对抗性动作与训练代替奖励函数

摘要

在机器人的强化学习训练中,对奖励函数的参数限制过少往往会导致其在现实中表现异常。为了改善异常行为，训练者常常设置十分复杂的奖励函数，鼓励机器人做出表现正常的行为。然而，过于复杂的奖励函数往往需要手动设置并调整，难以跨平台跨任务的使用奖励函数。对此，本文采用“风格奖励”代替复杂的奖励函数，通过动作捕捉演示数据集进行训练，采用“风格奖励”与对抗预训练结合，使机器人的动作趋于自然。

关键词：强化学习；对抗预训练

1 引言

对于多足机器人连续移动的自然步态一直是控制类的研究重点。其早期工作侧重于针对特定环境下，对机器人进行特定参数的限制，并令其实现预期目标。然而，基于此类算法的强化训练产生的控制系统往往是高度专业化的，这限制了系统在多个其他环境进行适配，在其它领域进行迁移的能力。近期，使用强化学习的算法训练移动参数在模拟环境下产生了不错的效果，但在其对于现实环境的迁移仍是没有得到好的效果。在计算机图形学领域，用 GAN 从参考训练集中学习，以“风格”奖励代替常规的奖励函数产生轨迹分布，使数据集的轨迹分布于策略轨迹分布偏差最小化，本文引用该方法用于训练机器人的移动轨迹，并发现，采用该方法可降低训练成本，还会产生自然的步态，在不同速度下更节省能量。

2 相关工作

2.1 采用深度强化学习控制机器人

机器人领域的最新工作表面，将深度强化学习运用于各类机器人控制任务取得了很好的效果，深度强化学习给控制策略提供参数，使控制系统不需要手动设置。然而，深度强化学习对控制器的训练会导致机器人产生不稳定，不自然的步态，这些行为会使机器人的偏差函数最小化，但不符合实际运用需求，因此，常常需要认为设置机器人参数以接近现实需求。但这些参数的设置会使控制系统针对特定任务特化，需要大量训练符合任务需求，不符合通用任务需要。

2.2 动作模仿

动作模仿为开发广泛技能的控制器提供了一种通用方法，否则很难手动编码到控制器中。这些技术通常利用某种形式的运动跟踪，其中控制器通过明确跟踪参考轨迹指定的目标姿态序列来模仿所需的运动。在模拟领域，运动跟踪与强化学习相

结合已被证明对再现大量复杂和动态的运动技能非常有效。虽然运动跟踪对于模仿单个运动片段是非常有效的，但跟踪目标往往会约束控制器密切跟踪参考运动，这会限制根据需要开发更通用和多样化的行为以实现辅助任务目标的能力。此外，应用基于跟踪的技术来模仿不同运动数据集的行为将会很困难，并产生额外的运算，如运动规划器和运动片段的特定任务注释。在本文中，我们使用了一种基于对抗性模仿学习的更灵活的运动模仿方法，该方法允许我们的系统使用非结构化运动数据集来塑造智能体的行为，同时也为智能体提供了更大的灵活性，以根据需要开发新的行为，从而实现任务目标。

2.3 对抗性模仿学习

对抗性模仿学习为模仿来自不同演示数据集的行为提供了一种灵活且可扩展的方法。对抗性技术旨在学习与数据集的轨迹分布相匹配的策略，而不是明确地跟踪单个运动片段这可以为代理在数据集中显示的行为之间进行组合和插值提供更大的灵活性。这是通过训练对抗性鉴别器来区分策略产生的行为和演示数据中描述的行为来实现的。然后，鉴别器作为训练控制策略以模仿演示的风格奖励。虽然这些方法在低维领域中显示出了不错的结果，但当应用于高维连续控制任务时，这些方法产生的动作演示远远落后于最先进的基于跟踪的技术。最近，对抗性运动先验（AMP）被提出，它将对抗性模仿学习与辅助任务目标相结合，从而使模拟代理能够执行高级任务，同时模仿大型非结构化运动数据集的行为。我们将利用这种对抗性技术来学习多足机器人的运动技能。本文将表明，学习到的运动先验会导致更自然、更合理、更节能的行为，从而更容易从模拟转移到真实世界的机器人。

