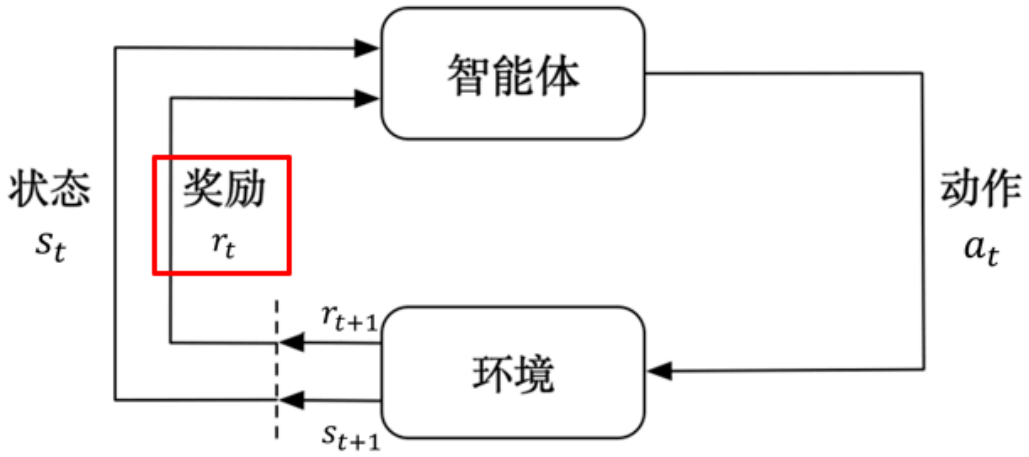
使用强化学习控制旋转倒立二阶摆

0. 背景简介

由于传统控制方法存在显著的固有问题，例如在硬件发生变动后需要重新进行繁杂的调参工作、控制算法本身较为复杂，泛化性差等，我们从而希望寻找一种更加通用、鲁棒的控制方法。那么，该方法必然是结合了数据驱动特性的—同时注意到，旋转二阶倒立摆的起摆任务有一个显著特点：目标轨迹不明确，也就是说，我们并没有一个明确的过程目标，而只有一个确定的最终态，即稳定在竖直向上的位置。在常见的数据驱动方法中，例如为人所熟知的监督学习与无监督学习都是依赖于已知数据的；与之对应地，强化学习将会考虑智能体（agent）与环境地交互，并对智能体的表现进行评价。基于此，该任务事实上符合强化学习的特点，故以下将从强化学习的角度来展开。

为了使用强化学习来控制旋转倒立二阶摆，我们必然需要解决 *sim2real* 的问题。强化学习的实现包含了对智能体的训练过程，而该过程通常需要大量的交互数据。由于旋转倒立二阶摆的物理模型较为复杂，直接在真实环境中进行训练会非常耗时，并且对硬件带来较大的磨损；同时硬件在长时间运行后，其行为也会产生变化。因此，常见的解决方案是在仿真环境中首先进行智能体的预训练，随后再将其迁移到真实环境中进行微调。该过程中所主要涉及的取舍是仿真环境与真实环境的差异，通常称为 *sim2real gap* —若此差异很小，那么迁移过程就会比较顺利，反之则需要采集较多的真实数据来进行微调；但是，想要缩小仿真环境与真实环境的差异，同样也需要耗费大量的时间与精力。更进一步地，强化学习方法还可以分为 model-free 与 model-based 两种。前者不需要对环境进行建模，而后者则需要对环境进行建模并利用模型来进行规划，较为先进的技术一般是使用一个神经网络来对环境进行建模。在此，限于时间与精力的限制，我们选择与传统控制的基本流程保持一致，因此使用了 model-free 的方法，并依赖于传统控制中建立的简化模型来进行仿真。



强化学习的基本流程

1. 强化学习算法

首先，我们对一些常用的强化学习算法进行了基本的调研与学习，如下表所示。然而仅从这个表格来介绍显然是不够直观的，因此我们转而从一些经典的强化学习例子出发。

> **Note**
>
> - 本次大作业中，笔者是从零开始学习强化学习，所以这些例子也都是来源于学习过程中所尝试复现的任务。此外，本文也可能会有一些作为初学者而表述地不够准确的地方。
> - 大部分例子将会收集在代码仓库中，如果觉得内容太少，大概率是因为笔者还没有更新仓库 ...

