

13

Learning to Run

在这一章，我们提供了一个实践项目，方便读者获得一些深度强化学习应用的经验，而这个项目是一个由 CrowdAI 和 NeurIPS¹ 2017 主办的挑战：**Learning to Run**。这个环境有 41 维的状态空间和 18 维的动作空间，二者都是连续的，因此，对于初学者获取经验而言是一个较大规模的环境。我们为这个任务提供了一个柔性 Actor-Critic（Soft Actor-Critic，SAC）算法的解决方案，同时也有一些辅助技巧来提高其表现。环境和代码链接见读者服务。

13.1 NeurIPS 2017 挑战：Learning to Run

13.1.1 环境介绍

Learning to Run 是一个由 CrowdAI 和 NeurIPS 2017 举办的竞赛，吸引了许多强化学习研究人员的参与。在这个任务中，参与者被要求开发一个控制器，使得一个生理学人体模型可以尽可能快地通过一条复杂而有障碍物的路线。任务中提供了一个肌肉骨骼模型和一个基于物理过程模拟环境。为了模拟这个物理和生物力学过程并用强化学习智能体对其导航，任务提供了一个基于 OpenSim 库的 osim-rl 环境，而 OpenSim 是一个对肌肉骨骼建模的标准物理和生物力学环境。如图 13.1 所示是一个包括主体在内的环境场景。

这个环境结合了一个包括两条腿、一个骨盆、一个代表上半身的部分（躯干、头部、手臂）的肌肉骨骼模型。不同部分之间由关节连接（比如膝关节和髋），而这些关节处的活动由肌肉激发来控制。模型中的这些肌肉有很复杂的路径（比如，肌肉可以经过不止一个关节，并且模型中有冗余肌肉），而肌肉激励器本身也是高度非线性的。这个智能体在 3 维世界中进行 2 维运动。为

¹该会议名称当时缩写为 NIPS。

为了便于理解和操作，我们在这个项目中使用的环境相比挑战赛中使用的有所简化，因此可能有些地方跟竞赛官方文档略微不同。如下是骨骼模型的所有组成部分。

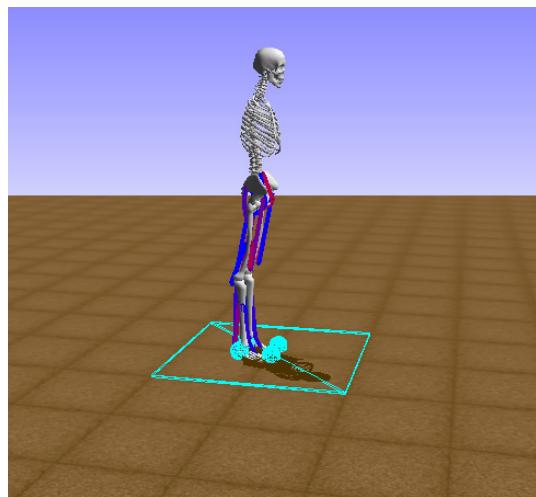


图 13.1 NeurIPS 2017 挑战赛: Learning to Run 环境 (见彩插)

- 观察量包括 41 个值:
 - * 骨盆位置 (角度, x 坐标, y 坐标)
 - * 骨盆速度 (角速度, x 速度, y 速度)
 - * 每个踝关节、膝关节和髋的角度 (6 个值)
 - * 每个踝关节、膝关节和髋的角速度 (6 个值)
 - * 质心位置 (2 个值)
 - * 质心速度 (2 个值)
 - * 头部、骨盆、躯干、左右脚趾、左右踝的位置 (共 14 个值)
 - * 左右腰肌强度: 对难度级别低于 2 (难度值是一个默认环境参数) 的环境, 其值为 1.0, 否则它是一个随机变量, 在整个模拟周期中采样于均值为 1.0、标准差为 0.1 的固定正态分布。(注意: 在我们简化的环境中, 这些腰肌强度值被设为 0.0。)
 - * 下一个障碍物: 到骨盆的 x 轴距离, 以及其中心相对地面的 y 坐标。(注意: 在我们的简化环境中, 所有这些值被设为 0.0, 无障碍物出现。)
- 动作包括 18 个标量值, 分别表示 18 块肌肉的激发程度 (每条腿 9 个):
 - * 胫绳肌腱
 - * 股二头肌
 - * 臀大肌
 - * 髂腰肌

- * 股直肌
- * 股肌
- * 腓肠肌
- * 比目鱼肌
- * 胫骨前肌

- 奖励函数:

- * 奖励函数由骨盆沿 x 轴运动距离减去由于使用韧带的惩罚计算得到。

- 其他细节:

- * “done” 信号表示这一步是环境模拟的最后一步。这会在 1000 次迭代到达或者骨盆高度低于 0.65 米时发生。

从以上对环境的描述中我们可以看出，相比于其他 OpenAI Gym 或 DeepMind Control Suite 中的游戏，这个竞赛的环境相对复杂，有着高维观察量空间和动作空间。因此，以较好的表现和较短的训练时间解决这个任务需要一些特殊的技巧。我们将介绍这些具体方法及用一个并行训练框架来解决这个任务。我们在随书的代码库中提供了这个环境的副本和解决方案的代码，因此，我们推荐读者用这个项目进行上手练习。

13.1.2 安装

根据官方库，这个环境可以用以下命令行安装：

1. 创建一个包含 OpenSim 软件包的 Conda 环境（命名为 opensim-rl）。

```
conda create -n opensim-rl -c kidzik opensim python=3.6.1
```