3 本文方法

3.1 以对抗性动作优先顺序作为风格奖励

对抗性模仿学习为模仿来自不同演示数据集的行为提供了一种灵活且可扩展的方法。对抗性技术旨在学习与数据集的轨迹分布相匹配的策略，而不是明确地跟踪单个运动片段这可以为代理在数据集中显示的行为之间进行组合和插值提供更大的灵活性。这是通过训练对抗性鉴别器来区分策略产生的行为和演示数据中描述的行为来实现的。然后，鉴别器作为训练控制策略以模仿演示的风格奖励。虽然这些方法在低维领域中显示出了不错的结果，但当应用于高维连续控制任务时，这些方法产生的动作演示远远落后于最先进的基于跟踪的技术。最近，对抗性运动先验（AMP）被提出，它将对抗性模仿学习与辅助任务目标相结合，从而使模拟代理能够执行高级任务，同时模仿大型非结构化运动数据集的行为。我们将利用这种对抗性技术来学习多足机器人的运动技能。本文将表明，学习到的运动先验会导致更自然、更合理、更节能的行为，从而更容易从模拟转移到真实世界的机器人。

3.2 特征提取及损失函数

原文对机器狗腿部运动问题建模为马尔科夫决策过程$(\mathcal{S}, \mathcal{A}, f, r_t, p_0, \gamma)$，将机器狗的四肢运动角速度，电机动力建模为需要学习的特征，并将机器狗与数据集在 x，y 轴上的速度与 z 轴上的角速度的差值建立为运动损失，将机器狗与数据集的状态差异建立为风格函数损失，合成损失函数。

