

The task space in our model is defined to be the same 'reacher' structure with one target point at different positions. The number of joints of 'reacher' is chosen to be 3 with length between each pair of joints to be 200, 140 and 100, respectively. The screen size is 1000 and the fixed start point of the 'reacher' is at the center of the map. The target position is sampled from a random distribution within the allowed area.

2.4.2 Reward Function

The reward function is critical for a RL problem. It needs to satisfy at least two requirements:

1. the reward function should match with the objective of the task, sometimes it's biased, but it at least has preferences for the optimization towards the objective.

2. the reward function should help with the optimization process for the policy of RL, which usually means it needs to be differentiable for a neural network approximator.

There are usually two types of reward functions: the sparse reward and dense reward. A sparse reward is more close to the situations in real world tasks for human learning process, where the reward signals are rare. While in simulation learning task for RL, people tends to use the man-made dense reward functions for improving the learning efficiency. In practice, RL with sparse reward function could be hard to learn for the agent, techniques like Hindsight Experience Replay (HER) [11] can be applied to alleviate the problem when learning in multi-goal tasks like in *Reacher* with random goal positions.

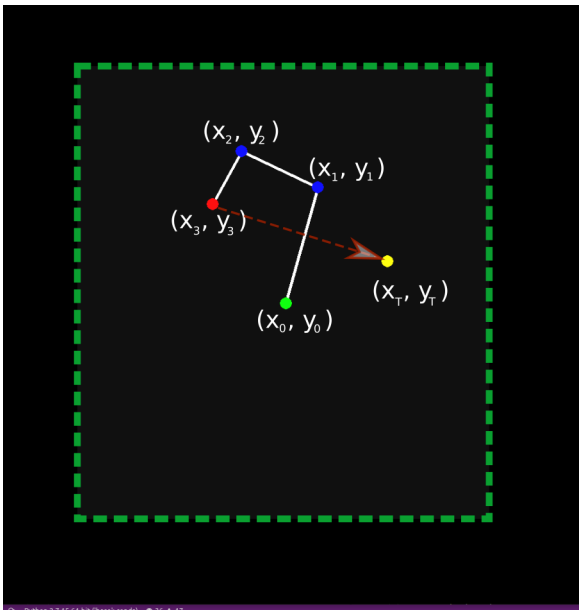

Figure 4: *Reacher* task with three joints and one target point of different possible positions. The green dashed line is the area of potential target positions in the task space.

The **dense reward** function for each step is defined as:

$$R = \frac{R_0}{\sqrt{(x_3 - x_T)^2 + (y_3 - y_T)^2} + 1}, \tag{15}$$

where $R_0 = 100$ is the maximum of reward value, so $R \in (0, 100]$. x_3, y_3 is the end position of the reacher, and x_T, y_T is the target position.

We could also use a **sparse reward** function as:

$$R = \begin{cases} R_T & \sqrt{(x_3 - x_T)^2 + (y_3 - y_T)^2} < r \\ -1 & \text{Otherwise} \end{cases} \tag{16}$$

where r is an arbitrary radius value (a precision factor) of the range we think the reacher has reached the goal, and R_T is a constant reward of it.

In the following parts of this paper, experiments are conducted with the dense reward as default.

2.4.3 Inverse Kinematics

Inverse kinematics is a inverse process of forward kinematics, to derive the joint angle values with respect to the end position. Considering the end position as a transformation function of joint angels: $\mathcal{T}(\theta, L)$, where $\{\theta_i\}$ are joint angles and $\{L_i\}$ are link lengths, the forward kinematics is:

$$(x_{end}, y_{end}) = \mathcal{T}(\theta, L) \tag{17}$$

while the inverse kinematics needs the Jacobian of end position with respect to the joint angles parameters $\{\theta_i\}$, and to reach a specific target position (x_T, y_T), the expected change of angle values

are:

$$(\Delta(x_{end}), \Delta(y_{end})) = \alpha(\frac{\partial(\mathcal{T}(\theta, \text{Ł}))}{\partial\theta})^{-1} \cdot (x_T, y_T) \tag{18}$$

where α is the step size.

And in practice, we usually use Moore–Penrose pseudo-inverse for calculation of the inverse Jacobian, which will induce approximation error in trajectories.

3 Efficient Reinforcement Learning with Demonstrations

3.1 Approaches of Demonstrations Generation

A good demonstration is a pre-collected trajectory for the agent to have relatively large rewards in the episode. By leveraging good demonstrations with proper methods, reinforcement learning can learn faster and robuster through imitating or drawing lessons from them. It could be extremely useful for training robotics in reality, where the training cost could be huge practically. There are at least three approaches for generating the demonstrations:

1. **Model-based reinforcement learning**. Model-based reinforcement learning can efficiently train a policy with small numbers of samples, like model predictive control (MPC) [12]. The learned policy can be used to initialize a model-free reinforcement learning model with supervised imitation learning.

2. Analytic-form solutions like **inverse kinematics**. For *Reacher* environment, we can analytically calculate the action values to generate a trajectory reaching the goal, called the inverse kinematics. Demonstrations can be collected in simulation with inverse kinematics. Details are shown in the below section.

3. **Human expert demonstrations**. Above two approaches can be solved in simulation, but in reality, it could be hard to neither model the real world environment or solve a analytic solution for it. The most direct way to generate good demonstrations in reality is to collect demonstrations from human experts [13]. However, this could be expensive in human-labor and time consumption, therefore the number of demonstrations from human is supposed to be smaller, compared with those demonstrations generated in simulation.

3.2 Supervised Imitation Learning with Demonstrations

The imitation learning with demonstrations can be taken as a supervised learning problem. Consider an initialization policy π_{ini} parameterized by θ with input state s and output action a in a RL setting, we have demonstrations dataset $\mathcal{D} = \{(s_i, a_i)\}$ generated from above approaches, which could be used to train the policy, with an objective as follows:

$$\min_{\theta} \frac{1}{2} \sum_{(s_i, a_i) \sim \mathcal{D}} ||a_i - \pi_{ini}(s_i)||_2^2 \tag{19}$$

This supervised learning approach to directly imitating the demonstrations is sometimes called the **behavioural cloning (BC)** or policy replacement in this paper.

DAGGER

DAGGER [14] is a more advanced no-regret iterative algorithm for imitation learning from demonstrations. It proactively choose demonstration samples that the policy has larger chances to meet

afterwards according to the previous training iterations, which makes DAGGER more effective and efficient for online imitation learning in sequential prediction problem like RL.

Variational Dropout

Another method for alleviating generalization problem in imitation learning is pre-training with variational dropout [15], instead of cloning the behavior of expert demonstrations in policy replacement (or called behavioural cloning, BC) methods. Variational dropout [16] can be taken as a more advanced method than noise injection in the weights of trained neural networks, it reduces the sensitivity of choosing the magnitude of noise.

Others

Although imitation learning can show a relatively good performance for sequential prediction problem for those samples similar to the dataset the policy is trained with, it will suffer from bad performances for those samples it never meet before, especially in a scenario like RL. Some methods provide a way to generalize demonstrations to more general scenarios in a task using framework like **dynamic movement primitive (DMP)** [17] in an analytic form, but these methods are restricted in applications with simple analytic-form descriptions like robot arm controls. And method **one-shot imitation learning** [18] uses soft attention on demonstrations to generalize model to unseen scenarios in training data. It is a meta-learning scheme to map one demonstration of one task to an effective policy for the task.

Generally, the policy learned from imitating the demonstrations, including the **BC, DAGGER, Variational Dropout and so on**, can be regarded as a good initialization for RL policy, using methods like **policy replacement or residual policy learning** described in the following sections.

3.3 Initialized Reinforcement Learning with Supervised Learning Policy – Policy Replacement

Reinforcement learning algorithms usually take a long time to achieve a good policy for a specific task. And the policy always needs to be re-trained if the settings of the task are changing, even if slightly.

There are several ways to improve the learning efficiency of reinforcement learning algorithms. One promising way is to initialize the reinforcement learning policy with a pre-trained policy, in a manner of **policy replacement**, like cloning the behaviours of demonstrations. **The pre-trained policy with supervised learning approach described in the last section is used to replace the policy in RL as an initialization.** And this is especially effective when the pre-trained policy is trained on a similar task with the one the reinforcement learning is trying to solve. Moreover, if the pre-trained policy is trained across the task space containing all possible settings of a specific task that the RL is solving, it can leverage the learning performance of RL for all tasks defined in this task space.

In our model, we try to initialize RL with sub-optimal policy pre-trained before the RL process, and the policy is trained with supervised learning with expert trajectories sampled from inverse kinematics on *Reacher* tasks.

3.3.1 Pre-trained Policy with Supervised Learning

We apply a neural network of 5 layers to train an initialization policy, with the data generated from inverse kinematics. The inputs of neural network are positions of joints and position of the target. Each hidden layer contains 100 nodes and uses ReLu as activation function. The output layer is Tanh activated with a scaling factor of 360, to make outputs range from -360 to 360 degree as a reasonable

action value of the joints. The AdamOptimizer is applied with stochastic gradient descent in the training process. The training curve is shown in Fig. 3. Note that the y-axis is transformed to be in range [-1,1] instead of [-360, 360] as the mean squared error per action value .

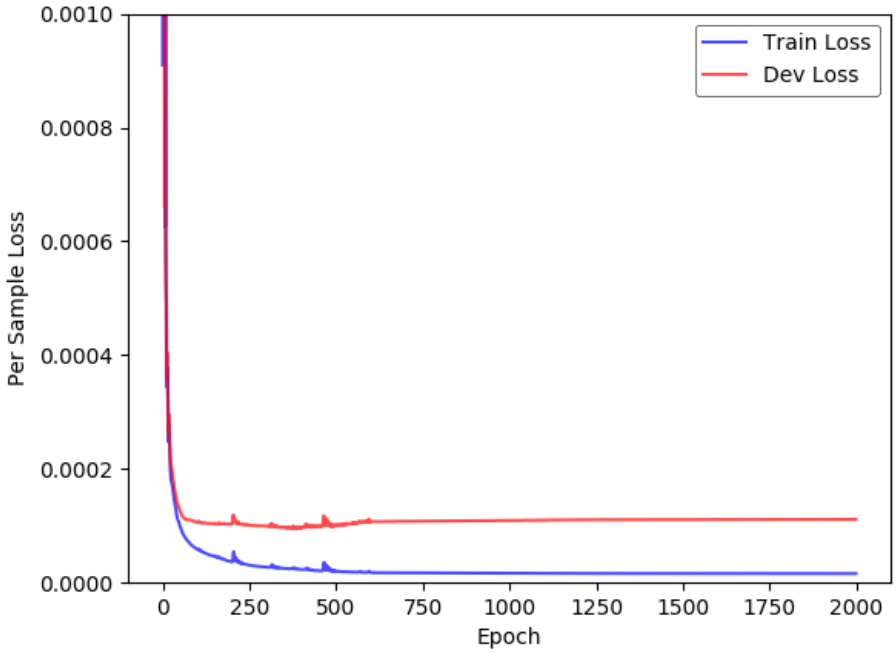

Figure 5: Training curve of supervised learning with data from inverse kinematics.

Here are several specifications about following experiments: (1). whenever there is a preheating process, the initial 600 steps in the learning curves are only for updating the critic (no actor update), so there is very small variance in learning curves when the noise applied is small; (2). as we are exploring the initialized policy for RL learning process, we only show experiments about the initial training steps (e.g. 2000 steps) for displaying the initialization effects.

3.3.2 Policy Replacement as Initialization

Policy replacement is one approach to apply the policy from supervised learning in initialization of RL. The basic idea is to use the same network structure in supervised learning and the actor in RL, so that a copy can be made from the supervised learning policy to the actor as an initialization of RL. Some tricks can be applied in this process to guarantee a performance improvement, like the preheating process.

However, potential problems exist in policy replacement approach for initialization of RL, e.g. the initialized policy is fragile in the initial training process even with small exploration noise scale, which means the initialized good policy may degrade to some worse polices (the decrease on learning curve at initial training stage) and get improved through a re-learning process. We analysed the reason for the problems and proposed some strategies for solving or alleviating it. Further analysis seen in the following subsections.

3.3.3 DDPG with Policy Replacement

General Process

We apply the DDPG algorithm for the task, and show the comparisons with or without initialization for the policy.

The DDPG algorithm contains four neural networks: the actor network, the critic network, the target-actor network, and the target-critic network. For initializing the DDPG, we define the actor network and target-actor network to have exactly the same structure as the network in supervised learning.

Initialization process of DDPG with supervised learning policy are as follows: (1). the weights of pre-trained supervised learning policy are loaded into the actor network and the target-actor are updated immediately; (2). samples are generated with initialized but frozen actor network with noise to pre-train the critic network; (3). train both the actor and critic network (also the target-actor and target-critic networks). Experiments show that the second step is non-trivial process for effective training DDPG with initialization. Details are explained in Sec. 3.3.3.

Comparison of initial learning performances of DDPG with and without initialization policy on *Reacher* task are shown in Fig. 4. The DDPG with initialization policy could significantly outperforms the one without initialization.

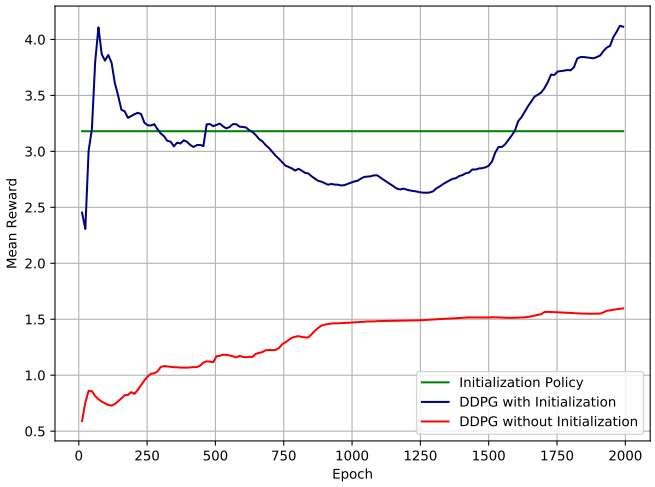

Figure 6: Comparison of initial learning performances of DDPG with/without initialization policy on *Reacher* task. The green line is the mean reward of initial policy trained with supervised learning. The blue line and red line are mean rewards of DDPG with and without initialization in initial 2000 training epochs .