2. 激活我们刚创建的 Conda 环境。

在 Windows 上，运行：

```
activate opensim-rl
```

在 Linux/OS X 上，运行：

```
source activate opensim-rl
```

你需要在每次打开一个新的终端时输入上面的命令。

3. 安装我们的 Python 强化学习环境。

```
conda install -c conda-forge lapack git  
pip install osim-rl
```

自从 2017 年以后，这个挑战已连续举办了三年（至 2019 年）。因而，最初的 Learning to Run 环境由于版本更新已经被废弃。虽然如此，我们仍旧选择用这个原始的 2017 版本环境来做示范，因为它相对简单。于是，在我们的项目中提供了一个仓库存放 2017 版本的环境：

```
git clone  
https://github.com/deep-reinforcement-learning-book/Chapter13-Learning-to-Run.git
```

我们所用的强化学习算法代码和环境的封装也都在上述仓库中提供。

通过以上几步，我们已经完成了环境的安装，可以通过以下命令检验安装是否成功：

```
python -c "import opensim"
```

如果它能正常运行，说明安装已成功；否则，可以在这个网站找到解决方案。

要用随机采样执行 200 次模拟迭代，我们可以用 Python 解释器运行以下命令（在 Linux 环境）：

```
from osim.env import RunEnv # 导入软件包  
env = RunEnv(visualize=True) # 初始化环境  
observation = env.reset(difficulty = 0) # 重置环境  
for i in range(200): # 采集样本  
    observation, reward, done, info = env.step(env.action_space.sample())
```

这个环境由于已被写成 OpenAI Gym 游戏的格式，对用户十分友好，而且有一个定义好的奖励函数。我们的任务是得到一个从当前观察量（一个 41 维矢量）到肌肉激活动作（18 维矢量）的映射函数，使得它能够最大化奖励值。如前所述，奖励函数被定义为一个迭代步中骨盆沿 x 轴的位移减去韧带受力大小，从而尽可能鼓励智能体在最小身体损耗的情况下向前移动。

13.2 训练智能体

为了更好地解决这个任务，在训练框架中需要实现一系列技巧，包括：

- 一个可以平衡 CPU 和 GPU 资源的并行训练框架；
- 奖励值缩放；
- 指数线性单元 (Exponential Linear Unit, ELU) 激活函数；
- 层标准化 (Layer Normalization)；
- 动作重复；
- 更新重复；
- 观察量标准化和动作离散化可能是有用的，但我们未在提供的解决方案中使用；

- 根据智能体双腿的对称性所做的数据增强可能是用的，但我们未在提供的解决方案中使用。

注意，根据竞赛参与团体的实验和报告，后两个技巧也可能是有用的，但由于它们更多基于该具体任务的方法而不对其他任务广泛适用，我们未在这里的解决方案中使用。然而，要知道观察量标准化、动作值离散化和数据增强是可以根据一些任务的具体情况应用来加速学习过程的。

这个环境一个典型的缺陷是模拟速度太慢，在一个普通 CPU 上完成单个片段至少需要几十秒时间。为了更高效地学习策略，我们需要将采样和训练过程并行化。

13.2.1 并行训练

至少有两个原因需要我们对这个任务进行并行训练。第一个是由于上面所述 Learning to Run 环境较慢的模拟速度，至少耗时几十秒完成一个模拟片段。第二个是由于该环境有较高的内在复杂度。基于作者经验，这个环境用普通的无模型（Model-Free）强化学习算法，如深度决定性策略梯度（Deep Deterministic Policy Gradient, DDPG）或柔性 Actor-Critic（Soft Actor-Critic, SAC），需要至少上百个 CPU/CPU 计算小时来获得一个较好的策略。因此，这里需要一个多进程跨 GPU 的训练框架。

由于 Learning to Run 环境的高复杂度，训练过程需要用多个 CPU 和 GPU 来并行分布实现。此外，CPU 和 GPU 之间的平衡对这个任务也很关键，因为与环境交互采样的过程一般是在 CPU 上，而反向传播训练过程一般是在 GPU 上。整个过程的训练效率在实践中满足短板效应。关于并行训练中如何均衡 CPU 和 GPU 计算的内容在第 12 章和第 18 章中也有讨论。这里有一种解决这个任务的方案。

如图 13.2 所示，在一般的单进程深度强化学习中，训练过程由一个进程来处理，而这通常无法充分发挥计算资源的潜力，尤其在有多个 CPU 核和多个 GPU 的情况下。

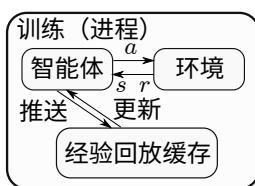


图 13.2 在离线策略深度强化学习中进行单进程训练：只有一个进程来采样和训练策略

图 13.3 展示了在多个 CPU 和多个 GPU 上部署离线策略（Off-Policy）深度强化学习的并行训练架构，其中，一个智能体和一个环境被封装进一个“工作者”来运行一个进程。多个工作者可以共享同一个 GPU，因为有时单个工作者无法完全占用整个 GPU 内存。在这种设置下，使用同一个 GPU 的进程数量和工作者数量可以被手动设置，从而在学习过程中最大化所有计算资源的利用率。