4 复现细节

4.1 已有开源代码

```
class AMPPPO:
    actor_critic: ActorCritic
    def __init__(self,
                 actor_critic,
                 discriminator,
                 amp_data,
                 amp_normalizer,
                 num_learning_epochs=1,
                 num_mini_batches=1,
                 clip_param=0.2,
                 gamma=0.998,
                 lam=0.95,
                 value_loss_coef=1.0,
                 entropy_coef=0.0,
                 learning_rate=1e-3,
                 max_grad_norm=1.0,
                 use_clipped_value_loss=True,
                 schedule="fixed",
                 desired_kl=0.01,
                 device='cpu',
                 amp_replay_buffer_size=100000,
                 min_std=None,
                 ):

        self.device = device

        self.desired_kl = desired_kl
        self.schedule = schedule
        self.learning_rate = learning_rate
        self.min_std = min_std

        # Discriminator components
        self.discriminator = discriminator
```

```
            self.discriminator.to(self.device)
            self.amp_transition = RolloutStorage.Transition()
            self.amp_storage = ReplayBuffer(
                discriminator.input_dim // 2, amp_replay_buffer_size, device)
            self.amp_data = amp_data
            self.amp_normalizer = amp_normalizer

            # PPO components
            self.actor_critic = actor_critic
            self.actor_critic.to(self.device)
            self.storage = None # initialized later

            # Optimizer for policy and discriminator.
            params = [
                {'params': self.actor_critic.parameters(), 'name': 'actor_critic'},
                {'params': self.discriminator.trunk.parameters(),
                 'weight_decay': 10e-4, 'name': 'amp_trunk'},
                {'params': self.discriminator.amp_linear.parameters(),
                 'weight_decay': 10e-2, 'name': 'amp_head'}]
            self.optimizer = optim.Adam(params, lr=learning_rate)
            self.transition = RolloutStorage.Transition()

            # PPO parameters
            self.clip_param = clip_param
            self.num_learning_epochs = num_learning_epochs
            self.num_mini_batches = num_mini_batches
            self.value_loss_coef = value_loss_coef
            self.entropy_coef = entropy_coef
            self.gamma = gamma
            self.lam = lam
            self.max_grad_norm = max_grad_norm
            self.use_clipped_value_loss = use_clipped_value_loss

        def init_storage(self, num_envs, num_transitions_per_env, actor_obs_shape,
critic_obs_shape, action_shape):
            self.storage = RolloutStorage(
                num_envs, num_transitions_per_env, actor_obs_shape, critic_obs_shape,
action_shape, self.device)

        def test_mode(self):
            self.actor_critic.test()

        def train_mode(self):
            self.actor_critic.train()
```

```
    def act(self, obs, critic_obs, amp_obs):
        if self.actor_critic.is_recurrent:
            self.transition.hidden_states = self.actor_critic.get_hidden_states()
        # Compute the actions and values
        aug_obs, aug_critic_obs = obs.detach(), critic_obs.detach()
        self.transition.actions = self.actor_critic.act(aug_obs).detach()
        self.transition.values = self.actor_critic.evaluate(aug_critic_obs).detach()
        self.transition.actions_log_prob =
self.actor_critic.get_actions_log_prob(self.transition.actions).detach()
        self.transition.action_mean = self.actor_critic.action_mean.detach()
        self.transition.action_sigma = self.actor_critic.action_std.detach()
        # need to record obs and critic_obs before env.step()
        self.transition.observations = obs
        self.transition.critic_observations = critic_obs
        self.amp_transition.observations = amp_obs
        return self.transition.actions

    def process_env_step(self, rewards, dones, infos, amp_obs):
        self.transition.rewards = rewards.clone()
        self.transition.dones = dones
        # Bootstrapping on time outs
        if 'time_outs' in infos:
            self.transition.rewards += self.gamma * torch.squeeze(self.transition.values *
infos['time_outs'].unsqueeze(1).to(self.device), 1)

        not_done_idxs = (dones == False).nonzero().squeeze()
        self.amp_storage.insert(
            self.amp_transition.observations, amp_obs)

        # Record the transition
        self.storage.add_transitions(self.transition)
        self.transition.clear()
        self.amp_transition.clear()
        self.actor_critic.reset(dones)