There are several key components in initializing the RL with supervised learning policy affecting the performance of initialized RL (DDPG), including the scale of action noise in initialized RL, different ways of normalization in supervised learning and RL, learning rates of the actor and the critic in initialized RL, number of steps of single episode and so on. Improper settings of all above factors will affect the initialization policy to be effective for RL process. Some general intuitions about making the initialization more effective for RL are like: ensuring the inputs and outputs of the replaced policy from initialization to be as similar as possible between the supervised learning and RL, therefore applying

the same input and output normalization to make sure they are of the same ranges of value, the same initial positions for each episode in 'Reacher' and the same maximum length of single episode in supervised learning and RL, etc.

Choice of Noise

Noise is an important factor for RL learning process. As shown in Fig. 7, different noise scales affect the performance of DDPG algorithm on *Reacher* tasks greatly. For initialized policy π_i trained with expert trajectories, the optimal trajectory for the same task should be some neighboring trajectories of the initialized trajectory, which means the difference of optimal policy π^* and π_i should be within a small range Δ,

$$D_{KL}(\pi^*||\pi_i) < \Delta \tag{20}$$

This can be derived easily by $(\pi^*(a^*|s_t) - \pi_i(a^*|s_t)) < \delta$, where a^* is the optimal action for state s_t. However, if we consider another initial policy π_i' which is not as good as π_i, then $(\pi'^*(a^*|s_t) - \pi_i(a^*|s_t)) < \delta'$ and $\delta' > \delta$, also we have $D_{KL}(\pi^*||\pi_i') < \Delta'$ and $\Delta' > \Delta$. In order to handle different initialization policies like π or π', we need to apply different noise for a better learning performance.

Generally, a larger noise scale or larger probability to have a noise action will help worse policy π' to have larger chances to sample an optimal action. We can testify the case of larger probability of have a noise through ϵ-greedy policy, as shown in Fig. 8. The ϵ-greedy policy is defined as follows:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|A(s)|}, & a = a* \\ \dfrac{\epsilon}{|A(s)|}, & a \neq a* \end{cases} \tag{21}$$

In Fig. 8, the ϵ-greedy policy can be regarded as a uniform distributed noise on the possible range of action value. It shows that the noise helps the initial policy to have a larger probability to sample the optimal action a^*, when the initial policy has a relatively large difference from the optimal policy, which is the case of π' with the KL-divergence upper bound $\Delta' > \Delta$. Fig. 9 shows that for the same optimal distribution and different initial distributions, same ϵ will cause different effects for noisy policy. Generally, it shows that large ϵ value (0.8) only helps with large divergence of initial policy and optimal policy. And this is the reason to apply decayed ϵ in practice for ϵ-greedy policy.

Fig. 10 shows that when two policies have large divergence Δ', larger ϵ in will help more for the initial policy to sample optimal actions with ϵ-greedy policy. And larger ϵ generally means larger chances to have noisy actions.

Above analysis is based on ϵ-greedy policy, which represents the variety of probability to apply the noise in actions. However, the case of the variety in scale of noise has similar effects for different distributions. Therefore, we can derive some conclusions that for better initial policy, it means the smaller the divergence it is from the optimal policy, therefore it needs to apply noise with a smaller probability or apply smaller scale noise. So, how does it work in initialized RL?

In initialized RL, the cases are the same. Experiments with different initialized policy for DDPG on *Reacher* task are shown in Fig. 11, in which the relatively good initial policy π is called from expert, and the relatively bad but better than random initial policy π' is called the half-expert. Fig. 11 shows that for a bad initial policy, which has a relatively large divergence from the optimal policy, larger noise scale works better for improving the learning performance. And this results testify the above analysis of the case of ϵ-greedy policy. Note that the experiments shown in Fig. 11 are all without preheating process, which is the second step described in the general process section. However, if we take experiments on the policy initialized from a better policy, Fig. 12 shows smaller noise is better for DDPG initialized with an expert policy, which also testifies above analysis results.

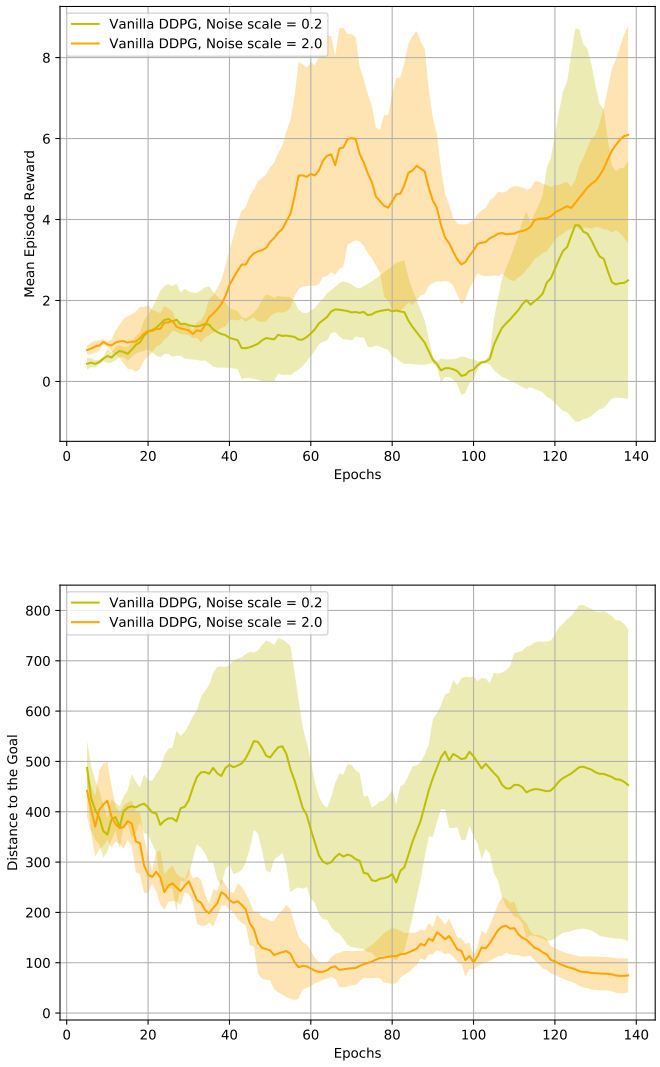

Figure 7: The effects of different noise scale for vanilla DDPG algorithm on *Reacher* tasks. The experiments are conducted with general settings. The figure on the top shows the changing of mean episode reward during training, and the one below shows the changing of distance of the reacher's end to the goal position.

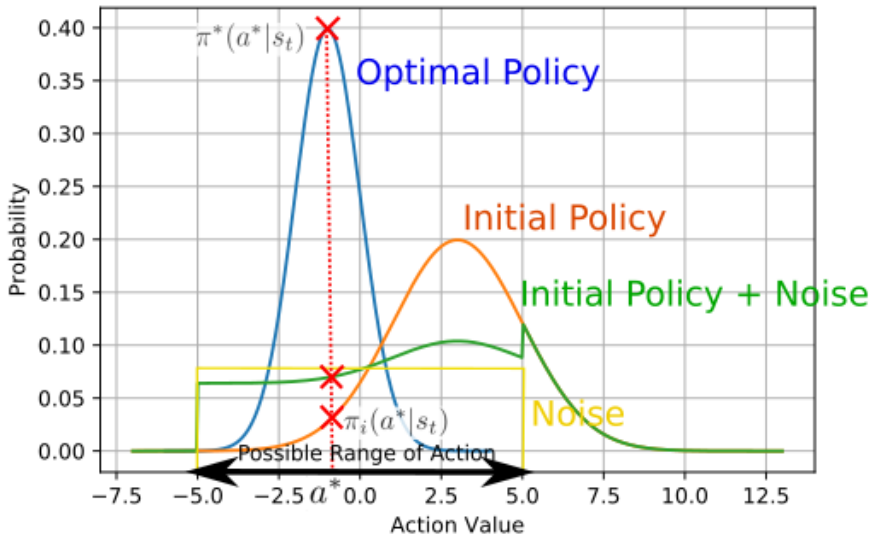

Figure 8: A figure showing why adding noise would help for exploring better actions, with noise type of ϵ-greedy policy. Suppose the initial policy and the optimal policy are Gaussian distributions with different means and variances, and the noise is a uniform distribution within specific range (range of possible action value). The figure shows how the noise of ϵ-greedy helps to increase the possibility for non-optimal policy to sample the optimal action

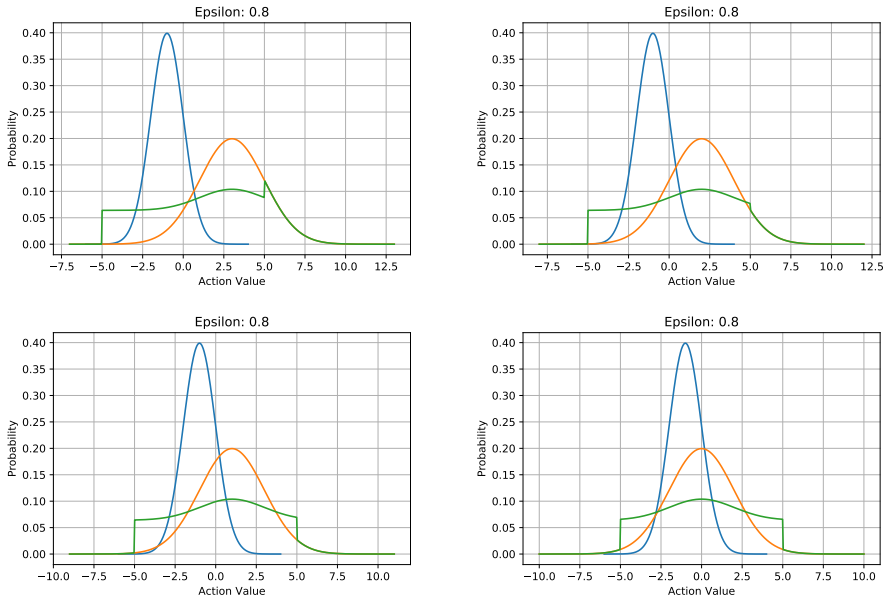

Figure 9: ϵ-greedy policy with different initial distributions and same ϵ value. It shows the same ϵ will cause different effects for noisy policy, it benefits the top two initial distribution and hurts the bottom two distributions. So large ϵ value (0.8) only helps with large divergence of initial policy and optimal policy.

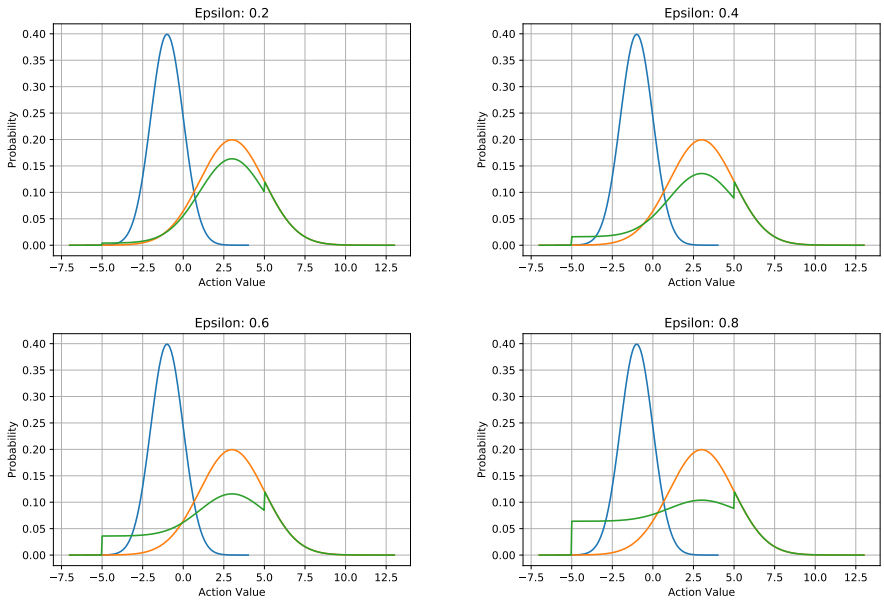

Figure 10: ϵ-greedy policy with different ϵ value and same initial and same optimal distribution. Larger ϵ value benefits more for large divergence between the optimal policy and the initial policy.

As for the learning rate settings, Fig. 13 shows a comparison of different actor learning rate for initialized DDPG with a proper noise scale. For initialized DDPG from an expert policy, we actually need only tune the parameters in the actor neural networks in a small range, which requires a small scale of actor learning rate (e.g. 10^{-4}). A larger learning rate could change the networks dramatically and also the outputs. However, when the actor learning rate is too small, it will not help with learning process as shown in Fig. 13.

Pre-train the Critic

The actor-critic scheme is important in DDPG, as the critic instructs the actor's choise of action values through evaluating the Q-value of each action. Therefore, a good critic is critical for the DDPG to show great learning performance in a task. However, in initialized policy for RL, the supervised learning with inverse kinematics could only mapping from the input states to the output actions, without any reference of the estimated value of each action. It means the critic of DDPG cannot be directly initialized with policy from supervised learning. Therefore, the role of the critic is amplified in the learning process of initialized DDPG, which is testified in experiments of this section. To solve the problem that the supervised learning policy can only initialized the actor networks, we propose a pre-training process of the critic in initialized DDPG called the 'preheating', with which the initialized policy could be more effective than without it.