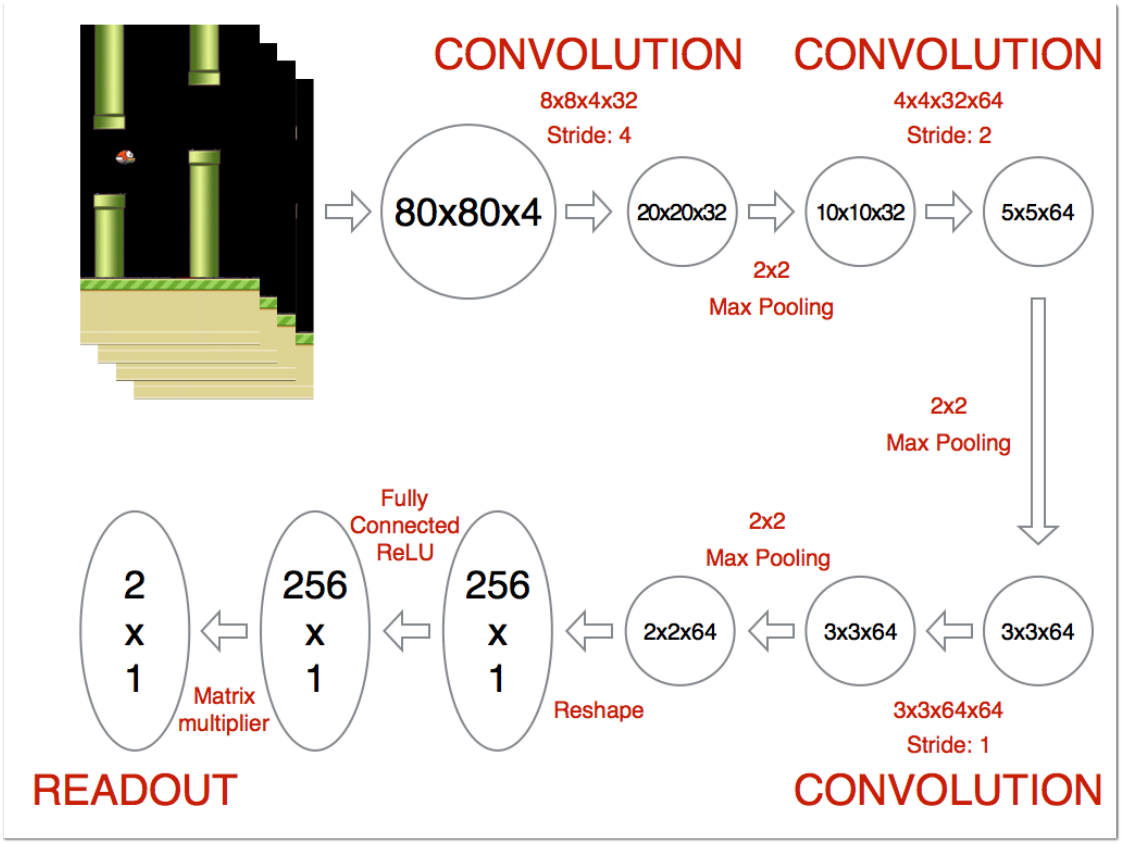
Name	`Box`	`Discrete`	`MultiDiscrete`	`MultiBinary`	Multi Processing
ARS	✔	✔	✘	✘	✔
A2C	✔	✔	✔	✔	✔
DDPG	✔	✘	✘	✘	✔
DQN	✘	✔	✘	✘	✔
HER	✔	✔	✘	✘	✔
PPO	✔	✔	✔	✔	✔
SAC	✔	✘	✘	✘	✔
TD3	✔	✘	✘	✘	✔
TRPO	✔	✔	✔	✔	✔

1.1. Flappy Bird

Flappy Bird 是一个非常经典的强化学习任务，主要是通过控制小鸟的上下移动来躲避障碍物。该任务的状态空间较小，动作空间也相对简单（点击屏幕/不点击，即一个 01 动作空间），因此很多入门级的强化学习教程都会使用 Flappy Bird 作为示例。该任务的目标是让小鸟尽可能长时间地存活下来，避免与障碍物碰撞。