    def compute_returns(self, last_critic_obs):
        aug_last_critic_obs = last_critic_obs.detach()
        last_values = self.actor_critic.evaluate(aug_last_critic_obs).detach()
        self.storage.compute_returns(last_values, self.gamma, self.lam)

    def update(self):
        mean_value_loss = 0
        mean_surrogate_loss = 0
```

```
        mean_amp_loss = 0
        mean_grad_pen_loss = 0
        mean_policy_pred = 0
        mean_expert_pred = 0
        if self.actor_critic.is_recurrent:
            generator =
self.storage.reccurent_mini_batch_generator(self.num_mini_batches,
self.num_learning_epochs)
        else:
            generator = self.storage.mini_batch_generator(self.num_mini_batches,
self.num_learning_epochs)

        amp_policy_generator = self.amp_storage.feed_forward_generator(
            self.num_learning_epochs * self.num_mini_batches,
            self.storage.num_envs * self.storage.num_transitions_per_env //
                self.num_mini_batches)
        amp_expert_generator = self.amp_data.feed_forward_generator(
            self.num_learning_epochs * self.num_mini_batches,
            self.storage.num_envs * self.storage.num_transitions_per_env //
                self.num_mini_batches)
        for sample, sample_amp_policy, sample_amp_expert in zip(generator,
amp_policy_generator, amp_expert_generator):

                obs_batch, critic_obs_batch, actions_batch, target_values_batch,
advantages_batch, returns_batch, old_actions_log_prob_batch, \
                    old_mu_batch, old_sigma_batch, hid_states_batch, masks_batch =
sample
                aug_obs_batch = obs_batch.detach()
                self.actor_critic.act(aug_obs_batch, masks=masks_batch,
hidden_states=hid_states_batch[0])
                actions_log_prob_batch =
self.actor_critic.get_actions_log_prob(actions_batch)
                aug_critic_obs_batch = critic_obs_batch.detach()
                value_batch = self.actor_critic.evaluate(aug_critic_obs_batch,
masks=masks_batch, hidden_states=hid_states_batch[1])
                mu_batch = self.actor_critic.action_mean
                sigma_batch = self.actor_critic.action_std
                entropy_batch = self.actor_critic.entropy

                # KL
                if self.desired_kl != None and self.schedule == 'adaptive':
                    with torch.inference_mode():
                        kl = torch.sum(
```

```
                                    torch.log(sigma_batch / old_sigma_batch + 1.e-5) +
(torch.square(old_sigma_batch) + torch.square(old_mu_batch - mu_batch)) / (2.0 *
torch.square(sigma_batch)) - 0.5, axis=-1)
                                kl_mean = torch.mean(kl)

                                if kl_mean > self.desired_kl * 2.0:
                                    self.learning_rate = max(1e-5, self.learning_rate / 1.5)
                                elif kl_mean < self.desired_kl / 2.0 and kl_mean > 0.0:
                                    self.learning_rate = min(1e-2, self.learning_rate * 1.5)

                                for param_group in self.optimizer.param_groups:
                                    param_group['lr'] = self.learning_rate


                    # Surrogate loss
                    ratio = torch.exp(actions_log_prob_batch -
torch.squeeze(old_actions_log_prob_batch))
                    surrogate = -torch.squeeze(advantages_batch) * ratio
                    surrogate_clipped = -torch.squeeze(advantages_batch) * torch.clamp(ratio,
1.0 - self.clip_param,

1.0 + self.clip_param)
                    surrogate_loss = torch.max(surrogate, surrogate_clipped).mean()

                    # Value function loss
                    if self.use_clipped_value_loss:
                        value_clipped = target_values_batch + (value_batch -
target_values_batch).clamp(-self.clip_param,

self.clip_param)
                        value_losses = (value_batch - returns_batch).pow(2)
                        value_losses_clipped = (value_clipped - returns_batch).pow(2)
                        value_loss = torch.max(value_losses, value_losses_clipped).mean()
                    else:
                        value_loss = (returns_batch - value_batch).pow(2).mean()

                    # Discriminator loss.
                    policy_state, policy_next_state = sample_amp_policy
                    expert_state, expert_next_state = sample_amp_expert
                    if self.amp_normalizer is not None:
                        with torch.no_grad():
                            policy_state = self.amp_normalizer.normalize_torch(policy_state,
self.device)
```