The 'preheating' process is conducted as follows: after loading the weights from pre-trained policy to the actor networks in DDPG, we sample from the frozen actor to generate near-expert samples with noise scale σ and feed them into the memory (DDPG is off-policy learning), then train the critic networks with generated samples for N_{pre} epochs. The frozen actor can be achieved through setting the actor learning rate to be 0 in practice. After the 'preheating' step of N_{pre} epochs, we train the initialized DDPG in a general way.

As shown in Fig. 14, without pre-training the critic, there is always a severely decrease in performances

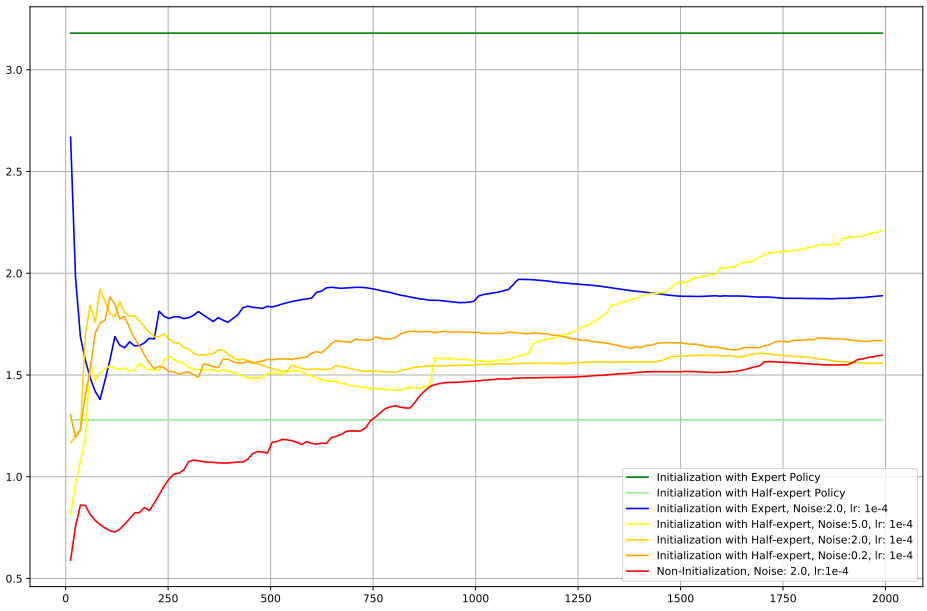

Figure 11: Experiments with different initialized policy with different scale of exploration noise for DDPG on *Reacher* task. Expert and half-expert are two initialization policies of different level of performance.

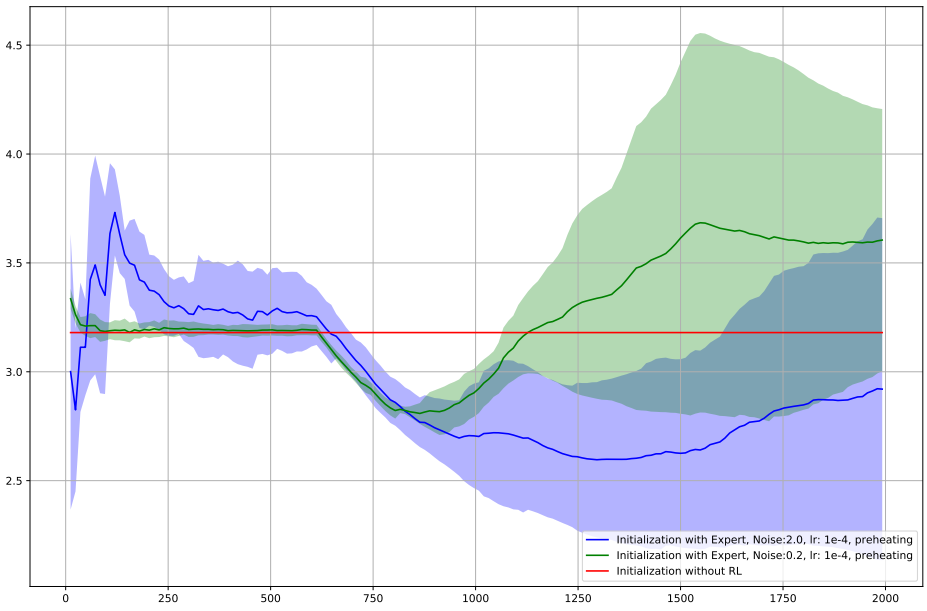

Figure 12: Experiments with different initialized policy for DDPG on *Reacher* task

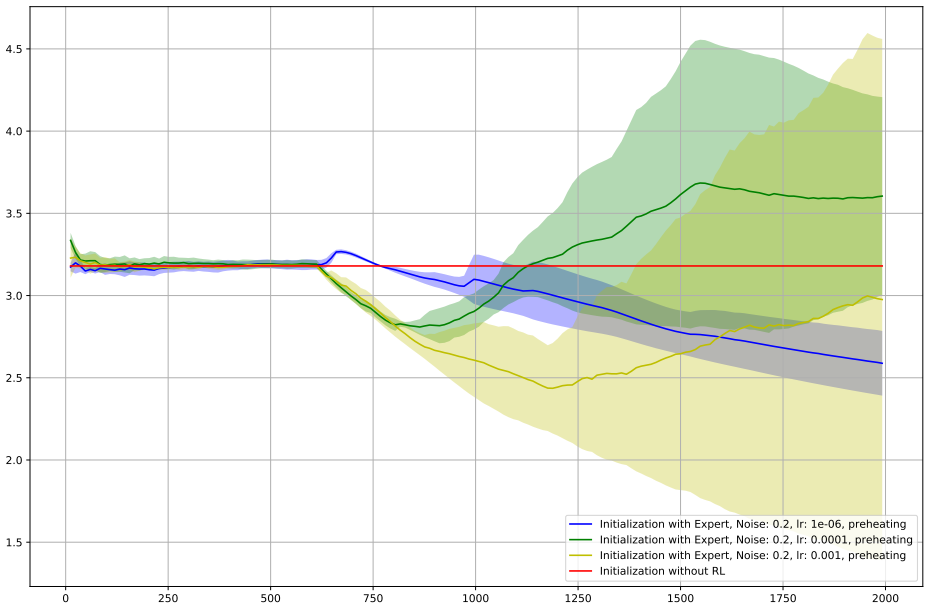

Figure 13: Experiments with different initialized policy for DDPG on *Reacher* task

of the initialized DDPG algorithm, no matter what the scale of exploration noise is. The decrease phenomenon for initialized DDPG without preheating can be explained as follows: the main problem of initialized policy for RL without preheating is that the virtue of initialized policy is ruined too quickly through changing of the weights of the initialized actor, and this is always the case as long as the learning rate is of a relatively considerable value. Because no matter how much the exploration noise is, the weights of the actor will be changed according to the learning rate of the actor, and neural network as an estimator has property of sensitivity on weights, which ends up with a dramatic changing in output policy of the actor. Therefore the actor will diverge fast from the initial near-expert policy, without a good critic. A good critic means it shows positive instructions for the actor to update weights in the correct direction. On the other hand, a good actor here in initialized DDPG will have great chances to decrease its performance with a random initialized critic. Therefore the 'pre-heating' process actually prevent the actor from updating to hurt the performances, or at least alleviate it. This is shown clearly in Fig. 14 that the lower bound of initialized DDPG with preheating is much higher than without the preheating process.

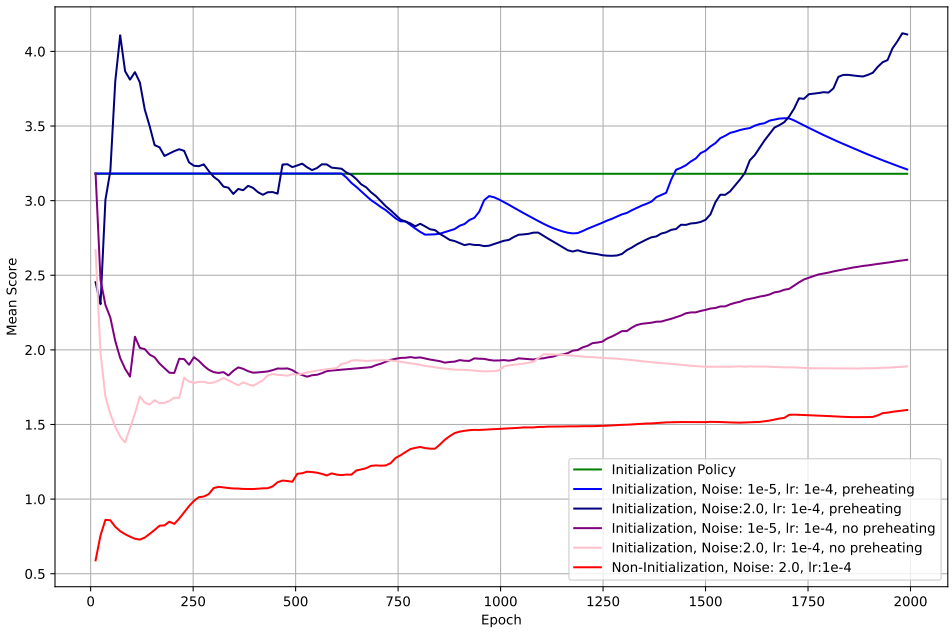

Figure 14: Comparison of initialized DDPG with/without pre-training the critic on *Reacher* task. The green line is the mean reward of initial policy trained with supervised learning.

Conclusions Some conclusions can be derived from above analysis and experiments about DDPG with initialization policy for *Reacher* task: (1). Different initial policy requires different noise scales to achieve better performances. For expert initialization policy, smaller exploration noise for DDPG shows better learning effects; for half-expert initialization policy, larger noise shows better learning effects.

(2). The preheating process is important for initialized policy on DDPG: with the pre-training of the critic before general learning process of DDPG, the learning performance is always better than without

it, It reduces the dramatic decrease at the initial training phase when the DDPG is initialized with an expert policy. The important intuition from this is that we should not use the critic for instructing the actor unless it's good enough for a good actor.

3.3.4 PPO with Policy Replacement

PPO is an on-policy RL algorithm, and its exploration is controlled by the variance σ of output action distribution from the policy neural network, which is learned during the training process. We can control the scale of exploration through adding a value to the variance, which is called noise base σ_0 here. So the actual action variance is $\sigma = \sigma_0 + \sigma_{pre}$ where σ_{pre} is trained output variance of the policy network.

The preheating process for PPO is conducted by pre-training the value evaluation network, similar to pre-training the actor in DDPG.

Fig. 15 shows comparisons of initialized PPO for *Reacher* task. Similar to conclusions in initialized DDPG algorithm, the preheating process is effective for reducing the decrease in initial training phase of PPO when initialized from an expert policy.

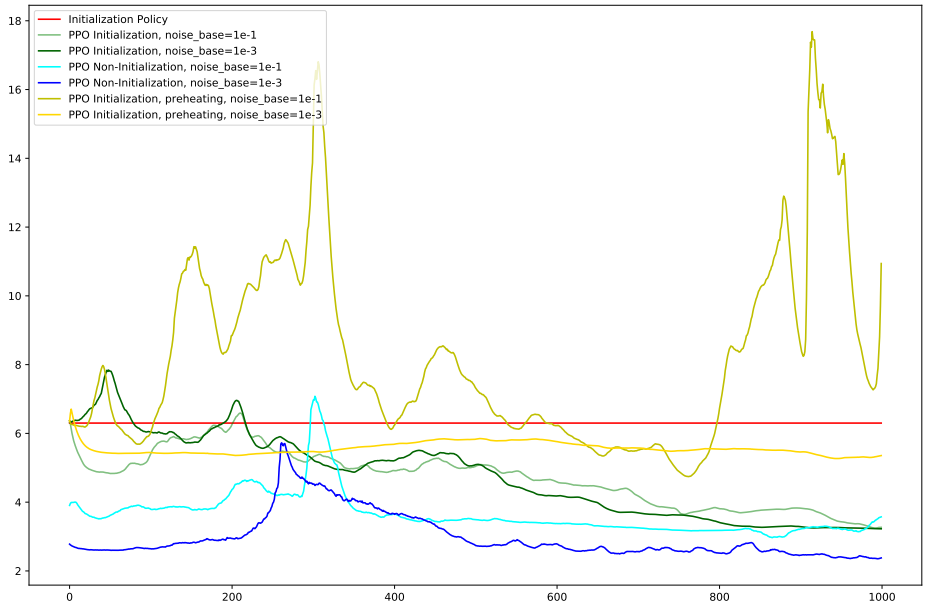

Figure 15: Comparisons of PPO with or without initialized policy and preheating process with different noise scale.

Fig. 16 shows different actor learning rates for initialized PPO. A proper range of actor learning rate is needed for effective learning.

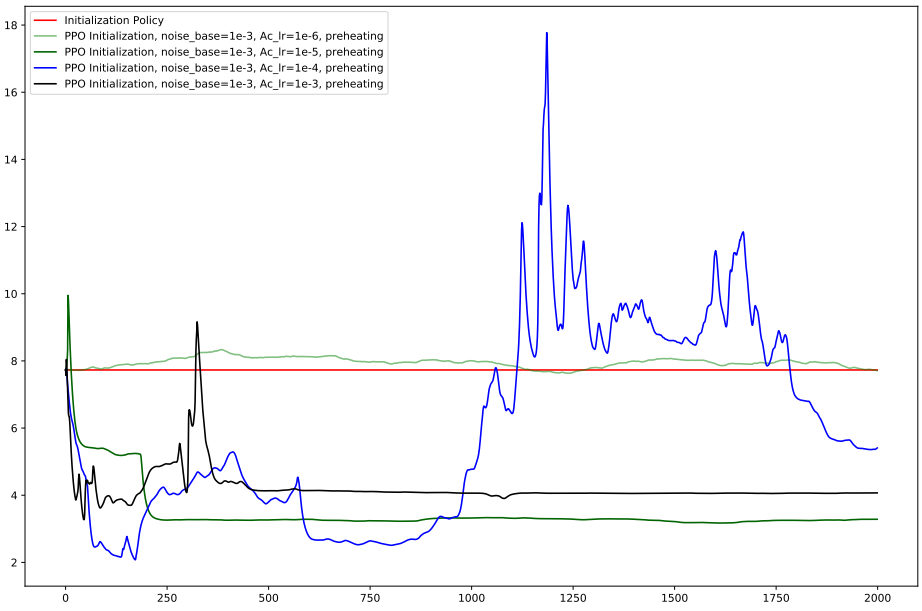

Figure 16: Comparison of different actor learning rates for initialized PPO with preheating process on *Reacher* task.