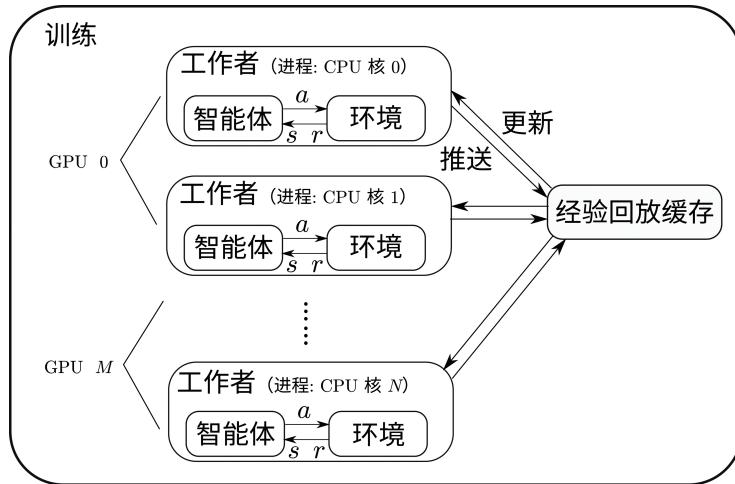


图 13.3 一个离线策略深度强化学习的并行训练架构。每个工作者包含一个与环境交互的智能体，策略被分布在多个 GPU 上训练

我们的项目提供了一个高度并行化的 SAC 算法，它使用上述架构来解决这个需要多进程和多 GPU 计算的任务。由于多进程的内存之间互相不共享，需要用特殊的模块来处理信息交流和参数共享。在代码中，回放缓冲区通过 Python 内的 `multiprocessing` 模块共享，训练过程中的网络和参数更新由 PyTorch 的 `multiprocessing` 模块共享（在 Linux 系统上）。

实践中，尽管每个工作者包含一个智能体，但是智能体内的网络实际在多个工作者间共享，因此实际上只保留了一套网络（用于一个智能体的）。PyTorch 的 `nn.Module` 模块可以处理使用多个进程更新共享内存中网络参数的情况。由于 Adam 优化器在训练中也有一些统计量，我们使用以下 `ShareParameters()` 函数来在多进程中共享这些值：

```

def ShareParameters(adamoptim):
    # 共享 Adam 优化器的参数便于实现多进程
    for group in adamoptim.param_groups:
        for p in group['params']:
            state = adamoptim.state[p]
            # 初始化：需要在这里初始化，否则无法找到相应量
            state['step'] = 0
            state['exp_avg'] = torch.zeros_like(p.data)
            state['exp_avg_sq'] = torch.zeros_like(p.data)

            # 在内存中共享
            state['exp_avg'].share_memory_()
            state['exp_avg_sq'].share_memory_()

```

在训练函数中，我们用以下方式设置 SAC 算法中的共享模块，包括网络和优化器：

```
# 共享网络
sac_trainer.soft_q_net1.share_memory()
sac_trainer.soft_q_net2.share_memory()
sac_trainer.target_soft_q_net1.share_memory()
sac_trainer.target_soft_q_net2.share_memory()
sac_trainer.policy_net.share_memory()

# 共享优化器参数
ShareParameters(sac_trainer.soft_q_optimizer1)
ShareParameters(sac_trainer.soft_q_optimizer2)
ShareParameters(sac_trainer.policy_optimizer)
ShareParameters(sac_trainer.alpha_optimizer)
```

`share_memory()` 是一个继承自 PyTorch 的 `nn.Module` 模块的函数，可用于共享神经网络。我们也可以共享熵因子，但是在这个代码里没有实现它。“forkserver” 启动方法是在 Python 3 中使用 CUDA 子进程所需的，如代码中所示：

```
torch.multiprocessing.set_start_method('forkserver', force=True)
```

回放缓冲区可以用 Python 的 `multiprocessing` 模块共享：

```
from multiprocessing.managers import BaseManager

replay_buffer_size = 1e6
BaseManager.register('ReplayBuffer', ReplayBuffer)
manager = BaseManager()
manager.start()
replay_buffer = manager.ReplayBuffer(replay_buffer_size)

# 通过 manager 来共享经验回放缓存
```

在克隆下来的文件夹中运行以下命令来开始训练（注意，由于使用“forkserver”启动方法，所以在 Windows 10 上无法进行这样的并行训练）：

```
python sac_learn.py --train
```

我们也可用以下命令测试训练的模型：

```
python sac_learn.py --test
```

13.2.2 小技巧

然而，即便使用了上面的并行架构，我们仍旧不能在这个任务上取得很好的表现。由于任务的复杂性和深度学习模型的非线性，损失函数上的局部最优和非平滑甚至不可微的曲面都容易使优化过程陷入困境（对于策略或价值函数）。在使用深度强化学习方法的过程中经常需要一些微调策略，尤其是对像 Learning to Run 这样的复杂任务。所以，下面介绍我们使用的一些小技巧，来更高效和稳定地解决这个任务。

- **奖励值缩放：**奖励值缩放遵循一般的值缩放规则，即将奖励值除以训练过程中所采批样本的标准差。奖励值缩放，或叫标准化和归一化，是强化学习中使训练过程稳定而加速收敛速度的常用技术手段。如 SAC 算法后续的一篇文章 (Haarnoja et al., 2018) 所报道的，最大熵强化学习算法可能对奖励函数的缩放敏感，这不同于其他传统强化学习算法。因此，SAC 算法的作者添加了一个基于梯度的温度调校模块用作熵正则化项，这显著缓解了实践中超参数微调过程的困难。
- **指数线性单元 (Exponential Linear Unit, ELU) (Clevert et al., 2015)** 激活函数被用以替代整流线性单元 (Rectified Linear Unit, ReLU) (Agarap, 2018)：为了得到更快的学习过程和更好的泛化表现，我们使用 ELU 作为策略网络隐藏层的激活函数。ELU 函数定义如下：

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha \exp(x - 1), & \text{if } x \leq 0 \end{cases} \quad (13.1)$$