> **What We Talk About When We Talk About RL?**
>
> 对于一个强化学习任务，我们首先需要明确其状态空间与动作空间。前者是指智能体在每个时间步所处的环境状态（state），而后者则是智能体在每个时间步可以选择的动作（action）。以 Flappy Bird 为例，其状态空间可以是小鸟的位置、速度以及障碍物的位置等信息，而动作空间则是小鸟的上下移动。从这里也可以看出，状态空间与智能体在实际环境中所能感知到的量，即观测（observation）空间，并不完全相同。例如，Flappy Bird 的状态空间可能包含了小鸟的加速度，但实际中它并没有 IMU，因此观测空间中事实上并不包含加速度。这进一步涉及到强化学习中的一个重要概念：*部分可观测 Markov 决策过程（POMDP）*， 但这里不再展开。

下图展示了一个利用 CNN 识别 Flappy Bird 游戏画面的强化学习结构。将游戏画面作为输入的好处是与实际应用场景更加接近，例如使用相机来获取环境状态。但相应地，其计算复杂度较高，训练速度也较慢。为了提高训练速度，通常会直接获取仿真环境当中的状态信息，而不是使用游戏画面推算这些信息。由于其动作空间是离散的，因此基于 DQN （Deep Q-Network）算法的强化学习方法可以较为容易地实现。DQN 是一种基于 Q-learning 的深度强化学习算法，它使用神经网络来近似 Q 函数，从而可以处理高维状态空间。

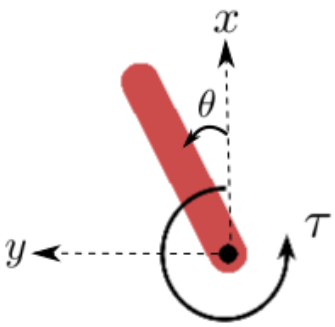


参考

- https://github.com/DebasmitaGhose/DQN_Flappy_Bird
- https://github.com/chncyhn/flappybird-qlearning-bot

1.2. Pendulum

我们注意到，DQN 算法非常方便（神经网络解决了大部分的体力劳动），但其在处理连续动作空间时会遇到困难。因此一种较为简单的解决方案是将连续动作空间离散化，从而可以使用 DQN 等离散动作空间的强化学习算法。Pendulum 任务的目标是通过控制摆杆的角度来保持其竖直向上，如下图所示（注意，此任务不同于**倒立摆**）。



该任务的原始动作空间是对摆杆施加的力矩，为了使用 DQN 算法，一种思路便是将力矩离散化为几个固定的值，例如 -2，-1，0，1，2。这可以简化计算，并允许研究者使用 DQN 算法来便捷地解决此问题。但相对应地，离散化后的控制精度也会降低，从而在更复杂的控制任务中，该思路可能不再适用。

参考

- https://gymnasium.farama.org/environments/classic_control/pendulum/

1.3. Rotary Inverted Double Pendulum

对于我们的旋转倒立二阶摆（Rotary Inverted Double Pendulum，RIDP）任务（为了与硬件配置相统一，我们将动作空间定义为电机的力矩，状态空间定义为三根杆的角度和角速度），如果沿用上面的离散化思路，那么动作空间就必须变得十分精细才能实现较好的控制效果，而这样会导致训练时间大幅增加。因此，这时候应当选择使用连续动作空间的强化学习算法。我们尝试使用了 PPO、SAC 和 TD3 等算法。其中，PPO 算法训练速度快，但是较为不稳定，依赖于较多的经验性技巧；TD3 算法在处理连续动作空间时表现较好，但训练速度较慢，并且对超参数敏感；SAC 算法则在训练速度和稳定性之间取得了较好的平衡。因此，之后的工作主要基于 SAC 算法进行。

2. 奖励函数设计

初步确定训练算法后，任务的核心便是设计一个合适的奖励函数，其直接影响到智能体的学习效果。我们先观察直线倒立二阶摆在稳摆任务中的奖励函数，如下所示：

直线倒立二阶摆的奖励函数

```python
def _get_rew(self, x, y, terminated):
    v0, v1 = self.data.qvel
    theta = self.data.qpos[0]
    dist_penalty = 0.01 * (x - 0.2159) ** 2 + (y - 0.5365) ** 2 + 0.02 * abs(theta)
    vel_penalty = 1e-4 * v0 + 1e-3 * v1**2
    alive_bonus = self._healthy_reward * int(not terminated)

    reward = alive_bonus - dist_penalty - vel_penalty

    reward_info = {
        "reward_survive": alive_bonus,
        "distance_penalty": -dist_penalty,
        "velocity_penalty": -vel_penalty,
    }

    return reward, reward_info
```