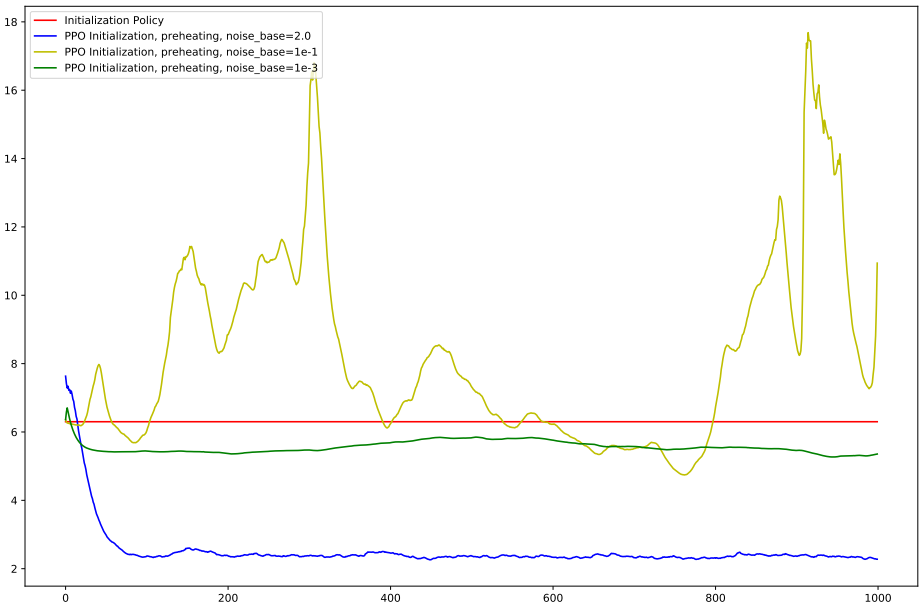

Figure 17: Comparison of different noise base for initialized PPO with preheating process on *Reacher* task.

Fig. 17 shows different noise base for initialized PPO. A proper range of noise is needed for effective exploration and learning.

3.3.5 Challenges of Policy Replacement

In practice, policy replacement as a method of behaviour cloning in leveraging demonstrations for RL does not work very well, even with tricks like pre-heating process etc. The main challenges of this method lie in two aspects:

1. **Compounding error problem**: if we train a policy with a demonstration dataset as an initialization for RL, those states met in explorations of RL but not seen in the training dataset will drift the policy further away from the initialization policy trained on demonstrations, until the policy is able to make good decisions for those unseen states in the training dataset. And in policy replacement, the policy may be drifted rapidly in the RL process and make it not able to perform well in those seen states in demonstrations. This process can be regarded as a generalization process from the initialization policy to a more general policy for the task.

2. **Difference in training objectives**: as we know, the training objective in imitation learning to train the initialization policy is of the form Eq.(19). However, when we initialize the RL with this pre-trained initialization policy, take DDPG as an example, the loss function of the actor network is to maximize the action value $Q(s, a)$ of its predicted action a with the input state s, and it is back-propagated from the critic network. The difference in training objectives in supervised imitation learning and RL algorithms will cause a discrepancy in the direction of optimization, which is another source of wasting of computation in training. Therefore, **it may have better performance if the training objective of supervised imitation learning could be implemented in the same form as the RL training does**.

3.4 Initialized Reinforcement Learning with Supervised Learning Policy – Residual Policy Learning

In addition to the policy replacement approach for initialization of RL, residual policy learning [19] is another approach to realize initialization. It s based on good but imperfect controllers for robot manipulation tasks, and to learn a residual policy on top of that initial controller. For robot manipulation in real world, the initial controller could be a pre-trained policy in simulation; and for robot manipulation in simulation, the initial controller could be from the pre-trained supervised learning with expert trajectories as in Section 3.2.

The action in residual policy learning is the sum of the initial policy π_{ini} and the residual policy π_{res}:

$$a = \pi_{ini}(s) + \pi_{res}(s) \tag{22}$$

In this way, the residual policy learning is able to preserve the initialized policy performance to the best advantage. Some tricks are applied to guarantee that:

(1). The weights of the output layer of the actor in RL are initialized to be 0;

(2). The same preheating process as in DDPG with initialization using policy replacement approach: train the critic while fixed the weights of the actor to alleviate the decrease in initial RL training stage caused by a bad critic;

(3). Apply amplification factor for noise scale during the preheating process: for pre-training the critic to have better generalization ability, apply noise multiplied by an amplification factor in the preheating process and reduce it to normal when training the actor.

Note that with above settings, the actor network outputs are actually all 0 during the preheating process, so the action values are fully attributed to the amplified noise and initial policy, which means $\pi_{res} = \epsilon$ and also:

$$a = \pi_{ini}(s) + \alpha * \epsilon, \alpha > 1 \tag{23}$$

where ϵ denotes the action noise and the α denotes the amplification factor. And the action values stored in the memory (with DDPG algorithm) are $\pi_{res} = \epsilon$ during preheating. This is not a problem as we only train the residual policy π_{res} instead of the composite policy $\pi_{ini} + \pi_{res}$, and action values stored in memory are just $\pi_{res}(s)$ instead of the real action value a. Correspondingly, the estimated Q-values learned by the critic are also only for the residual policy, which are $Q(s, \pi_{res}(s)))$.

However, there are still several freedoms to be set in residual policy learning, e.g. the scale of the output action from the residual policy can be chosen arbitrarily, which is actually a trade-off between the initialized policy and the learning policy, etc.

3.4.1 DDPG with Residual Policy Learning

We experimentally apply the DDPG algorithm for leveraging the demonstrations with residual policy learning. According to the residual learning, the actor policy in DDPG consists two parts: one is the pre-trained initialization policy, which will be fixed after initialization; and another one is the residual policy to be trained during the learning process. The initialization policy is pre-trained with the demonstration samples generated from inverse kinematics, which is the same as the policy replacement method. The pre-trained initialization policy only works for the actor part in DDPG. The process of applying residual policy learning in DDPG is as follows:

(1). Initialize all neural networks in DDPG with residual learning, including a general initialization of the critic, target critic, and an initialization with zero-value final layers for the residual policy and the target residual policy, and a policy replacement for the initial policy and the corresponding target, totally 6 networks. Fix the initialized policy and its target, and start the training process.

(2). Let the agent interact with the environment, and the action value is the sum of the action values from the initialization policy and the residual policy: $a = a_{ini} + a_{res}$; store samples in the form of $(s, a_{res}, s', r, done)$.

(3). Draw samples $(s, a_{res}, s', r, done)$ from the memory buffer, we have:

$$Q_{target}(s, a_{res}) = r + \gamma Q^{\mathcal{T}}(s, A^{\mathcal{T}}_{res}(s)) \tag{24}$$

where $Q^{\mathcal{T}}, A^{\mathcal{T}}_{res}$ denote that the values are calculated with the target networks, the target critic and the target residual actor policy respectively. The critic loss is $\text{MSE}(Q_{target}(s, a_{res}), Q(s, a_{res}))$. And the objective for the actor is to maximize the action-value function of state s and action a_{res} as follows:

$$Q(s, a_{res}) = Q(s, A_{res}(s)) \tag{25}$$

(4). Repeat above steps (2)(3) until the policy is converged.

Compared with general DDPG algorithm, the difference of applying residual policy learning is just to learn action-value function and the policy with respect to the residual policy actions instead of the overall action values for the agent.

3.4.2 Comparison of Residual Learning and Policy Replacement

We chose the goal position to be easy enough to reach for the agent, in order to shorten the training process and show the overall learning curves, like the left one in Fig. 18.

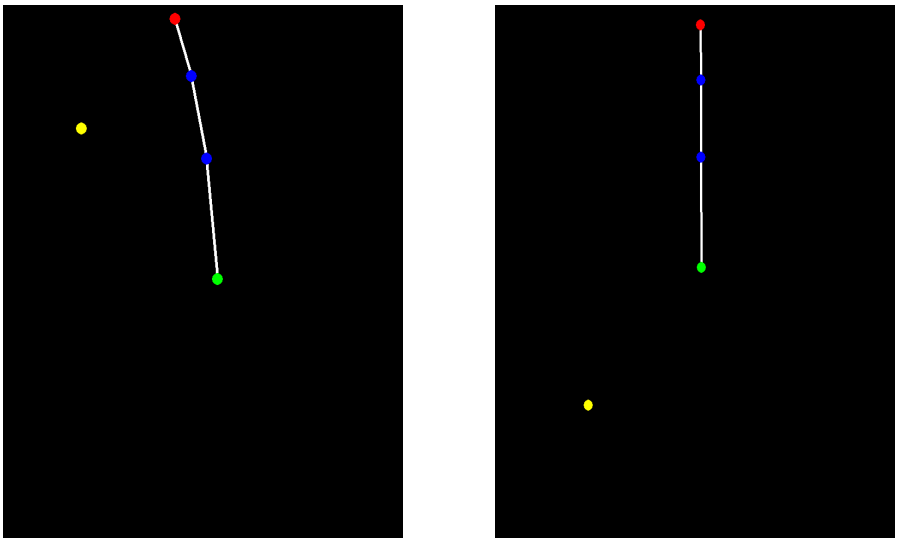

Figure 18: Different goal positions (the yellow dot) for *Reacher*. The left one is easier to reach than the right one with vertical initialization for *Reacher*.

With the same experiment settings: actor learning rate = 1e-4, critic learning rate = 1e-4, steps per episode = 12, same actor and critic network structures, preheating steps = 600 in initialized RL, noise scale = 2.0. The comparisons of non-initialized policy and initialized policy with residual Learning and policy replacement on DDPG for a easy-to-reach goal are shown in Fig. 19. For some easy-to-reach goals, the policy replacement approach seems to outperform the residual policy learning and non-initialization RL.

Comparison in Fig. 20 shows that, for some other goals, initialization with residual learning may not learn as well as policy replacement at the beginning of training, but will eventually learn better policy. This may benefit from the performance guarantee of the initial good policy for residual policy learning, while in policy replacement the learning process may search in a larger trajectories space (testified in observing the experiments).

With above analysis, we can see that the performances of different approaches to initialize the RL actually have large variance, which are significantly affected by parameters settings and task specifications like goal position, etc.

3.4.3 Modelling Analysis of Residual Policy Learning

Consider that the residual policy learning is actually using policy $\pi_{res}(s)$ to generate the residual action, whose inputs s are exactly the same as the initial policy $\pi_{ini}(s)$, the modelling of residual policy is actually may be more complicated than a general policy in RL, although the value range of outputs for residual policy may be smaller (a smaller searching space for delicate actions). In order to have a more accurate composite action, the residual policy not only needs to model the environment but also the initial policy. The modelling relationship can be interpreted as follows.

We denote the modelling of a general policy in RL as $\mathcal{M}_g$, the modellings of the initial policy and the residual policy are $\mathcal{M}_i, \mathcal{M}_r$, respectively. A modelling means a mechanism that the policy learned from interactions with its environment to represent the its environment and based on which the policy

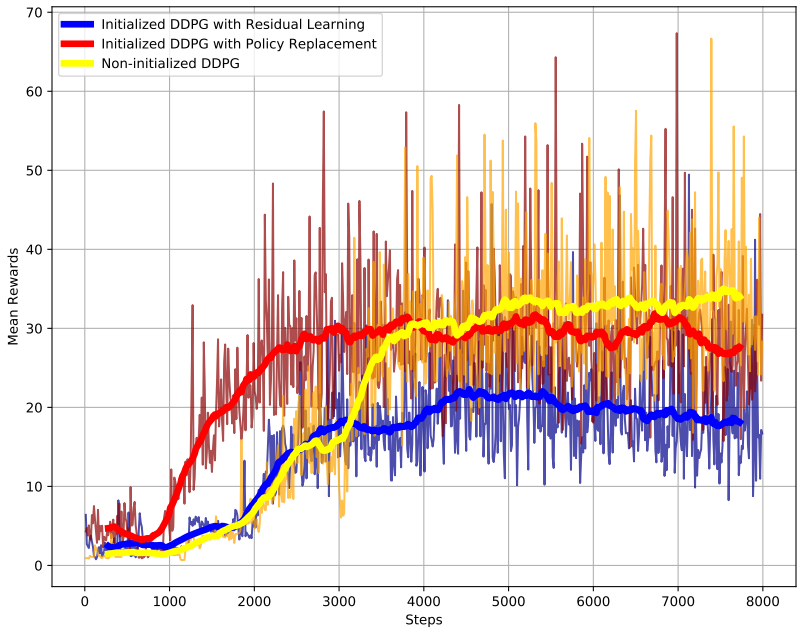

Figure 19: Comparison of non-initialized policy and initialized policy with residual learning and policy replacement for DDPG with an easy-to-reach goal position. The bold lines are moving average of episode rewards during learning process. The initialized policy has a preheating process of 600 steps.