ELU 和 ReLU 的对比如图 13.4 所示。相比于 ReLU，ELU 有负数值，这使得它能够将神经单元激活的平均值拉至更接近 0 的位置，如同批标准化，但是却有着更低的计算复杂度。均值移动到趋于 0 可以加速学习，因为它通过减少神经单元激发造成的移动偏差，使得一般的梯度更加接近于神经网络单元的自然梯度。

- **层标准化：**我们也对价值网络和策略网络的每个隐藏层使用层标准化 (Ba et al., 2016)。相比于批标准化 (Batch Normalization)，层标准化对单个训练样本在某神经网络层上的神经元的累加输入计算均值和方差来进行标准化。每个神经元有其与众不同的适应性偏差 (Bias) 和增益 (Gain)，这些值在标准化之后和非线性激活之前被添加到神经元的值上。这种方法在实际中可以帮助加速训练过程。
- **动作重复：**我们在训练过程中使用一个常见的技巧叫动作重复（或叫跳帧），来加速训练的执行时间 (Wall-Clock Time)。DQN 原文中使用跳帧和像素级的最大化 (Max) 算子来实现在 Atari 2600 游戏上基于图像的学习。如果我们定义单个帧的原始观察量是 o_i ，其中 i 表示帧指标，原始 DQN 文章中的输入是 4 个堆叠帧，其中每个是两个连续帧中的最大值，即 $[\max(o_{i-1}, o_i), \max(o_{i+3}, o_{i+4}), \max(o_{i+7}, o_{i+8}), \max(o_{i+11}, o_{i+12})]$ ，对应的跳帧率就是 4（实际上，对于不同游戏，该跳帧率可以是 2,3 或 4）。在这些跳过的帧中，动作被重复执

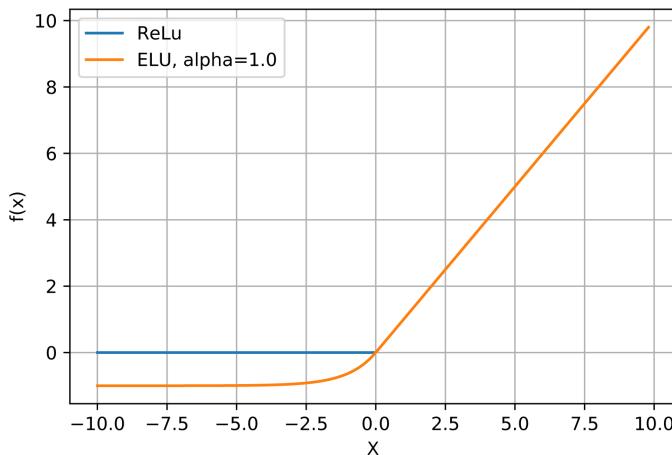


图 13.4 对比 ReLU 和 ELU 激活函数。ELU 在零点可微

行。最大化算子在图像观察量上按像素计算，奖励函数对所有跳过和不跳过的帧累加。原始 DQN 中的跳帧机制增加了随机性，同时加速了采样率。然而，在我们的任务中，我们使用一种不同的设置，不使用最大化算子和堆叠帧：每个动作在跳过的帧上进行简单的重复执行，包括跳过帧和未跳过帧在内的所有样本被存入回放缓冲区。实践中，我们使用 3 作为动作重复率，减少了策略与环境交互所需的正向推理时间。

- **更新重复：**我们也在训练中使用一个小的学习率并重复更新策略的技巧，从而策略以重复率 3 在同一个批样本上进行学习。

13.2.3 学习结果

通过以上设置和 SAC 算法上的这些小技巧，智能体能够在 3 天的训练时长下学会用人类的方式奔跑很长的一段距离，训练是在一个 4GPU 和 56CPU 的服务器上进行的，结果如图 13.5 所示。图 13.6 展示了学习曲线，包括原始的奖励函数值和移动平均的平滑曲线，呈现了上升的学习表现。纵轴是一个片段内的累计奖励值，显示了智能体奔跑的距离和姿势状况。

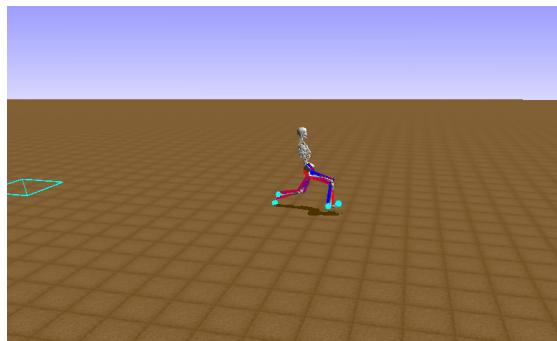


图 13.5 Learning to Run 任务中奔跑智能体的最终表现 (场景) (见彩插)

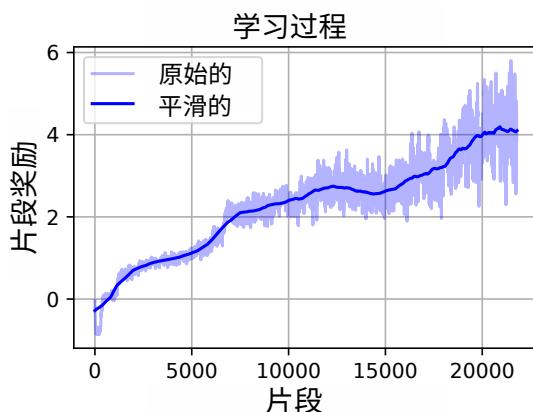


图 13.6 Learning to Run 任务的学习过程 (见彩插)

参考文献

- AGARAP A F, 2018. Deep learning using rectified linear units (relu)[Z].
- BA J L, KIROS J R, HINTON G E, 2016. Layer normalization[Z].
- CLEVERT D A, UNTERTHINER T, HOCHREITER S, 2015. Fast and accurate deep network learning by exponential linear units (elus)[Z].
- HAARNOJA T, ZHOU A, HARTIKAINEN K, et al., 2018. Soft actor-critic algorithms and applications[J]. arXiv preprint arXiv:1812.05905.