在此奖励函数中，主要包含存活奖励（alive_bonus）、距离惩罚（dist_penalty）和速度惩罚（vel_penalty）。存活奖励是为了鼓励摆尽可能保持在最高点的稳定状态，而距离惩罚和速度惩罚则主要是对其位姿进行限制。可以观察到，距离惩罚与速度惩罚中权重系数相对较小，这是因为若奖励函数限制地过于严苛，则智能体会倾向于采取较为保守的策略，从而导致学习效果不佳。此外，设计奖励函数时还需要考虑奖励变化的梯度，并针对训练效果微调。总得来说，一学期的调试经验表明，其设计原则可以概括为如下两条：

- 奖励函数应当反映目标要求，但不应过于严格。
- 奖励函数中各项奖励的梯度与其重要性应当成正相关，权重系数根据数值比例合理分配。

下面给出了我们最终训练旋转倒立二阶摆时所使用的奖励函数：

旋转倒立二阶摆的奖励函数

```python
def compute_reward_swingup_to_stabilize(self, x, y, terminated):
    # 完全竖直且速度为0时的奖励: 16.619
    theta0, theta1, theta2 = self.data.qpos
    v0, v1, v2 = self.data.qvel
    ctrl = (
        self.data.ctrl[0]
        if isinstance(self.data.ctrl, np.ndarray)
        else self.data.ctrl
    )
    ctrl = np.array(ctrl)

    x_goal, y_goal = 0.2159, 0.5365
    target_y = y_goal
```

```
    shift = 0.0

    swing_reward = 0.0
    posture_reward = 0.0
    alive_bonus = 0.0
    peak_slow_bonus = 0.0
    ctrl_penalty = 0.2 * np.sum(ctrl**2)

    uprightness = theta1**2 + (theta1 + theta2) ** 2
    angle_diff = abs(theta1 + theta2) + abs(theta1)

    if y < 0.3:
        if y < -0.32:
            theta_shift = abs(theta1 - np.pi) + abs(theta2)
            swing_reward = 1.2 * theta_shift

        if y > 0:
            swing_reward += 1.5

    if y >= 0.3 and y < 0.45:
        height_bonus = np.exp(-8 * (y - target_y) ** 2) * 3
        angle_bonus = (
            np.exp(-3 * abs(theta1 - theta2) - 5 * abs(np.sin(theta1))) * 1.5
        )
        posture_reward = height_bonus + angle_bonus
        ctrl_penalty *= 1.5

    if y >= 0.45:
        dist_penalty = 5.0 * ((x - x_goal) ** 2 + (y - y_goal) ** 2)
        angle_penalty = 0.1 * (theta1**2 + (theta1 + theta2) ** 2)
        ctrl_penalty *= 1.5
        speed_sum = abs(v1) + abs(v2)
        peak_slow_bonus = 2.0 * max(1.2 - speed_sum, 0.0)
        posture_reward = 4.0 * np.exp(-8 * uprightness - 8 * angle_diff) + 4 * y
        alive_bonus = (posture_reward + 3.0) * int(not terminated)
    else:
        dist_penalty = 0.1 * (x - x_goal) ** 2
        angle_penalty = 0.0

    base_vel_penalty = 7 * v0**2 + 3 * v1**2 + 3 * v2**2
    height_factor = 0.5 + 0.5 * np.tanh(5 * (y - 0.45))
    vel_penalty = base_vel_penalty * 7e-3 * height_factor + 0.04 * ctrl_penalty

    reward = (
        alive_bonus
        + swing_reward
        + peak_slow_bonus
        + posture_reward
        - dist_penalty
        - angle_penalty
        - vel_penalty
        - ctrl_penalty
        + shift
    )

    reward_info = {
        "reward_survive": alive_bonus,
        "swing_reward": swing_reward,
        "posture_reward": posture_reward,
        "distance_penalty": -dist_penalty,
        "angle_penalty": -angle_penalty,
        "velocity_penalty": -vel_penalty,
        "ctrl_penalty": -ctrl_penalty,
        "peak_slow_bonus": peak_slow_bonus,
        "uprightness": uprightness,
        "angle_diff": angle_diff,
        "y": y,
    }
```

```
    return reward, reward_info
```