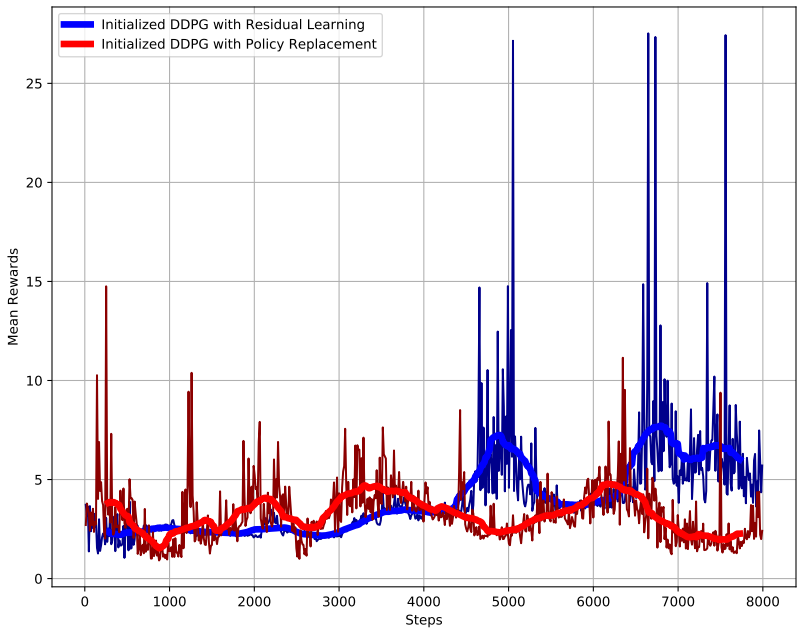

Figure 20: Comparison of residual learning and policy replacement for DDPG with another goal position (the one on the right in Fig. 18).

determines actions. The relationship of above modellings is:

$$\mathcal{M}_g = \mathcal{M}_i \oplus \mathcal{M}_r \tag{26}$$

which means the $\mathcal{M}_i, \mathcal{M}_r$ are actually dual modellings with respect to a $\mathcal{M}_g$. As we only train the residual policy in the learning process of residual RL, we have the modelling to construct in this training process the $\mathcal{M}_r$:

$$\mathcal{M}_r = \mathcal{M}_g \ominus \mathcal{M}_i \tag{27}$$

where the $\oplus, \ominus$ denotes a positive or negative modelling composite relationship. And the composite relationship does not determine the complexity of the composite modelling to be more or less than original constituent modellings, which means the modelling of $\mathcal{M}_r$ may not be less complicated than the general policy $\mathcal{M}_g$, and sometimes even more complicated than $\mathcal{M}_g$. This means the learning of a residual does not have to be easier than directly learning the general policy.

Discussions

In practice, take the example of using neural networks as approximators for the initialization policy (like an expert controller) and the residual policy, we can actually manually choose the complexity of the networks and range of the outputs of each of them, which can be induced by some human knowledge prior or preference. The knowledge prior can be understood as that if we know that our initialization policy is good enough, then we can restrict the output range of the residual policy as an additional action to be small; and the preference can be understood as that if we require the robotics trained in physical world to move within a safe range [20], we can also achieve it through restricting the output range of the residual policy. **The residual policy learning enables us to apply some extra tricks (restrictions, bias, preferences, etc) on the residual policy without affecting the initial policy, to improve the learning performance overall.**

However, it may not works sometimes with restrictions on output range of the residual policy, especially when we apply an activation function like tanh or sigmoid and the optimal output values may fall out of the range of the output values we restrict the residual policy to have. It will cause the neural saturation in the output layer and vanishing gradients, and therefore slow down the learning process greatly or even stop the improvement. And if we don't restrict the output of the residual policy, the large variance of its exploration will have chances to overwhelm the initial policy. And according to above analysis, the learning of residual policy does not have to be simpler than the over actor policy, therefore learning a residual aside from a good initialization policy may not show advantages over learning the actor policy directly.

3.5 Off-Policy Reinforcement Learning with Demonstrations

We show different initialization strategies for efficient RL as above, and leveraging demonstrations for deep RL is an alternative approach to learn efficiently. Both initialization approach and leveraging demonstrations approach are taking advantages of the expert trajectories, however, they treat them differently. Instead of pre-training a policy to initialize the RL policy, the approach of leveraging demonstrations directly feeds those expert trajectories into memory of off-policy RL (e.g. DDPG), to train the policy with both demonstrations and explorations. We show some experiments about the method of DDPG from demonstrations (DDPGfD) [21], without using the technique of prioritized experience replay [22].

We apply a more complicated environment for this experiment as shown in Fig. 21, with two penalty areas. The optimal trajectories for the agent is to go through the middle of two penalty areas. And we generate good demonstrations by manually setting an intermediate goal position at the middle of two penalty areas.

Experiment results of the DDPG from demonstrations approach for efficient RL with different ratios of demonstration data over all training data are shown in Fig. 22. The demonstration data contains 50 episodes of trajectories in the above *Reacher* environment with obstacles, with 21 steps for each episode. Aside from a general memory for storing exploration data in off-policy DDPG, there is an additional demonstration buffer for storing the data from expert-demonstrations at the beginning of the training. The ratio of demonstration data is used for balancing the data sampled from demonstration buffer or from the exploration memory for each training step. From Fig. 22, the learning performance of DDPG algorithm trained without demonstration data (ratio of 0.0) or totally from the demonstration data (ratio of 1.0) are both worse than a proper combination of demonstration and exploration data, e.g. ratio of 0.8, which testifies the positive effects of leveraging demonstrations in DDPG.

In the original DDPGfD [21], Mel Vecerik et al applied the prioritized experience replay as a natural balance of the two sources of training data. Actually in practice, in order to get a better performance, more flexible approaches of leveraging the demonstrations can be applied, like a decaying ratio of demonstration data, or only training the model with demonstration data when the explored trajectories is not better than them, etc.

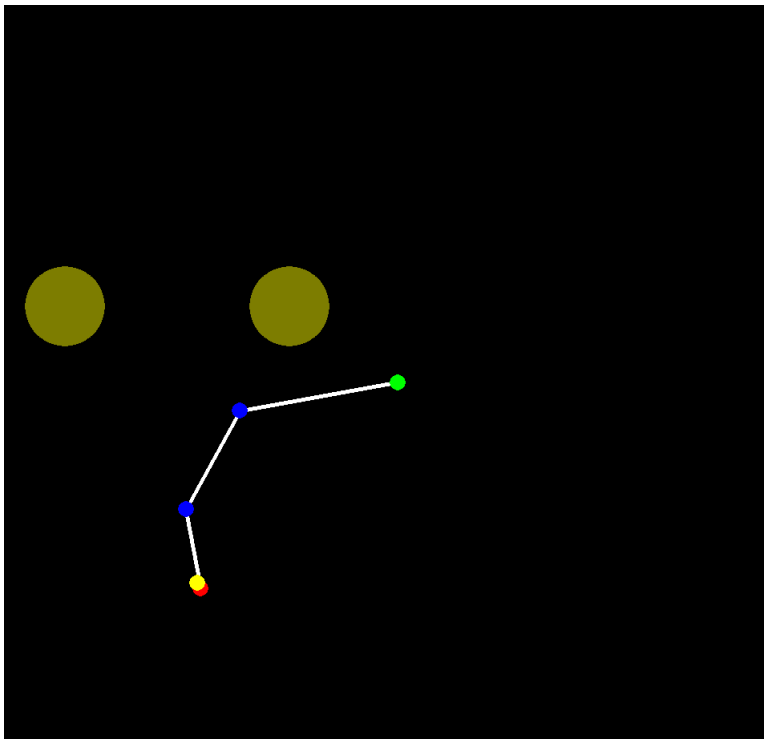

Figure 21: *Reacher* environment with two penalty areas.

3.6 Benchmark Methods of Reinforcement Learning with Demonstrations

3.6.1 Variances in Experiments

We first need to show the property of various and unstable performances in RL's training process, especially in algorithms like DDPG, etc. We take the residual policy learning with DDPG as example, displays the large variance in different trials of training. The unstable property of the learning process

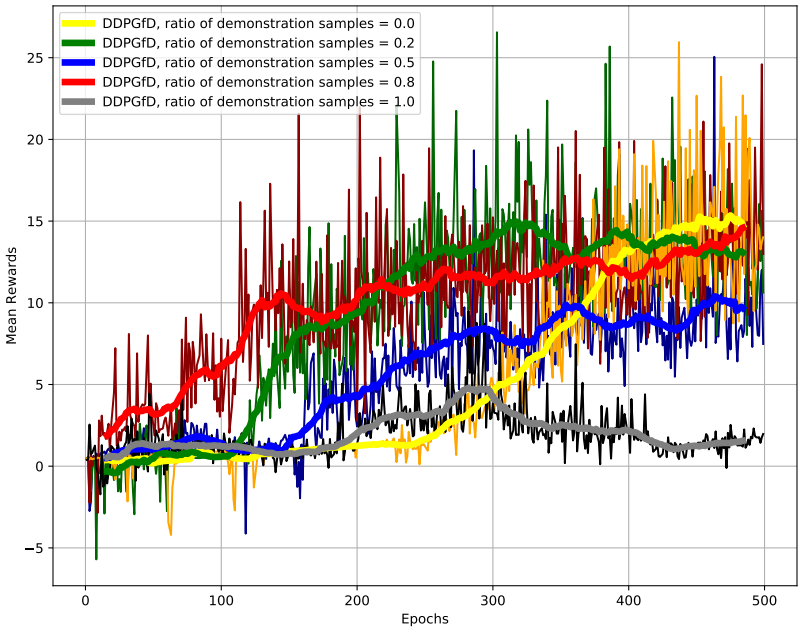

Figure 22: Results of DDPG from demonstrations with different ratios of demonstrations over all training samples. With demonstration sample ratio equivalent to 0.0, the DDPG policy actually learns from scratch without any demonstrations; and with the ratio equivalent to 1.0, it learns totally from the demonstrations without any samples from exploration. And in both of the above cases, the learning performance is as good as a proper ratio of demonstration between 0.0 and 1.0, e.g. 0.8.

in RL contains several parts, including a non-converged optimal policy, large differences in performance of different trials, etc.

Experiments with three trials using residual policy learning for DDPG are shown in Fig. 23. We can see from the first two trials that some near optimal policies (as the black crosses in the figure) are learned at a relatively early learning stage with residual learning, but it does not converge to the policy and eventually learns a worse policy than that. And the third trial shows that the agent does not learn anything useful over the whole training epochs sometimes. Moreover, it shows that there is large performance variance in the learning process even with exactly the same methods and experimental settings, which is a typical property in experiments of RL. Therefore it will be common to see some large variances in experiment results shown in the following sections.

3.6.2 Comparisons of Methods for Reinforcement Learning with Demonstrations

In this section, we display comparisons of experiment results with some of the above described efficient RL methods using demonstrations in the *Reacher* environment, including policy replacement, residual policy learning and directly feeding demonstrations into the buffer, etc. All methods are based on the same DDPG algorithm with same settings in hyper-parameters and neural network architectures. We use general experiment settings which are used throughout this paper.

General Experiment Settings

* Environment: the leanring environment is a *Reacher* environment with 2 penalty areas of radius 50 as shown in Fig. 21 and joint length of $[200, 140, 100]$ with the screen size 1000, as described above.

* Neural networks: the actor network includes 5 fully connected layers with ReLu as hidden layer activation function and Tanh as output activation, and the size of each layer is 100 for the first 4 layer and 400 for the last layer. The critic network are almost the same just without activation for the last layer and so with Tanh for the last but one layers. The learning rate is 10^{-4} for the actor and 10^{-3} for the critic. Batchsize is 640, and exploration uses normal noise of scale 2.0 for experiments in Fig. 24.

* Dataset of demonstrations: we use two datasets of demonstrations, a small one and a large one. The small dataset contains 50 episodes of expert trajectories and the large one contains 1000 episodes, with 21 steps for each episodes. The demonstration trajectories are generated with a intermediate goal at between the two penalty areas via the inverse kinematics, and with injected normal noise. The demonstration data is used in several methods, including learning from demonstrations, pre-training the policy network as initialization and pre-training the policy network in residual learning.

* Training: both the initialization and residual learning methods are set to have a pre-heating process for training the critic, and the length of the pre-heating is 600 steps. The number of overall training steps is set to be 10^4 for all methods.

Experiments with Dense Rewards

Comparison of different methods using demonstrations and dense reward function as in Eq.(15) with DDPG algorithm for *Reacher* environment are shown in Fig. 24, including policy replacement, residual policy learning and directly feeding demonstrations into the buffer ((demonstration ratio: 0.5)). We set the neural network architectures and other experiment parameters including the noise scale (=2.0) and activation function of output layer ($\alpha*$Tanh, $\alpha = 30$) to be exactly the same for all the methods, including vanilla DDPG. For the method of residual policy learning, the residual policy network is set to be the same with actor policy networks in other methods. Experiments in Fig. 24 shows that, directly feeding demonstrations into the memory buffer has more robust improvement in the learning process with general experiment settings, and the feeding ratio of demonstrations is 0.5 as default.