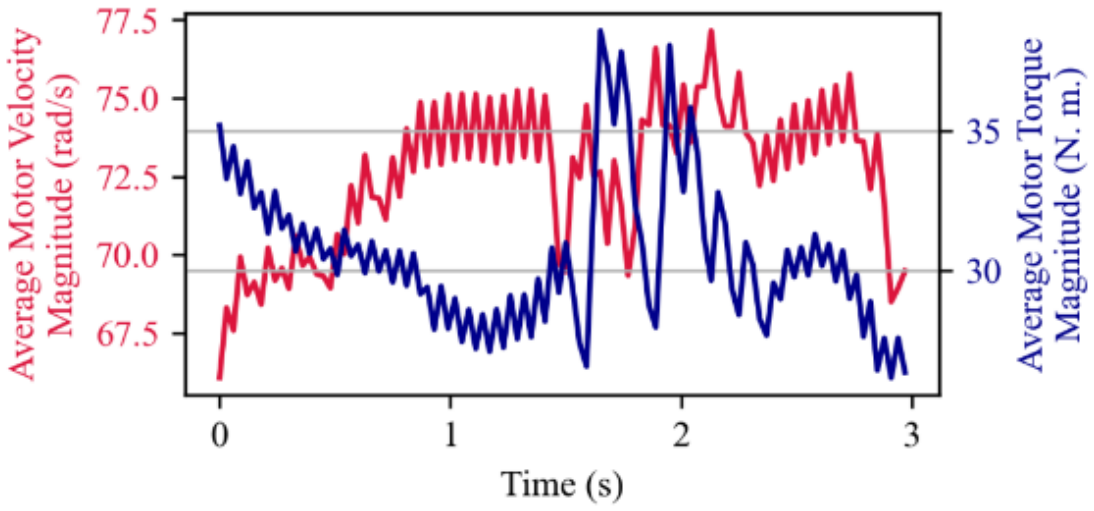
```
                                policy_next_state =
self.amp_normalizer.normalize_torch(policy_next_state, self.device)
                                expert_state = self.amp_normalizer.normalize_torch(expert_state,
self.device)
                                expert_next_state =
self.amp_normalizer.normalize_torch(expert_next_state, self.device)
                        policy_d = self.discriminator(torch.cat([policy_state, policy_next_state],
dim=-1))
                        expert_d = self.discriminator(torch.cat([expert_state, expert_next_state],
dim=-1))
                        expert_loss = torch.nn.MSELoss()(
                            expert_d, torch.ones(expert_d.size(), device=self.device))
                        policy_loss = torch.nn.MSELoss()(
                            policy_d, -1 * torch.ones(policy_d.size(), device=self.device))
                        amp_loss = 0.5 * (expert_loss + policy_loss)
                        grad_pen_loss = self.discriminator.compute_grad_pen(
                            *sample_amp_expert, lambda_=10)

                        # Compute total loss.
                        loss = (
                            surrogate_loss +
                            self.value_loss_coef * value_loss -
                            self.entropy_coef * entropy_batch.mean() +
                            amp_loss + grad_pen_loss)

                        # Gradient step
                        self.optimizer.zero_grad()
                        loss.backward()
                        nn.utils.clip_grad_norm_(self.actor_critic.parameters(),
self.max_grad_norm)
                        self.optimizer.step()

                        if not self.actor_critic.fixed_std and self.min_std is not None:
                            self.actor_critic.std.data =
self.actor_critic.std.data.clamp(min=self.min_std)

                        if self.amp_normalizer is not None:
                            self.amp_normalizer.update(policy_state.cpu().numpy())
                            self.amp_normalizer.update(expert_state.cpu().numpy())

                        mean_value_loss += value_loss.item()
                        mean_surrogate_loss += surrogate_loss.item()
                        mean_amp_loss += amp_loss.item()
                        mean_grad_pen_loss += grad_pen_loss.item()
```

```
                mean_policy_pred += policy_d.mean().item()
                mean_expert_pred += expert_d.mean().item()

        num_updates = self.num_learning_epochs * self.num_mini_batches
        mean_value_loss /= num_updates
        mean_surrogate_loss /= num_updates
        mean_amp_loss /= num_updates
        mean_grad_pen_loss /= num_updates
        mean_policy_pred /= num_updates
        mean_expert_pred /= num_updates
        self.storage.clear()

        return mean_value_loss, mean_surrogate_loss, mean_amp_loss,
mean_grad_pen_loss, mean_policy_pred, mean_expert_pred
```