3. 仿真实验结果

仿真实验结果请参考本项目的源代码仓库，其 `README.md` 文件中给出了倒立摆相关的实验过程及结果。

4. 硬件部署

然而，仿真与硬件部署几乎是截然不同的两个过程。在尝试中，我们发现，将仿真中训练好的智能体迁移到硬件上的难度远超于预期（根据算法复杂度和文献调研的估计）。主要原因在于以下几点：

- 仿真训练时，一个重要的参数是 mujoco 仿真中设置的 `timestep`，即仿真每一步的时间间隔。不同 `timestep` 下训练得到的强化学习模型不能直接相互迁移。而在硬件上，`timestep` 与设备的采样频率、实际控制频率、传感器的响应时间等因素密切相关，因此会出现智能体“以为”的时间间隔与实际时间间隔不一致的情况。这会严重影响智能体的表现及硬件训练效果。
- 硬件实验包含了大量噪声，即便仿真已经将这些噪声考虑在内，包括传感器读数噪声、电机噪声、环境噪声、响应延迟等，但是在实际部署时，这些噪声的影响仍然会显著降低智能体的表现。
- 仿真模型与实际硬件之间仍然有显著的差异。我们目前先尝试了对仿真的输出结果做线性拟合来缩小这种差异，但其效果并不理想。后续可能需要更精细地调整仿真模型，或直接利用 model-based 强化学习方法来对环境进行建模。

我们接下来主要讨论硬件部署中已经实现的效果，即模型的计算加速。为了与现有的代码框架相兼容，硬件部署时仍然选择使用 Python 语言来实现。然而 Python 语言的计算性能较差，例如，其在进行 SAC 模型推理时的速度仅为 100Hz 左右，而目标速度需要达到 1000Hz 以上。为此，我们需要从多个方面来实现计算的加速。同时需要注意的是，我们所使用的边缘计算设备并不包含 NPU，CPU 的计算性能也较差，因此我们可能需要一些较为激进的方法来实现加速。概括地讲，在经过一系列文献调研与基准测试之后，我们的加速方案为：在训练时使用 PyTorch + Intel® Extension for PyTorch（IPEX）框架，部署推理时使用 ONNX Runtime，并在硬件调用上使用 Windows ML。

4.1. 使用 ONNX Runtime

ONNX Runtime 是一个高性能的推理引擎，其主要功效是将训练好的模型转换为 ONNX 格式，并在推理时使用该引擎来加速计算。一个直观的例子时，利用 ONNX Runtime 优化后的 LLM 模型可以在核显轻薄本上实现推理速度的大幅翻倍。针对 SAC 模型的转换代码如下：

ONNX 模型转换代码

```
import torch

torch.onnx.export(
    actor,
    dummy_input,
    ONNX_PATH,
    input_names=["obs"],
    output_names=["action"],
    dynamic_axes={"obs": {0: "batch_size"}, "action": {0: "batch_size"}},
    opset_version=OPSET,
)
```

利用 ONNX Runtime 进行推理的代码如下。需要说明的是，`ort.InferenceSession` 的 `providers` 参数可以指定使用的计算设备，例如我们也测试了使用 Intel XPU 进行推理（替换为 *Intel® Extension for*

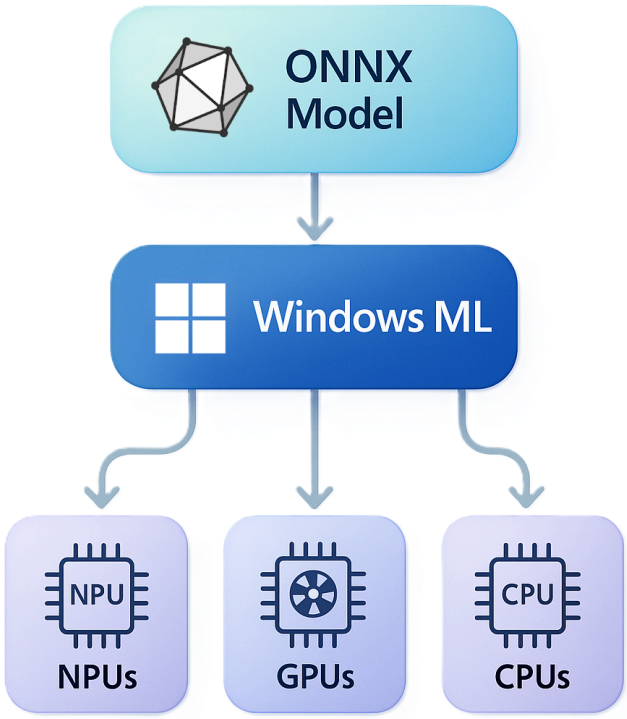
PyTorch，即 *IPEX* 后端），但是由于模型较小，批次大小也较小，因此在 CPU 上的推理速度已经足够快了。