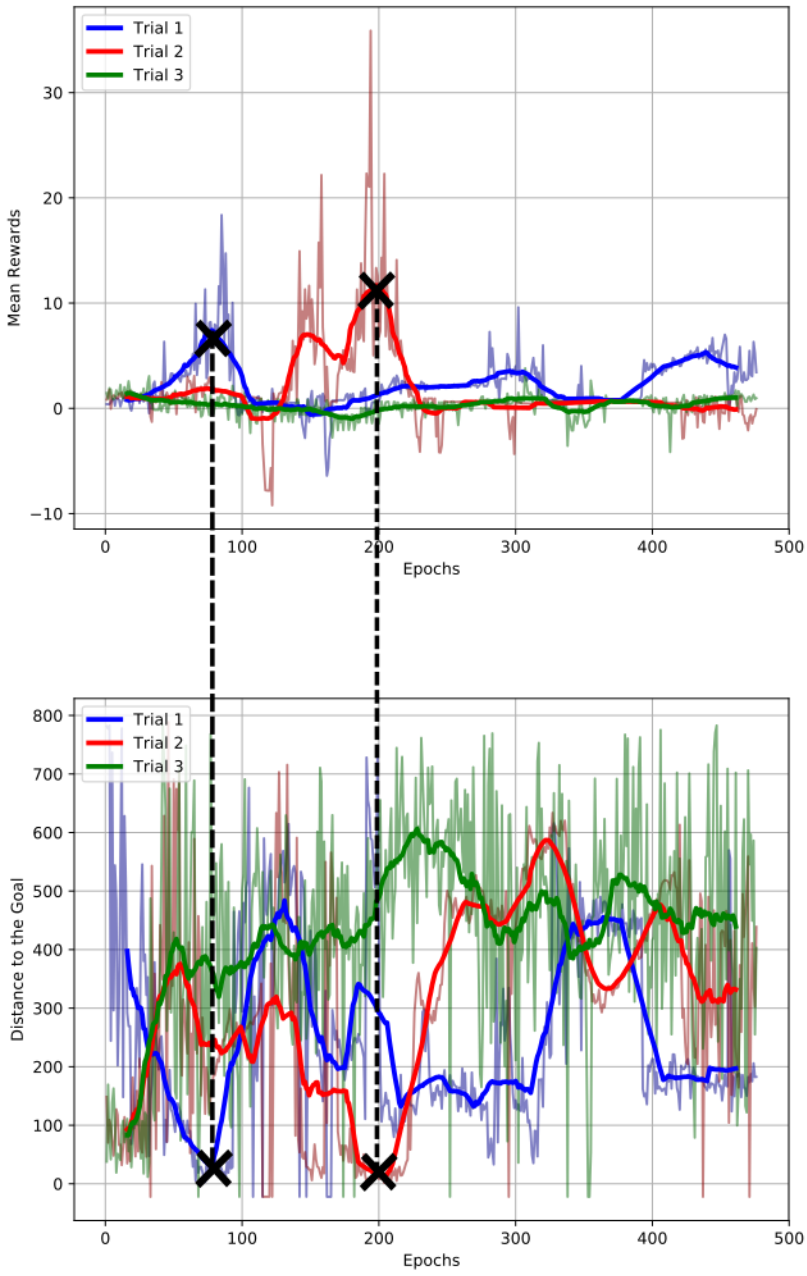

Figure 23: Variances of different trials in residual policy learning with DDPG in *Reacher* environment. There are three trials in above experiments. The figure on the top shows the mean reward over the training process; and the bottom one shows the distance of the end of reacher from the target goal during training epochs.

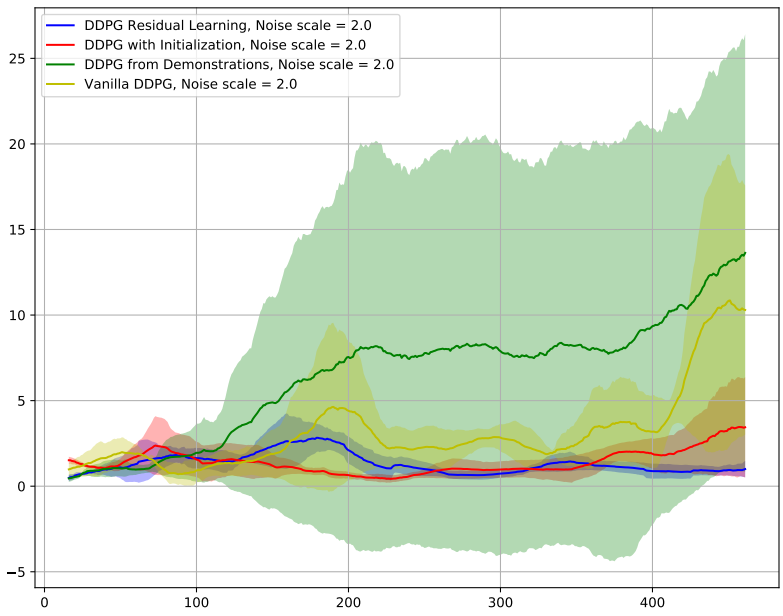

Figure 24: Comparison of different methods with demonstrations for efficient reinforcement learning (DDPG), including policy replacement, residual policy learning, directly feeding demonstrations (demonstration ratio: 0.5) into the buffer and vanilla DDPG. The overall length of training step is 10000, and there is a 600-step pre-heating process for policy replacement and residual policy learning.

We also compare above methods with a same neural network architecture but different activation functions for the output layer and different hyperparameter settings, to fine-tune (showing the best performance) each method on the same *Reacher* task, as shown in Fig. 25. Different methods use different output activation functions, range of action values and noise scales, etc. The activation function of output layer of vanilla DDPG and policy replacement is the LeakyRelu function, while for the method of directly feeding demonstrations into memory buffer (DDPGfD) the function of $\alpha*$Tanh works more robustly and the α (=30) is a factor of value range. And for the residual learning, Tanh activation function in the output layer is used for the initialized policy but is easily saturated for the resisual polciy in experiments, therefore we apply an activation function of LeakyRelu6 (LeakyRelu with a clip on positive axis of range [0,6]) with a scaling parameter $\alpha(= 0.2)$ for the residual policy. Therefore the action value range of the residual policy is much smaller than a general action policy, which indicates a delicate learning upon a good initialization policy. Actually, the better the initialization policy, the smaller the range of both action value and noise scale is needed for residual learning, which is explained in the section of residual policy learning. Moreover, different methods use different noise scale for their own benefits as shown in the figures. Then uninitialized policy tends to use a larger noise scale for better exploration, while the initialized policy like the residual learning uses a smaller noise scale as it only learns a residual upon a pre-trained policy.

Two figures are shown in Fig. 25, the mean episode reward versus training epochs and the final distance of the end of reacher to the goal position versus training epochs. As different methods requires different experiment settings including the outputs of the learned policy (e.g. the action values in the residual learning are affected by two policies, and they may have different ranges of action value compared with other methods, which will cause the pace of the reacher's motion and average episode rewards to be different), it is not a fair comparison with only the figure of mean episode reward. Therefore we show an additional figure of final distance to the goal position. And there are some other cases may cause the discrepancies in the two figures in Fig. 25: sometimes the reacher may go through the goal position but not stay at it until the end of an episode and still get a relatively large reward; and sometimes the reacher reaches the final goal position but through the penalty areas with a relatively small episode reward.

Experiment with Sparse Reward

Experiments above uses dense reward function as in Eq.(15). In this section, we show comparisons of different methods with the sparse reward function defined as follows (mentioned in former sections):

$$R = \begin{cases} R_T & \sqrt{(x_3 - x_T)^2 + (y_3 - y_T)^2} < r_T \\ -5 & \sqrt{(x_3 - x_P)^2 + (y_3 - y_P)^2} < r_P \\ -1 & \text{Otherwise} \end{cases} \tag{28}$$

where $R_T = 20, r_T = 30$ in experiments, and r_P is the radius of penalty areas equivalent to 50. Experiment results are shown in Fig. 26 with fine-tuning for each methods, including policy replacement, residual policy learning, directly feeding demonstrations into the buffer ((demonstration ratio: 0.5)) and vanilla DDPG. Basic settings of it is the same as the experiments shown in Fig. 25 in the above section with dense reward function. The residual policy learning with fine-tuning still performs the best as shown in the distance figure. And both methods of directly feeding demonstrations into memory and policy replacement show improvement in learning performance, the former one is better in learning effective trajectories.

Note that there are two unusual points in the Fig. 26: (1). the large reward with the approach of feeding demonstrations in the above figure is because of its 30$*$ Tanh (a relatively large action value makes the agent reach the goal faster) activation function for output layer. (2). as the noise scale is different in different methods, the final distance to the goal position is supposed to be larger on average in training with methods using relatively large noise scale like policy replacement and demonstrations

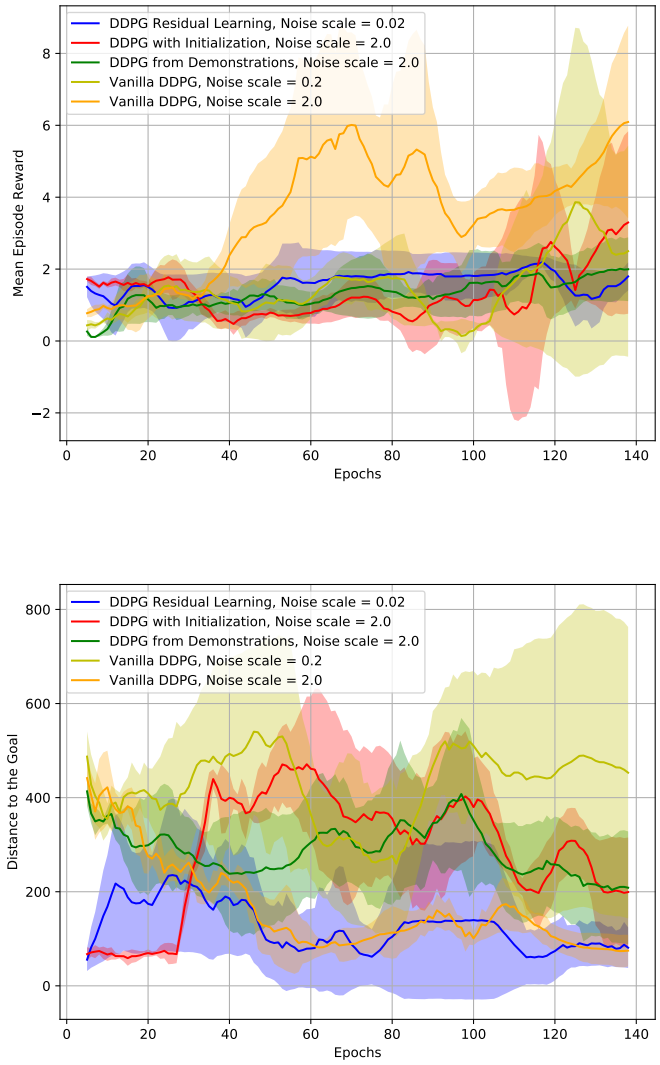

Figure 25: Early stage of training performances of different methods for efficient reinforcement learning (DDPG) with **dense reward** function, using different parameter settings (e.g. output layers of neural networks, noise scales, etc), including policy replacement, residual policy learning, directly feeding demonstrations into the buffer ((demonstration ratio: 0.5)) and vanilla DDPG. Each method is tested with three trials on the same environment. The overall length of training step is 3000, and there is a 600-step (28 epochs) pre-heating process for policy replacement and residual policy learning. The above figure is the mean episode reward versus the training epoch, and the one below is the final distance to goal position for each episode versus the training epoch. We can see from the second figure that the residual policy learning actually performs well with fine-tuning, as it learns upon a good initialization policy.

feeding approaches, compared with residual policy learning, even their policy performances may be similar in test without noise (e.g. they can all reach the goal quickly).

As for vanilla DDPG, it generally cannot learn anything useful in a training scope of 3000 steps. Therefore, above three methods for leveraging demonstrations actually have more significant improvement in learning performance with sparse reward than with dense reward.

Experiment with Small Set of Demonstrations

We use the large dataset with 1000 demonstrations in above experiments. And in this section we wonder if there is any bad effects if we use a smaller dataset, containing only 50 demonstrations, which can be derived easily even in real world scenario with human experts. All experiment settings are remained the same as above sections. And the reduction in demonstration dataset actually only affects the method with policy replacement as initialization and the residual policy learning, because the initialization policy will be different for these two approaches. As for the approach of feeding demonstrations into memory, it should not be affected as long as the number of demonstrations fed into the memory is sufficient for the sampling process of training, because those demonstrations are all generated by expert policy with random noise and the number of it does not matter. We also use sparse reward function in this experiment.

Experiment results are shown in Fig. 27. Compared with applying a large demonstration dataset in Fig. 26, no prominent bad effects are displayed through the experiments for those initialized policy methods.

Conclusions

Some conclusions can be derived with above experiments using described three approaches for leveraging demonstrations in RL process, as listed in the following:

1. The approach of learning with demonstrations fed directly into the memory buffer shows robust improvement in performance with general experiment settings;

2. The approach of residual policy learning works the best with fine-tuning and good initialization policy, benefiting from the restricted exploration range (sometimes manually set according to the performance of initialization policy) of the residual policy;

3. Vanilla DDPG could work well with dense reward function, but much worse with sparse reward function, even learn nothing useful in the early training stage;

4. The effect of improvement in learning performance using above methods for leveraging demonstrations, including policy replacement, residual policy learning and directly feeding demonstrations into memory, is more prominent with sparse reward function than with dense reward.

5. There is no prominent bad effects when the demonstration dataset for training the initialization policy using supervised imitation learning is small, for the case of sparse reward function. And the case of dense reward function is supposed to be similar. But of course, if the demonstration dataset is too large (if covering most cases of the action space, it could be directly used as the policy without RL) or too small (e.g. 1 or 2 demonstrations), it could make some differences.