14

鲁棒的图像增强

深度生成模型相较于经典的算法，在超分辨率、图像分割等计算机视觉任务中，取得了显著的进展。然而，这种基于学习的方法缺乏鲁棒性和可解释性，限制了它们在现实世界中的应用。本章将讨论一种鲁棒的图像增强方法，它可以通过深度强化学习与许多可解释的技术进行结合。我们将先从一些图像增强的背景知识进行介绍，接着将图像增强的过程看作一个由马尔可夫决策过程（Markov Decision Process, MDP）建模的处理流程。最后，我们将展示如何通过近端策略优化（Proximal Policy Optimization, PPO）算法构建智能体来处理这个 MDP 过程。实验环境由一个真实世界的数据集构建，包含 5000 张照片，其中包括原始图像和专家调整后的版本。项目代码链接见读者服务。

14.1 图像增强

图像增强技术属于图像处理技术。它的主要目标是使处理后的图像更适合各种应用的需要。典型的图像增强技术包括去噪、去模糊和亮度改善。现实世界中的图像总是需要多种图像增强技术。图 14.1 显示了一个包括亮度改善和去噪的图像增强流程。专业的照片编辑软件，如 Adobe Photoshop，提供强大的修图能力，但效率不高，需要用户在照片编辑方面具备专业知识。在诸如推荐系统这样的大规模场景中，图像的主观质量对用户体验至关重要，因此需要一种满足有效性、鲁棒性和效率的自动图像增强方法。其中鲁棒性是最重要的条件，尤其是在用户生成内容的平台上，比如 Facebook 和 Twitter，即使 1% 的较差情况（Bad Case）也会伤害数百万用户的使用体验。

与图像分类或分割有着其独特的真实值（Ground Truth）不同，图像增强的训练数据依赖于人类专家。因此，图像增强并没有大规模的公共图像增强数据集。经典的图像增强方法主要基于



图 14.1 一个图像增强流程的案例。左侧的原始图像存在 JPEG 压缩噪声并且曝光不足（见彩插）

的是伽马校正和直方图均衡化，以及先验的专家知识。这些方法也不需要大量的数据。伽马校正利用了人类感知的非线性，比如，我们感知光和颜色的能力。直方图均衡化实现了允许局部对比度较低的区域获得更高的对比度，以更好地分布在像素直方图上的思想，这在背景和前景为全亮或全暗（如 X 射线图像）时非常有用。这些方法虽然快速、简单，但是缺乏对上下文信息的考虑，限制了它们的应用。

最近，有学者使用基于学习的方法，试图用 CNN 拟合从输入图像到所需像素值的映射，并取得了很大的成功 (Bychkovsky et al., 2011; Kupyn et al., 2018; Ulyanov et al., 2018; Wang et al., 2019)。然而，这种方法也存在问题。首先，很难训练出一个能处理多种增强情况的综合神经网络。此外，像素到像素的映射缺乏鲁棒性，例如，在处理诸如头发和字符等细节信息时，它的表现不是很好 (Nataraj et al., 2019; Zhang et al., 2019)。一些研究者提出，将深度强化学习应用于图像增强，将增强过程描述为一系列策略迭代问题，以解决上述问题。在本章中，我们遵循这些方法，并提出一种新的 MDP 公式来进行图像增强。我们在一个包含 5000 对图像的数据集上用代码示例演示了我们的方法，以提供快速的实际学习过程。

在讨论算法之前，我们先介绍两个 Python 库：Pillow (Clark, 2015) 和 scikit-image (Van der Walt et al., 2014)。它们提供了许多友好的接口来实现图像增强。可以使用如下代码直接从 PyPI 安装它们：

```
pip install Pillow
pip install scikit-image
```

下面是 Pillow 的子模块 ImageEnhance 调整对比度的示例代码。

```
from PIL import ImageEnhance

def adjust_contrast(image_rgb, contrast_factor):
    # 调整对比度
    # 参数：
    # image_rgb (PIL.Image): RGB 图像
```

```
# contrast_factor (float): 颜色平衡因子范围从 0 到 1.
# 返回:
# PIL.Image 对象
#
enhancer = ImageEnhance.Contrast(image_rgb)
return enhancer.enhance(contrast_factor)
```

14.2 用于鲁棒处理的强学习

在将强化学习应用于图像增强时，首先需要考虑如何构造该领域的马尔可夫决策过程。一个自然出现的想法是将像素处理为状态，将不同的图像增强技术视为强化学习的动作。该构想提供了几种可控的初级增强算法的组合方法，以获得稳健、有效的结果。在本节中，我们将讨论这种基于强化学习的颜色增强方法。为了简单起见，我们只采取全局增强操作。值得一提的是，通过区域候选模块 (Ren et al., 2015) 来适应一般的增强算法也是很自然的想法。