ONNX 模型推理代码

```
import onnxruntime as ort

ort_session = ort.InferenceSession(ONNX_PATH, providers=["CPUExecutionProvider"])
action = sess.run(None, {input_name: samples_np[0:1]})
```

此外，在 ONNX Runtime 的硬件调用方面，我们还使用了 Windows ML 来实现更高效的硬件加速。Windows ML 本质上来说提供了在 Windows 系统上更高效使用硬件的接口，如下图所示：



4.2. 使用 Intel® Extension for PyTorch

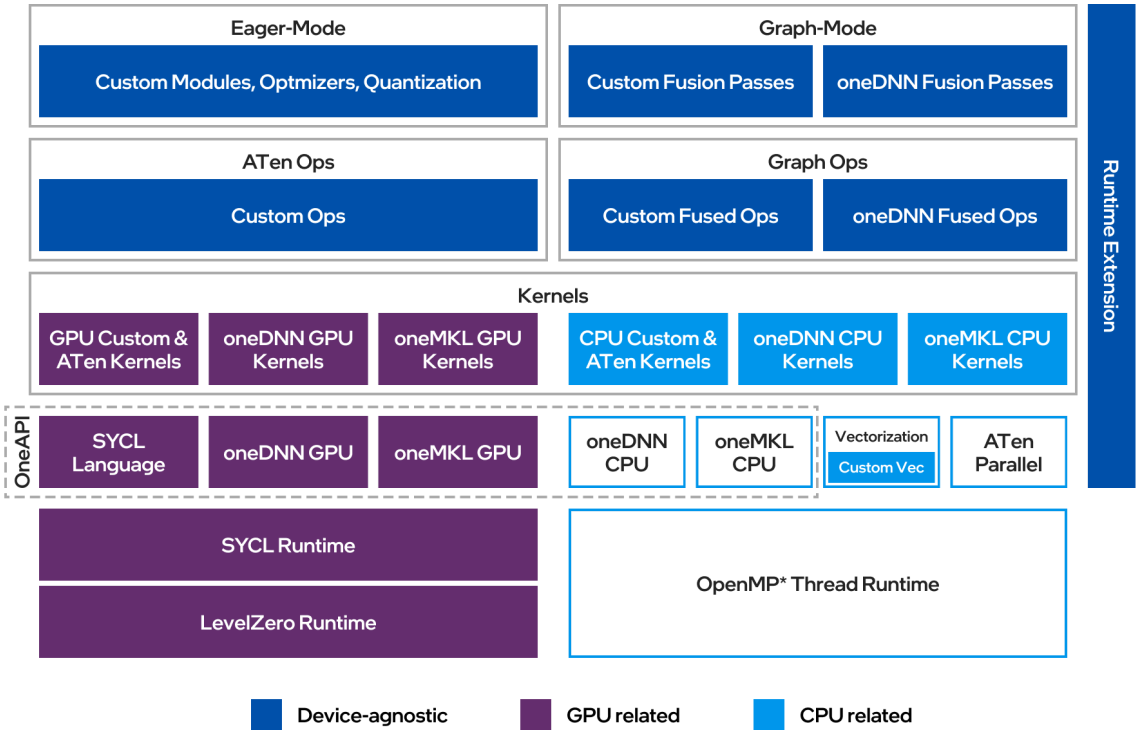
Intel® Extension for PyTorch（IPEX）是一个专门为 Intel 硬件优化的 PyTorch 扩展库，其框架如下图所示。它可以通过优化计算图、使用更高效的算子等方式来加速 PyTorch 的训练和推理过程。在训练和推理过程中，由于我们所使用的硬件为 Intel CPU 及 Intel GPU，因此使用 IPEX 可以显著提高计算性能。由于 Intel 官方提供的预编译版本的 IPEX 对于我们所使用的硬件型号不完全兼容，因此我们需要从源码编译 IPEX。