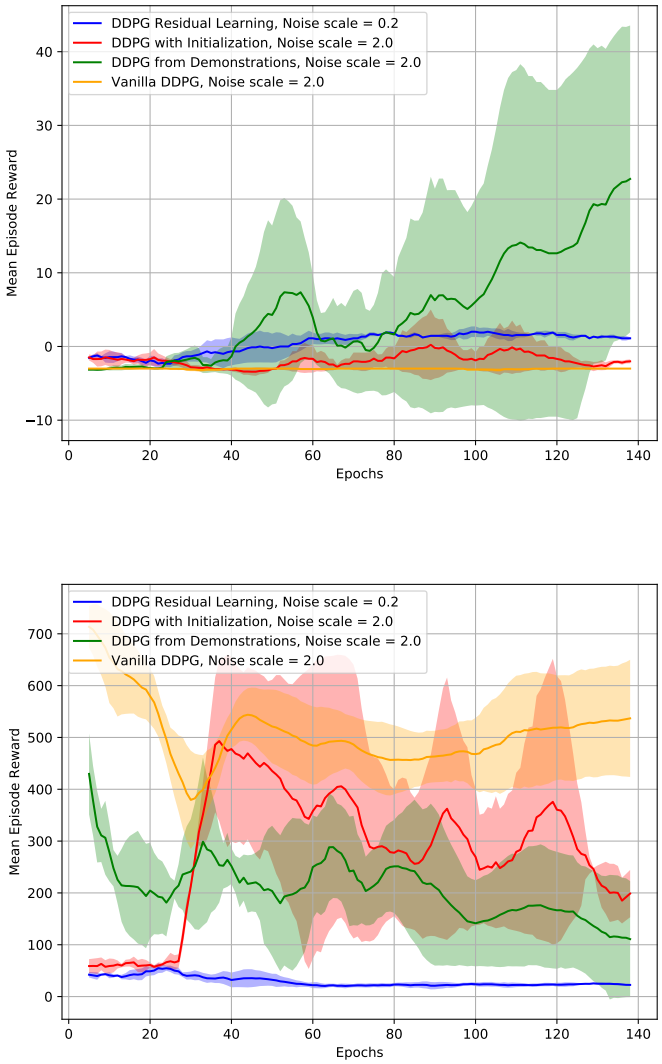

Figure 26: Early stage of training performances of different methods for efficient reinforcement learning (DDPG) with **sparse reward** function, using different parameter settings (e.g. output layers of neural networks, noise scales, etc), including policy replacement, residual policy learning, directly feeding demonstrations into the buffer ((demonstration ratio: 0.5)) and vanilla DDPG. Each method is tested with three trials on the same environment. The overall length of training step is 3000, and there is a 600-step (28 epochs) pre-heating process for policy replacement and residual policy learning. The above figure is the mean episode reward versus the training epoch, and the one below is the final distance to goal position for each episode versus the training epoch.

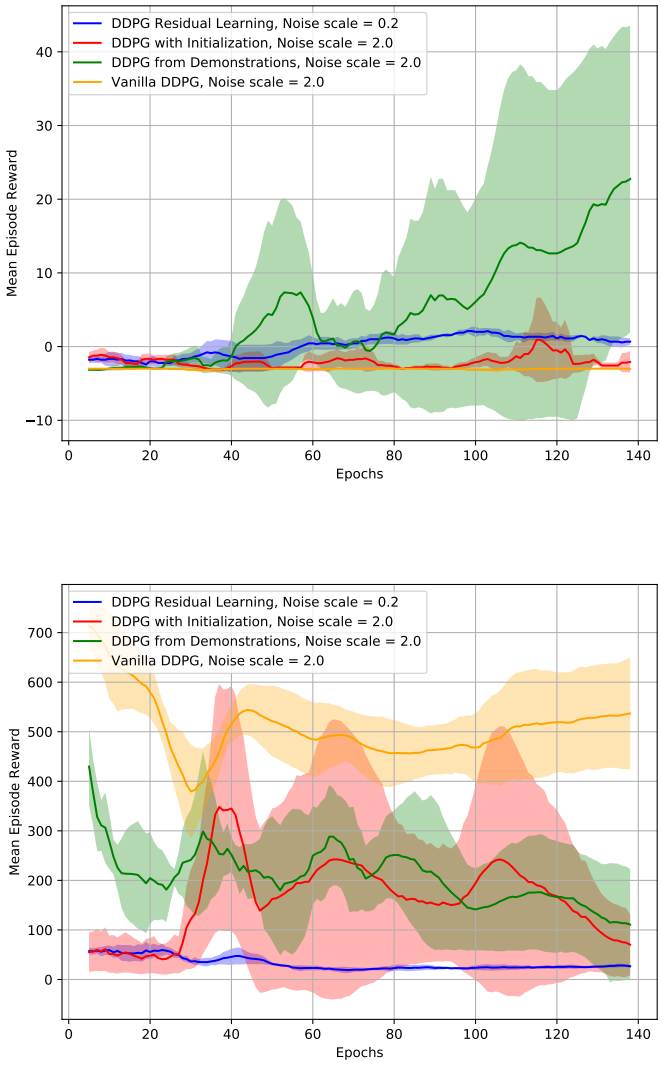

Figure 27: Early stage of training performances of different methods for efficient reinforcement learning (DDPG) with sparse reward function and **small dataset of demonstrations (50)**, using different parameter settings (e.g. output layers of neural networks, noise scales, etc), including policy replacement, residual policy learning, directly feeding demonstrations into the buffer (demonstration ratio: 0.5) and vanilla DDPG. Each method is tested with three trials on the same environment. The overall length of training step is 3000, and there is a 600-step (28 epochs) pre-heating process for policy replacement and residual policy learning. The above figure is the mean episode reward versus the training epoch, and the one below is the final distance to goal position for each episode versus the training epoch. We can see from the second figure that the residual policy learning actually performs well with fine-tuning, as it learns upon a good initialization policy.

4 Efficient Reinforcement Learning without Demonstrations

4.1 Meta-learning as Initialization for Reinforcement Learning

Meta-learning is also called learning to learn, lifelong learning, transfer learning, multi-task learning, etc. The goal of meta-learning is to learn transferable knowledge which can be generalized in multi-tasks, so as to accelerate the learning process in some computationally expensive tasks especially like reinforcement learning. Present meta-learning algorithms in reinforcement learning domain include model-agnostic meta-learning (MAML) [23], Reptile [24], E-MAML [25] and E-RL^2 [25], probabilistic MAML [26], Multi-Modal Model-Agnostic Meta-Learner (MuMoMAML) [27], Meta-Imitation Learning with MAML [28][29], and LSTM based methods [30] like RL^2 [31], Learn to RL [32], Learning to Learn by Gradient Descent by Gradient Descent [33], and meta-leaner with attention strategy like Simple Neural Attentive Learner (SNAIL) [34], etc. Some works point out potential problems of MAML like negative adaptation problem [35] and unstable training problems of MAML [36]. Modifications of MAML like MAML++ [36] for better performance and more stable training process are on the way.

Meta-learning generally works as a supervised learning problem, but it could be applied to RL problem, which is meta-RL. And meta-RL can be used in tasks like robotic control.

Task settings: as meta-learning is to learn across tasks, it is usually computationally expensive, especially for reinforcement learning tasks. In order to simplify the tasks, we reduce the number of joints in *Reacher* from 3 to 2. And the difference of tasks in a specific task domain is set to be the different positions of the target point.

4.1.1 First and Second Order Modal-Agnostic Meta-Learning Algorithms

MAML

MAML [23] is meta-learning method to unroll the computation graph with second derivatives for updating the meta policy across tasks. And it has an approximate version called first-order MAML (FOMAML) by just ignoring the second-order term. The updated policies of MAML contains an inner policy, which is a task-specific policy, and an outer policy, which is the meta policy. The goal of MAML is to learn a better meta policy for accelerating the learning process of every task-specific policies.

MAML can be regarded as updating the meta-policy according to the test error with updated parameters (according to the training error on old samples) on new samples. The update rule of the inner policy is a general one (w.r.t. the training error), for task $\mathcal{T}_i$ sampled from the task domain:

$$\theta_i' = \theta - \alpha \nabla_\theta L_{\mathcal{T}_i}(f_\theta) \tag{29}$$

and for the outer policy update (w.r.t. the test error on the new samples with updated parameters in inner update):

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} L_{\mathcal{T}_i}(f_{\theta_i'}) \tag{30}$$

where the α, β are stepsize variables, the θ, θ_i are parameters of the parameterization function f for the meta policy and policy of task $\mathcal{T}_i$ respectively (e.g. f_θ is policy π_θ parameterized by θ). The loss function L for task $\mathcal{T}_i$ with parameterized policy f_θ is generally:

$$L_{\mathcal{T}_i}(f_\theta) = \int R_{\mathcal{T}_i}(t) f_\theta(t) dt \tag{31}$$

and in RL algorithms like PPO:

$$L_{\mathcal{T}_i}(\pi_\theta) = \mathbb{E}[\log \pi_\theta(a|s) \cdot A_{\mathcal{T}_i}(s)] \tag{32}$$

where $A(s)$ is advantage function of state s.

FOMAML

FOMAML is a simplified version of MAML with only the first order derivatives of MAML. Therefore it is computationally more efficient than MAML. The update rules for FOMAML are:

$$\theta_i' = \theta - \alpha \nabla_\theta L_{\mathcal{T}_i}(f_\theta) \tag{33}$$

$$\theta \leftarrow \theta - \beta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \nabla_{\theta_i'} L_{\mathcal{T}_i}(f_{\theta_i'}) \tag{34}$$

for inner policy and outer policy update respectively.

Reptile

Reptile [24] is another meta-learning algorithm with only first-order derivatives similar to FOMAML for meta-learning with only first-order gradients. The update rule for Reptile is as follows:

For task $\mathcal{T}_i$,

$$\theta_i' = \theta - \alpha \nabla_\theta L_{\mathcal{T}_i}(f_\theta) \tag{35}$$

$$\theta \leftarrow \theta + \epsilon(\theta_i' - \theta) \tag{36}$$

Note that the first step of above update in Reptile is usually a k-step update and here just show a one-step update.

MuMoMAML

MuMoMAML [27] applies an extra model-based meta-learner on top of the gradient-based meta-learner in general MAML algorithms, which consists of an additional update of the modulation parameters ω in the outer policy update step :

$$\omega \leftarrow \omega - \beta \nabla_\omega \sum_{\mathcal{T}_i \sim p(\mathcal{T})} L_{\mathcal{T}_i}(f_{\theta_i'}, \tau) \tag{37}$$

where $\tau = g(\{x, y\}; \omega)$ and it is used for modulating the prior parameters of the gradient-based meta-learner with respect to the task specification data $\{x, y\}$, using the following equations:

$$\phi_i = \theta_i \tau_i \tag{38}$$

and in MuMoMAML the gradient-based meta-learner uses the modulated prior parameters ϕ_i instead of the original parameters θ_i. The intuition of the MuMoMAML is to let the gradient-based meta-learner in MAML be aware of the task specification with the help of the model-based meta-learner.

E-MAML

The E-MAML [25] algorithm changes the meta-policy loss function of MAML to be:

$$L_{\mathcal{T}_i}(\pi_{\theta'}) = \mathbb{E}[\log \pi_{\theta'}(a|s) \cdot A_{\mathcal{T}_i}(s) + \lambda \cdot A_{\mathcal{T}_i}(s) \cdot \sum_{s \sim \pi_\theta} \log \pi_\theta(a|s)] \tag{39}$$

in which it applies an interpretation of a stochastic gradient update of inner policy instead of the deterministic one in conventional MAML. And the λ is a trade-off parameter for smoothly tuning between the stochastic treatment and non-stochastic treatment of the inner loop policy update.

Directly optimizing original MAML loss function like Eq.(32) will not account for the impact of the original sampling distributions π_θ on the future rewards $R(t)$ on trajectories with updated (in inner loop) policy $\pi_{\theta'}$. The effect of the additional second term in E-MAML is to make the policy more exploratory.

Meta-Imitation Learning with MAML

Meta-Imitation Learning with MAML [28] is a another version of MAML algorithm for imitation learning with expert demonstrations. The difference of Meta-Imitation Learning with MAML from MAML is just the learning samples are from expert demonstrations instead of samples with current policy. The test error (outer policy) on expert demonstrations is the training error of the meta-learning.

Negative Adaptation in MAML

Tristan et al. [35] points out the adaptation phase (the inner policy update) in MAML update as in Eq.(30)has no guarantee to show improvement on an individual task, which could significantly decrease the performance on continuous control problems. The reason of it is that the meta-update in MAML encourages the adapted policy $\pi_{\theta'_\mathcal{T}}$ to have larger expected return while not considering the initial policy $\pi_{\theta_\mathcal{T}}$ before the adaptation phase. Therefore **the adapted policy $\pi_{\theta'_\mathcal{T}}$ may perform worse than the initial policy $\pi_{\theta_\mathcal{T}}$, which is called the negative adaptation**. And the authors propose a constraint on the meta-policy update:

$$\min_\theta \mathbb{E}_{\mathcal{T}\sim P(\mathcal{T})}[L(g(\mathcal{D}_\mathcal{T};\theta);\mathcal{D}'_\mathcal{T})] \quad \text{s.t.} P(P(\Gamma_\mathcal{T}(\theta) \leq 0) \leq 1-\beta) \leq 1-\delta \tag{40}$$

where $\mathcal{D}_\mathcal{T}, \mathcal{D}`_\mathcal{T}$ are data trajectories sampled from before and after the adaptation phase for task $\mathcal{T}$, g is the gradient function, and β, δ are improvement factors for the level of a specific task and for overall tasks respectively. $\Gamma_\mathcal{T}(\theta)$ is a function measuring the update of policy:

$$\Gamma_\mathcal{T}(\theta) = G_0(\pi_{\theta_\mathcal{T}}) - G_0(\pi_{\theta'_\mathcal{T}}) \tag{41}$$

and G_0 is a value function for measuring the expected return.

4.1.2 Understandings about Meta-Learning

We first need to clarify two types of policies that may be confusing:

1. an across-task policy, which is a single policy for all tasks without further training;

2. a meta-policy as initialization policy for the training process (sometimes few-shot learning) of each specific task.

The definition of these two different policies or say different formalization of tasks can be understood with the example of *Reacher* task with multi-goal positions. The former one can be considered as a powerful policy as a solution to *Reacher* task with whatever valid goal positions, and it can give effective actions for different goals and input states without further training for different goals. And the latter one can be considered a meta-learned policy as an initialization of further training for each specific goal position, therefore the policy will end up to be able to predict effective actions with input states for the specific goal position. It is easy to see that the first policy is a more generalized one and therefore needs more pareameters or larger neural networks as sufficient approximators, while the second policy is only a single-task level policy but with a meta-initialization, and therefore needs a much smaller neural network as a function approximator.

For meta-learning as an initialization like MAML, we only consider the second policy. However, if we need a more generalized policy like the first one, we could train a policy across task using techniques like hindsight experience replay and update the policy with only first order gradients

and averaging the gradients across tasks. But for MAML, we use a second order gradients update, **the intuition of MAML is just the difference of these two different kind of policy**. And we use the way the MAML update the parameters to achieve our goal of updating a meta-initialization policy properly.