假设训练集包含 N 对 RGB 图像 $\{(l_i, h_i)\}_{i=1}^N$ ，其中 l_i 为低质量原始图像， h_i 是高质量修复图像。为了保持数据分布，初始状态 S_0 应从 $\{l_i\}_{i=1}^N$ 中均匀采样。在每步中，智能体会执行一个预定义对动作，如调整对比度，再将它应用于当前状态。需要注意的是，当前状态和选择动作完全决定了状态转移。也就是说，环境没有不确定性。我们在之前工作的基础上 (Furuta et al., 2019; Park et al., 2018) 上继续研究，并使用了 CIELAB 颜色空间作为转移奖励函数。

$$\|L(h) - L(S_t)\|_2^2 - \|L(h) - L(S_{t+1})\|_2^2 \quad (14.1)$$

其中 h 是对应的高质量图像 S_0 ， L 是 RGB 颜色空间到 CIELAB 颜色空间的映射。

另一个重点是定义学习和评估时的终结状态。在游戏的强化学习应用中，终结状态可以由环境决定。而与此不同的是，在图像增强中的智能体需要由自己决定退出时机。文献 (Park et al., 2018) 提出了一个基于 DQN 的智能体，它在所有动作预测的 Q 值都为负数时会退出。然而，Q-Learning 中由函数近似引起的过估计问题可能会导致推理过程的鲁棒性降低。我们通过训练一个明确的策略并增加一个用于退出选择的“无操作”动作来处理这个问题。表 14.1 列出了所有预定义的动作，其中索引为 0 的动作表示“无操作”动作。

从零开始训练一个卷积神经网络需要大量的原始-修复图像对。因此，我们不使用原始图像状态作为观测值的方案，而是考虑使用在 ILSVRC 分类数据集 (Russakovsky et al., 2015) 上预训练的 ResNet50 网络中的最后一层卷积层的激活值作为深层特征输入。这样的深层特征十分重要，它可以提升许多其他视觉识别任务的效果 (Redmon et al., 2016; Ren et al., 2016)。受到前人工作 (Lee et al., 2005; Park et al., 2018) 的启发，我们在构造观测信息时进一步考虑使用直方图信息。具体来说，我们计算了 RGB 颜色空间在 $(0, 255), (0, 255), (0, 255)$ 范围内和 CIELab 颜色空间在 $(0, 100),$

表 14.1 全局图像增强动作集

索引	简介	索引	简介
0	无操作	7	红、绿色调整 $\times 0.95$
1	对比度 $\times 0.95$	8	红、绿色调整 $\times 1.05$
2	对比度 $\times 1.05$	9	蓝、绿色调整 $\times 0.95$
3	饱和度 $\times 0.95$	10	蓝、绿色调整 $\times 1.05$
4	饱和度 $\times 1.05$	11	红、蓝色调整 $\times 0.95$
5	亮度 $\times 0.95$	12	红、蓝色调整 $\times 1.05$
6	亮度 $\times 1.05$		

$(-60, 60), (-60, 60)$ 范围内的统计信息。这三个特征连成 $2048 + 2000$ 维的观测信息。接着，我们选择 PPO (Schulman et al., 2017) 作为策略优化算法。PPO 是一种 Actor-Critic 算法，它在一系列任务上已经取得了显著的成果。它的网络由 3 部分组成：3 层特征抽取作为主干网络、1 层行动者 (Actor) 网络和 1 层批判者 (Critic) 网络。所有层都是全连接的，其中特征抽取器中各层的输出分别为 2048、512 和 128 个单元，并都使用了 ReLU 作为激活函数。

我们在 MIT-Adobe FiveK (Bychkovsky et al., 2011) 数据集上对我们的方法进行了评估。其中包括 5000 张原始图像，而每张原始图像又有 5 个不同专家 (A/B/C/D/E) 修复后的图像。继之前的工作 (Park et al., 2018; Wang et al., 2019) 之后，我们只使用专家 C 修复的图像，并随机选择 4500 张图像进行训练，剩下的 500 张图像用于测试。原始图像是 DNG 格式的，而修复图像是 TIFF 格式的。我们使用 Adobe Lightroom 将它们都转换为质量为 100、颜色空间为 sRGB 的 JPEG 格式。为了更有效地训练，我们也调整了图像大小，使得每张图像的最大边为 512 像素。具体的超参数在表 14.2 中列出。

表 14.2 用于图像增强的 PPO 超参数

超参数	值	超参数	值
优化器	Adam	每次迭代的优化数	2
学习率	$1e-5$	最大迭代数	10000
梯度范数裁剪	1.0	熵因子	$1e-2$
GAE λ	0.95	奖励缩放	0.1
每次迭代的片段数	4	γ	0.95

接下来开始，我们将演示如何实现上述算法，首先需要构建一个环境对象。

```
class Env(object):
    # 训练环境
```

```
def __init__(self, src, max_episode_length=20, reward_scale=0.1):
    # 参数:
    # src (list[str, str]): 原始图像和处理图像路径的列表, 初始状态将从中均匀采样
    # max_episode_length (int): 最大可执行动作数量
    self._src = src
    self._backbone = backbone
    self._preprocess = preprocess
    self._rgb_state = None
    self._lab_state = None
    self._target_lab = None
    self._current_diff = None
    self._count = 0
    self._max_episode_length = max_episode_length
    self._reward_scale = reward_scale
    self._info = dict()
```

通过使用 TensorFlow 的 ResNet API, 我们可以通过 `_state_feature` 函数构建观测数据。过程如下所示:

```
backbone = tf.keras.applications.ResNet50(include_top=False, pooling='avg')
preprocess = tf.keras.applications.resnet50.preprocess_input
```

```
def get_lab_hist(lab):
    # 获取 Lab 图像的直方图
    lab = lab.reshape(-1, 3)
    hist, _ = np.histogramdd(lab, bins=(10, 10, 10),
                            range=((0, 100), (-60, 60), (-60, 60)))
    return hist.reshape(1, 1000) / 1000.0

def get_rgb_hist(rgb):
    # 获取 RGB 图像的直方图
    rgb = rgb.reshape(-1, 3)
    hist, _ = np.histogramdd(rgb, bins=(10, 10, 10),
                            range=((0, 255), (0, 255), (0, 255)))
    return hist.reshape(1, 1000) / 1000.0

def _state_feature(self):
    s = self._preprocess(self._rgb_state)
```

```
s = tf.expand_dims(s, axis=0)
context = self._backbone(s).numpy().astype('float32')
hist_rgb = get_rgb_hist(self._rgb_state).astype('float32')
hist_lab = get_lab_hist(self._lab_state).astype('float32')
return np.concatenate([context, hist_rgb, hist_lab], 1)
```