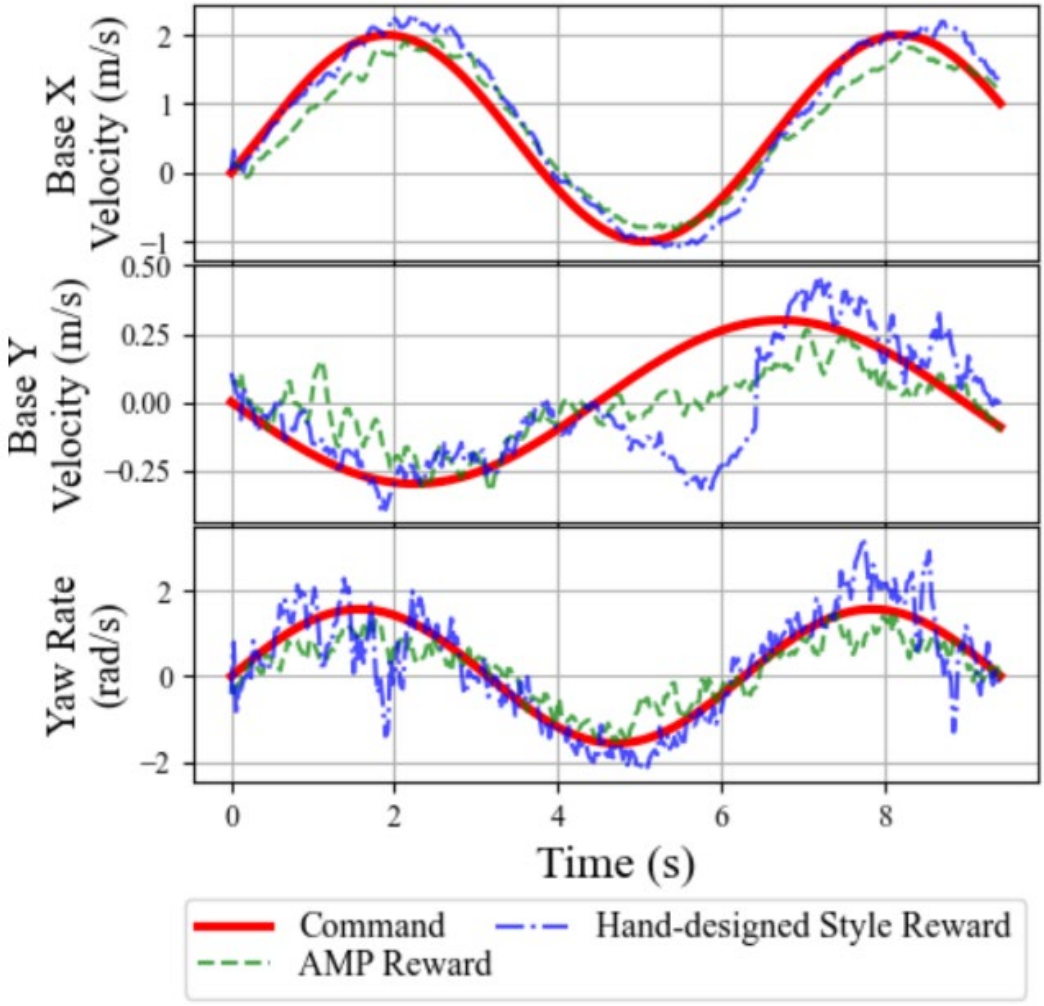
对源代码的引用没有做出修改

5 实验结果分析

文章在模拟中对三种奖励函数，即无风格奖励、对抗性运动先验风格奖励、复杂风格奖励的性能进行对比，此外，还额外对比了运输成本，运输成本常用语腿部运动领域，允许对不同的机器人或控制器进行能效比较



图中得知，使用 AMP 训练成功跟踪了向前速度，且表现出比竞争基线更低的运输成本。同时，无风格奖励的训练策略对机器人的腿部大幅移动以跟上所需速度，但同时带来了更大的振动，有着损坏机器人的风险，难以用于真实的机器人上。如下图所示，手动设计参数的机器人运输成本 COT 在 1.37 至 1.65 间浮动，而对抗性运动的风格奖励产生的 COT 在 0.93 至 1.12 间浮动。



6 总结和展望

通过使用对抗性模仿学习以产生风格奖励，文章避免了手动设计复杂的参数的需要，而对于对现实的应用，对抗性运动避免了大幅度动作带来的损伤。此外，文章比较了手动设计参数、AMP 风格奖励函数和无风格奖励策略的能效对比。AMP 风格奖励函数不仅能运用于四足机器人的运动，也能运用于其他类型的机器人，未来可考虑将 AMP 风格奖励函数运用于其他领域。

参考文献

[1] Escontrela A, Peng X B, Yu W, et al. Adversarial motion priors make good substitutes for complex reward functions[C]//2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2022: 25-32.