Optimization with neural network approximators is just a descent process on reward-parameters map with descent direction determined by the gradients of the loss function. So we can use the reward-parameter map for illustrations of the difference of above mentioned two-kind of policies.

We consider a simplified case with only two tasks (e.g. *Reacher* with two goal positions), a discrete task domain unlike the continuous task domain of random goal positions of **Reacher**. The reward-parameter map is also simplified to have only two parameter dimensions for visualization. And the parameters are actually the parameters of the policy (usually large dimensions), i.e. the parameters of neural networks if using the neural networks as approximator of the policy. The reward is actually the policy value function with different parameter values.

Reward-Parameter Map of the General Policy Across Tasks (First)

As the policy uses a large neural network as its policy approximator, we can suppose that the parameter space is sufficiently large to always contain a point (a set of parameter values) that is optimal for both of the tasks. It can be understood with the Fig. 28, where the policy for single task is supposed to have one parameter and the general policy for two tasks contains two parameters. It indicates that a larger parameter space with a universal approximator like neural network will always have a set of parameter values for the optimal policy across tasks. Just the reward map could be much more complicated than a single-task policy.

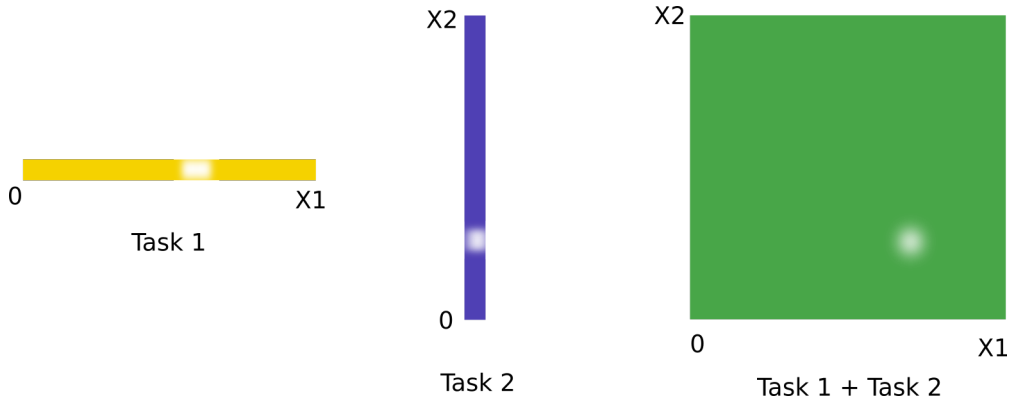

Figure 28: Figure of reward-parameter map for general policy across tasks. X_1, X_2 are parameters for policy in Task 1 and Task 2 respectively. The color represent the reward value of parameterized policy with corresponding parameter values. The brighter it is, the larger the value is. And for a general policy across tasks with more parameters than a single task, there will always be a point for optimal policy across tasks.

Reward-Parameter Map of the Meta-Initialization Policy (Second)

For the meta-initialization policy, the neural network is supposed to contain the same numbers of parameters as a policy for single task, different from above general policy. Therefore there may not be an global optimal point across tasks but only some suboptimal points for each task, as shown in Fig. 29.

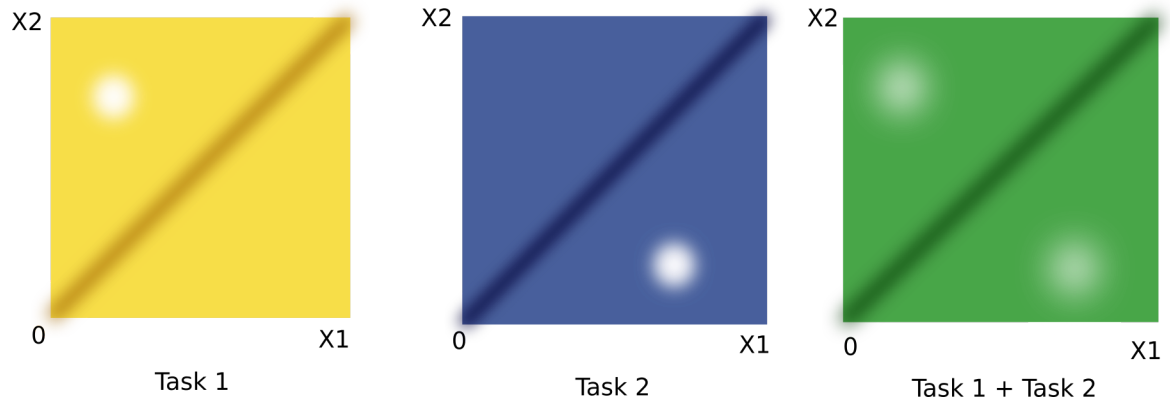

Figure 29: Figure of reward-parameter map for meta-initialized policy across tasks. X_1, X_2 are parameters for single-task policy. The left and middle are reward maps for each of two tasks. And the right one is the reward map of the meta-initialization policy. The color represent the reward value of parameterized policy with corresponding parameter values. The brighter it is, the larger the value is. For meta-initialization policy, there may not be an global optimal point across tasks, but some suboptimal points for each task.

Fig. 30 shows how meta-learning makes the problem of learning across tasks easier with the meta-policy as initialization instead of a random initialization. We can also see that the meta-update could end up with a bad policy for every task but close to every optimal policy of each task in parameter space. **The meta-update in MAML actually tends to drift the initial policy to a point in parameter space where all optimal policies of each tasks in the task domain are isotropic for the meta-policy**.

4.1.3 A potential framwork of modified MAML

Current MAML only keeps track of one step of adaptive updates, and update the initial policy parameter θ to be $\theta'_{\mathcal{T}}$. **The goal of MAML is to make the adaptively updated policy $\pi_{\theta'_{\mathcal{T}}}$ to have lower averaged loss value, instead of the initial policy π_θ. Therefore, the resulting policy of this objective is a policy with a one-step update for a specific task to have lowest loss**.

However, in practice, we do not expect the initialized policy to learn to be optimal policy with only one-step update, but N steps. We can keep track of N-step adaptive updates and unroll the gradients for Nth orders in the training phase for a more practical use. The resulting policy under this modified N-step MAML will be a policy with a N-step update for a specific task to have lowest loss.

Fig. 31 shows a demo of computation graph for simple multiplication operations with a 4th order (4-step) MAML using Tensorflow.

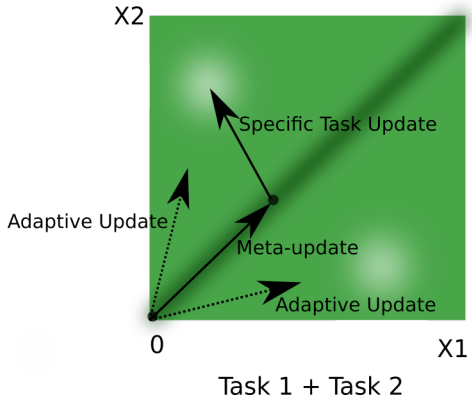

Figure 30: Figure of reward-parameter map for meta-initialized policy across tasks using MAML with fixed-step stochastic gradient descent . X_1, X_2 are parameters for meta-initialized policy. The color represent the reward value of parameterized policy with corresponding parameter values. The brighter it is, the larger the value is. The dashed arrows shows the adaptive phase update for each task, and the solid arrow shows the possible meta-update. And the final solid arrow shows the update for a specific task after the initialization using the meta-policy. It shows how meta-learning makes the problem of learning across tasks easier with the meta-policy as initialization instead of a random initialization.

4.1.4 Reptile + PPO

4.1.5 MAML + PPO

Bibliography

[1] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[3] Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2013.

[4] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

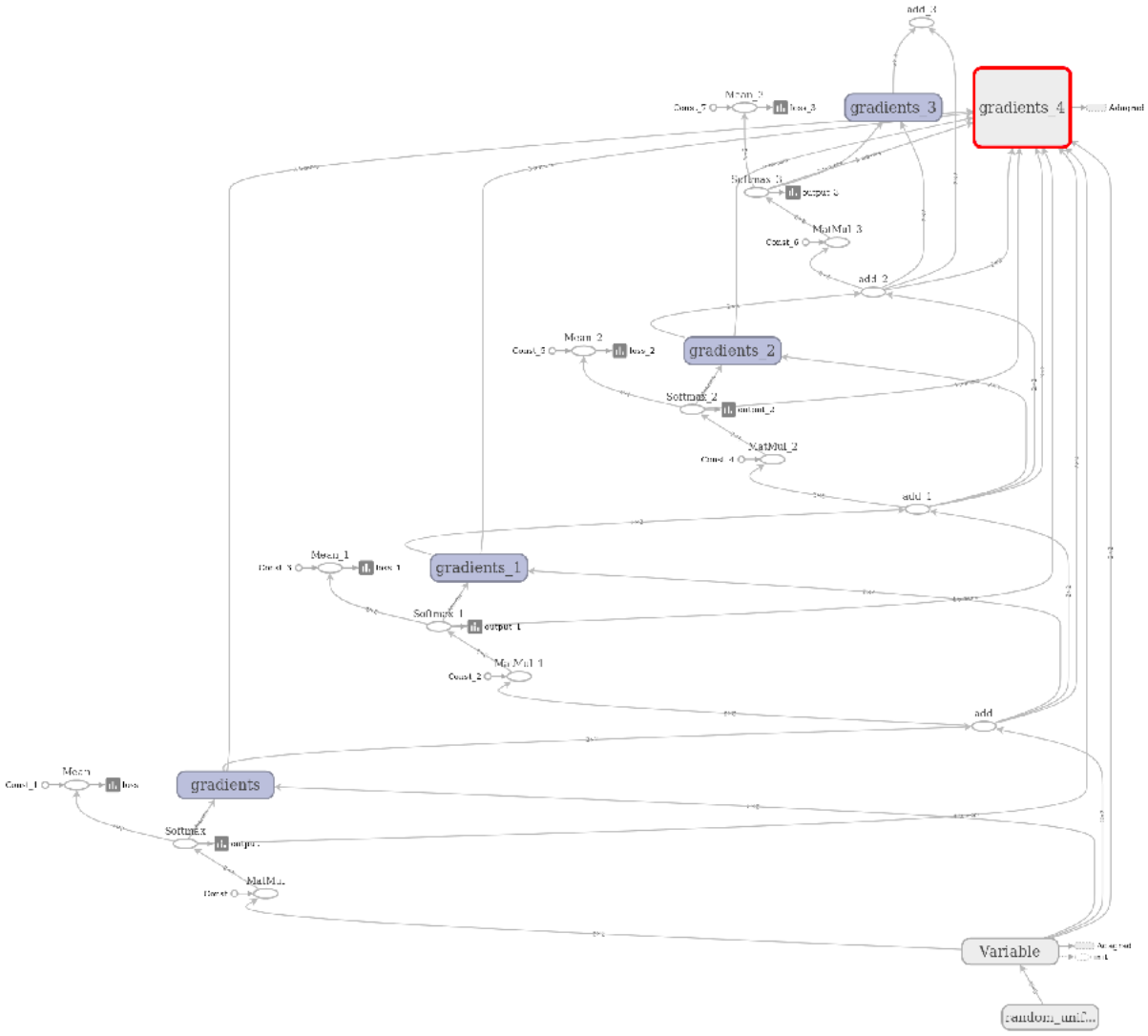

Figure 31: A demo computation graph of a 4th order MAML with Tensorflow.

[7] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[8] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[9] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *Icml*, volume 37, pages 1889–1897, 2015.

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[11] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[12] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[13] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[14] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[15] Tom Blau, Lionel Ott, and Fabio Ramos. Improving reinforcement learning pre-training with variational dropout. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4115–4122. IEEE, 2018.

[16] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.

[17] Peter Pastor, Heiko Hoffmann, Tamim Asfour, and Stefan Schaal. Learning and generalization of motor skills by learning from demonstration. In *2009 IEEE International Conference on Robotics and Automation*, pages 763–768. IEEE, 2009.

[18] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. In *Advances in neural information processing systems*, pages 1087–1098, 2017.

[19] Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. Residual reinforcement learning for robot control. *arXiv preprint arXiv:1812.03201*, 2018.

[20] Yinlam Chow and Mohammad Ghavamzadeh. Algorithms for cvar optimization in mdps. In *Advances in neural information processing systems*, pages 3509–3517, 2014.

[21] Matej Večerík, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

[22] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[23] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[24] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms. *CoRR, abs/1803.02999*, 2, 2018.

[25] Bradly C Stadie, Ge Yang, Rein Houthooft, Xi Chen, Yan Duan, Yuhuai Wu, Pieter Abbeel, and Ilya Sutskever. Some considerations on learning to explore via meta-reinforcement learning. *arXiv preprint arXiv:1803.01118*, 2018.

[26] Chelsea Finn, Kelvin Xu, and Sergey Levine. Probabilistic model-agnostic meta-learning. In *Advances in Neural Information Processing Systems*, pages 9516–9527, 2018.

[27] Risto Vuorio, Shao-Hua Sun, Hexiang Hu, and Joseph J Lim. Toward multimodal model-agnostic meta-learning. *arXiv preprint arXiv:1812.07172*, 2018.

[28] Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. *arXiv preprint arXiv:1709.04905*, 2017.

[29] Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot imitation from observing humans via domain-adaptive meta-learning. *arXiv preprint arXiv:1802.01557*, 2018.

[30] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.

[31] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl ^{2}: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[32] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[33] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[34] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141*, 2017.

[35] Tristan Deleu and Yoshua Bengio. The effects of negative adaptation in model-agnostic meta-learning. *arXiv preprint arXiv:1812.02159*, 2018.

[36] Antreas Antoniou, Harrison Edwards, and Amos Storkey. How to train your maml. *arXiv preprint arXiv:1810.09502*, 2018.