接着，我们构建和 OpenAI Gym (Brockman et al., 2016) 相同的接口。其中，我们按照表 14.1 定义转移函数 `_transit`，并依据公式 (14.1) 构建奖励函数 `_reward`。

```
def step(self, action):
    # 执行单步
    self._count += 1
    self._rgb_state = self._transit(action)
    self._lab_state = rgb2lab(self._rgb_state)
    reward = self._reward()
    done = self._count >= self._max_episode_length or action == 0
    return self._state_feature(), reward, done, self._info

def reset(self):
    # 重置环境
    self._count = 0
    raw, retouched = map(Image.open, random.choice(self._src))
    self._rgb_state = np.asarray(raw)
    self._lab_state = rgb2lab(self._rgb_state)
    self._target_lab = rgb2lab(np.asarray(retouched))
    self._current_diff = self._diff(self._lab_state)
    self._info['max_reward'] = self._current_diff
    return self._state_feature()
```

这里的 PPO 与 5.10.6 节实现有所不同。我们将 PPO (Schulman et al., 2017) 算法用于离散动作情况。需要注意的是，我们将 LogSoftmax 作为行动者网络的激活函数，这样能在计算替代目标时提供更好的数值稳定性。对 PPO 智能体，我们先定义它的初始化函数和行为函数：

```
class Agent(object):
    # PPO 智能体
    def __init__(self, feature, actor, critic, optimizer,
                 epsilon=0.1, gamma=0.95, c1=1.0, c2=1e-4, gae_lambda=0.95):
        # 参数：
        # feature (tf.keras.Model): 行动者和批判者的基础网络
```

```
# actor (tf.keras.Model): 行动者网络
# critic (tf.keras.Model): 批判者网络
# optimizer (tf.keras.optimizers.Optimizer): 优化器
# epsilon (float): 裁剪操作中的 epsilon
# gamma (float): 奖励折扣
# c1 (float): 价值损失系数
# c2 (float): 熵系数
self.feature, self.actor, self.critic = feature, actor, critic
self.optimizer = optimizer

self._epsilon = epsilon
self.gamma = gamma
self._c1 = c1
self._c2 = c2
self.gae_lambda = gae_lambda

def act(self, state, greedy=False):
    # 参数:
    # state (numpy.array): 1 * 4048 维的状态
    # greedy (bool): 是否要选取贪心动作
    # Returns:
    # action (int): 所选择的动作
    # logprob (float): 所选动作的概率对数
    # value (float): 当前状态的价值
    feature = self.feature(state)
    logprob = self.actor(feature)
    if greedy:
        action = tf.argmax(logprob[0]).numpy()
        return action, 0, 0
    else:
        value = self.critic(feature)
        logprob = logprob[0].numpy()
        action = np.random.choice(range(len(logprob)), p=np.exp(logprob))
        return action, logprob[action], value.numpy()[0, 0]
```

在采样过程中，我们通过 GAE (Schulman et al., 2015) 算法记录轨迹。

```
def sample(self, env, sample_episodes, greedy=False):
    # 从给定环境中采样一个轨迹
    # 参数：
    # env: 给定的环境
    # sample_episodes (int): 要采样多少片段
    # greedy (bool): 是否选取贪心动作
    trajectories = [] # s, a, r, logp
    e_reward = 0
    e_reward_max = 0
    for _ in range(sample_episodes):
        s = env.reset()
        values = []
        while True:
            a, logp, v = self.act(s, greedy)
            s_, r, done, info = env.step(a)
            e_reward += r
            values.append(v)
            trajectories.append([s, a, r, logp, v])
            s = s_
            if done:
                e_reward_max += info['max_reward']
                break
    episode_len = len(values)
    gae = np.empty(episode_len)
    reward = trajectories[-1][2]
    gae[-1] = last_gae = reward - values[-1]
    for i in range(1, episode_len):
        reward = trajectories[-i - 1][2]
        delta = reward + self.gamma * values[-i] - values[-i - 1]
        gae[-i - 1] = last_gae = \
            delta + self.gamma * self.gae_lambda * last_gae
    for i in range(episode_len):
        trajectories[-(episode_len - i)][2] = gae[i] + values[i]
    e_reward /= sample_episodes
    e_reward_max /= sample_episodes
    return trajectories, e_reward, e_reward_max
```