警告

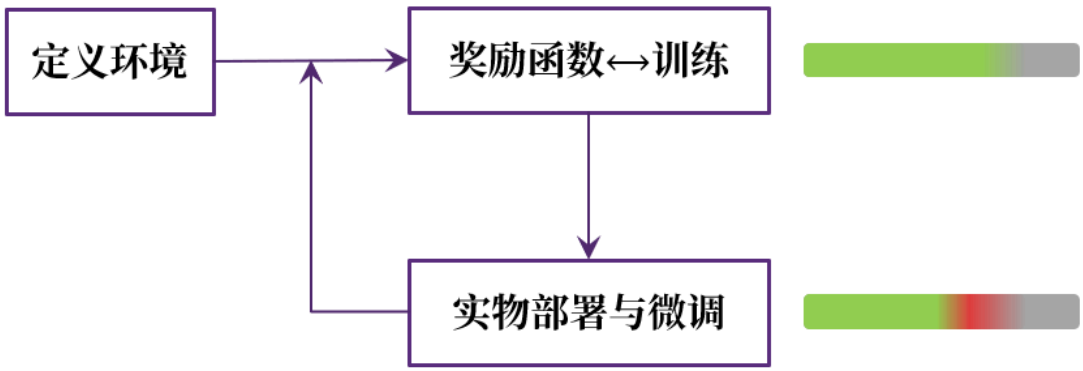
编译过程相当复杂，如有可能应当尽量使用预编译版本。我们尝试了一周左右，才成功跑通了编译过程。详细的记录可以参考此博客，这里不再赘述。

从结论而言，使用 Intel 优化版本的 Python，Numpy 及 Pytorch， 配合 IPEX 进行训练和推理，我们可以将代码的执行速度再提高约三倍。当然，如果条件允许，我们仍然推荐使用 CUDA 进行训练。

5. 总结



总得来说，使用强化学习来控制旋转倒立二阶摆的流程图如下所示。



工作流程

图中，训练部分的进度条表明，我们仍然有部分未实现的工作。这是因为在设计奖励函数时，我们发现仅使用单一模型而期冀实现旋转倒立二阶摆的起摆任务是非常困难的（尝试了大约一个月，然而还没有成功）—模型会倾向于落入一个局部最优解，即前文仿真结果页面中给出的 *BANNA* 状态。因此，我们不得不使用多模型的方式来实现起摆任务。具体而言，我们使用了一个模型来进行起摆训练，另一个模型来进行稳摆训练，然后再添加一个策略网络来实现起摆与稳摆之间的切换。但需要说明的是，这个方案并不符合强化学习的基本思想，其泛化性较差。最终目标应当是通过单一奖励函数的设计，便可以容易地得到适用于起摆任务的模型。针对此目标，后续要做的工作主要包括以下两点：

- 基于 SAC 和 TD3 算法，考虑一种新的强化学习算法来实现起摆任务。目前的实践说明，已有的常见算法在旋转倒立二阶摆任务中表现均欠佳，而且训练速度较慢。从文献调研结果来看，研究者在此复杂控制任务中也关注较少（特指旋转倒立二阶摆，几乎没有论文使用强化学习方法）。因此，新的算法设计是必须要考虑的。
- 得到一种更加通用的奖励函数设计策略。现在的奖励函数设计显著依赖于经验与调试，这也大幅影响了算法的泛化性。我们目前已经得到了一种针对此任务的基础实现方案，在此基础上就可以尝试更多自动化设计与调整

奖励函数的方法。这将有利于提高方法的通用性，使得我们在旋转倒立二阶摆任务中使用的工作流可以在其他控制任务中复用。

其他参考资料

- https://hrl.boyuai.com/
- Baek, Jongchan, et al. "Reinforcement learning to achieve real-time control of triple inverted pendulum." Engineering Applications of Artificial Intelligence 128 (2024): 107518.
- Bhourji, Rajmeet Singh, Saeed Mozaffari, and Shahpour Alirezaee. "Reinforcement learning ddpg–ppo agent-based control system for rotary inverted pendulum." Arabian Journal for Science and Engineering 49.2 (2024): 1683-1696.
- https://intel.github.io/intel-extension-for-pytorch/xpu/latest/tutorials/introduction.html
- https://onnxruntime.ai/
- https://learn.microsoft.com/en-us/windows/ai/new-windows-ml/overview