最后，策略优化的部分如下所示，其中价值损失裁剪和优势标准化遵循文献 (Dhariwal et al., 2017) 的描述。

```
def _train_func(self, b_s, b_a, b_r, b_logp_old, b_v_old):
    # 训练函数
    all_params = self.feature.trainable_weights + \
        self.actor.trainable_weights + \
        self.critic.trainable_weights
    with tf.GradientTape() as tape:
        b_feature = self.feature(b_s)
        b_logp, b_v = self.actor(b_feature), self.critic(b_feature)

        entropy = -tf.reduce_mean(
            tf.reduce_sum(b_logp * tf.exp(b_logp), axis=-1))
        b_logp = tf.gather(b_logp, b_a, axis=-1, batch_dims=1)
        adv = b_r - b_v_old
        adv = (adv - tf.reduce_mean(adv)) / (tf.math.reduce_std(adv) + 1e-8)

        c_b_v = b_v_old + tf.clip_by_value(b_v - b_v_old,
                                            -self._epsilon, self._epsilon)
        vloss = 0.5 * tf.reduce_max(tf.stack(
            [tf.pow(b_v - b_r, 2), tf.pow(c_b_v - b_r, 2)]), axis=1, axis=1)
        vloss = tf.reduce_mean(vloss)

        ratio = tf.exp(b_logp - b_logp_old)
        clipped_ratio = tf.clip_by_value(
            ratio, 1 - self._epsilon, 1 + self._epsilon)
        pgloss = -tf.reduce_mean(tf.reduce_min(tf.stack(
            [clipped_ratio * adv, ratio * adv], axis=1), axis=1))

        total_loss = pgloss + self._c1 * vloss - self._c2 * entropy
        grad = tape.gradient(total_loss, all_params)
        self.optimizer.apply_gradients(zip(grad, all_params))
    return entropy

def optimize(self, trajectories, opt_iter):
    # 基于给定轨迹数据进行优化
    b_s, b_a, b_r, b_logp_old, b_v_old = zip(*trajectories)
```

```

b_s = np.concatenate(b_s, 0)
b_a = np.expand_dims(np.array(b_a, np.int64), 1)
b_r = np.expand_dims(np.array(b_r, np.float32), 1)
b_logp_old = np.expand_dims(np.array(b_logp_old, np.float32), 1)
b_v_old = np.expand_dims(np.array(b_v_old, np.float32), 1)
b_s, b_a, b_r, b_logp_old, b_v_old = map(
    tf.convert_to_tensor, [b_s, b_a, b_r, b_logp_old, b_v_old])
for _ in range(opt_iter):
    entropy = self._train_func(b_s, b_a, b_r, b_logp_old, b_v_old)

return entropy.numpy()

```

最终经过训练后，智能体学到了图像增强的策略。图 14.2 展示了一个训练效果样例。



图 14.2 一个在 MIT-Adobe FiveK 数据集上使用全局增强的效果样例。当右上角的天空等区域需要局部增强时，全局亮度会增加（见彩插）

参考文献

-
- BROCKMAN G, CHEUNG V, PETTERSSON L, et al., 2016. OpenAI gym[J]. arXiv:1606.01540.
- BYCHKOVSKY V, PARIS S, CHAN E, et al., 2011. Learning photographic global tonal adjustment with a database of input/output image pairs[C]//CVPR 2011. IEEE: 97-104.
- CLARK A, 2015. Pillow (pil fork) documentation[Z].
- DHARIWAL P, HESSE C, KLIMOV O, et al., 2017. OpenAI baselines[J]. GitHub, GitHub repository.
- FURUTA R, INOUE N, YAMASAKI T, 2019. Fully convolutional network with multi-step reinforcement learning for image processing[C]//Proceedings of the AAAI Conference on Artificial Intelligence: volume 33. 3598-3605.

- KUPYN O, BUDZAN V, MYKHAILYCH M, et al., 2018. DeblurGAN: Blind motion deblurring using conditional adversarial networks[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 8183-8192.
- LEE S, XIN J, WESTLAND S, 2005. Evaluation of image similarity by histogram intersection[J]. Color Research & Application: Endorsed by Inter-Society Color Council, The Colour Group (Great Britain), Canadian Society for Color, Color Science Association of Japan, Dutch Society for the Study of Color, The Swedish Colour Centre Foundation, Colour Society of Australia, Centre Français de la Couleur, 30(4): 265-274.
- NATARAJ L, MOHAMMED T M, MANJUNATH B, et al., 2019. Detecting GAN generated fake images using co-occurrence matrices[J]. Journal of Electronic Imaging.
- PARK J, LEE J Y, YOO D, et al., 2018. Distort-and-recover: Color enhancement using deep reinforcement learning[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 5928-5936.
- REDMON J, DIVVALA S, GIRSHICK R, et al., 2016. You only look once: Unified, real-time object detection[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 779-788.
- REN S, HE K, GIRSHICK R, et al., 2015. Faster R-CNN: Towards real-time object detection with region proposal networks[C]//Advances in Neural Information Processing Systems. 91-99.
- REN S, HE K, GIRSHICK R, et al., 2016. Object detection networks on convolutional feature maps[J]. IEEE transactions on pattern analysis and machine intelligence, 39(7): 1476-1481.
- RUSSAKOVSKY O, DENG J, SU H, et al., 2015. Imagenet Large Scale Visual Recognition Challenge[J]. International Journal of Computer Vision (IJCV), 115(3): 211-252.
- SCHULMAN J, MORITZ P, LEVINE S, et al., 2015. High-dimensional continuous control using generalized advantage estimation[J]. arXiv preprint arXiv:1506.02438.
- SCHULMAN J, WOLSKI F, DHARIWAL P, et al., 2017. Proximal policy optimization algorithms[J]. arXiv:1707.06347.
- ULYANOV D, VEDALDI A, LEMPITSKY V, 2018. Deep image prior[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 9446-9454.
- VAN DER WALT S, SCHÖNBERGER J L, NUNEZ-IGLESIAS J, et al., 2014. scikit-image: image processing in python[J]. PeerJ, 2: e453.

WANG R, ZHANG Q, FU C W, et al., 2019. Underexposed photo enhancement using deep illumination estimation[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 6849-6857.

ZHANG S, ZHEN A, STEVENSON R L, 2019. GAN based image deblurring using dark channel prior[J]. arXiv preprint arXiv:1903.00107.