READYS: A Reinforcement Learning Based Strategy for Heterogeneous Dynamic Scheduling

摘要

在本文中，我实现了 READYS，这是一种强化学习算法，用于动态调度建模为有向无环图（DAG）的计算。在该算法中，分配和调度决策是在运行时根据系统状态做出的，其一般原则是逐步构建一个策略，该策略在给定系统状态的情况下，做出优化全局标准的决策。我的工作通过将图卷积网络（GCN）与 Actor-Critic 算法（A2C）相结合，实现了 READYS：它动态构建调度问题的自适应表示，并学习调度策略，旨在最小化 makespan，而且它构建了一个通用的调度策略，该策略既不局限于一个特定的应用程序或任务图，也不限于一个特定的问题大小，并且可用于调度任何 DAG。我的实验涵盖了源自线性代数分解核（CHOLESKY、LU、QR）的不同类型的任务图，并分析在给定组合上执行学习时 READYS 的性能。通过仿真，我的结果与论文中相似，能够证明调度代理获得的性能与基准算法非常相似甚至更优，尤其在存在不确定性的调度环境中表现强大。此外，代理展现了良好的泛化能力，可应用于不同规模的任务图。

关键词：动态调度；强化学习；图卷积网络

1 引言

1.1 选题背景

在现代计算机体系结构中，任务之间的调度对系统性能至关重要。而传统的静态调度算法在任务执行前就确定了调度计划，但对于具有不确定执行时间和通信成本的动态环境来说，这种方式可能不再适用。因此，引入了强化学习作为动态调度问题的一种解决方案。强化学习的优势在于它可以根据系统的实际状态和环境的动态变化，动态地学习并优化调度策略。

1.2 选题意义

该选题的意义在于提出一种名为 READYS 的强化学习算法，用于动态调度计算任务，其模型结合了图卷积网络（GCN）和演员-评论家算法（A2C）。READYS 旨在构建一个通用的调度策略，不受特定应用程序或任务图以及问题规模的限制，从而在不同的计算环境中实现性能优化。这一方法有望应用于各种异构计算系统，提高计算资源的利用率，降低计算任务的完成时间，对于提高计算效率和资源利用率具有重要的实际意义。

2 相关工作

2.1 静态调度算法——HEFT（Heterogeneous Earliest Finish Time）

HEFT 是一种经典的静态调度算法，用于优化任务图在异构计算环境中的执行 [5]。该算法通过估计每个任务的最早完成时间，然后根据这些估计值进行调度，以最小化整体任务完成时间（makespan）。HEFT 算法考虑了任务的计算和通信开销，以及任务在不同类型计算单元上的执行速度，从而提供了一个在异构资源上高效执行任务的静态调度策略。

2.2 GCN（Graph Convolutional Networks）

GCN 是一类用于图结构数据的深度学习模型 [2]。它的主要目标是捕捉图数据中节点之间的关系，以便在图上执行各种任务，如节点分类、图分类和链接预测。在任务图动态调度中，GCN 的应用通常结合强化学习算法，例如 Actor-Critic 算法，以制定智能的任务调度策略。通过将 GCN 与强化学习相结合，系统可以根据当前任务图和资源状态智能地调整调度决策，以最小化整体的任务完成时间。

2.3 A2C（Actor-Critic Algorithm）

A2C 是一种强化学习方法，它结合了策略梯度和值函数方法 [3]。A2C 通过同时训练一个策略网络（Actor）和一个值函数网络（Critic）来优化智能体的决策策略。Actor 网络学习在给定状态下输出行动概率分布，而 Critic 网络评估每个状态下采取行动的优劣，通过值函数指导 Actor 网络的训练。这种算法通过 Advantage 的概念，即某个状态下采取某个行动的相对优势，实现对策略和值函数的联合优化，以提高决策效率。A2C 通常支持并行训练，使其在处理大规模环境和任务时具备高效性。

3 本文方法

3.1 本文方法概述

本文旨在实现 READYS 算法，该算法是一种基于强化学习的动态任务调度方法，适用于将计算任务建模为有向无环图（DAG）的场景。主要包括以下内容：

- **强化学习环境构建：**创建任务调度环境，配置环境参数，包括节点数、节点类型、时间窗口等，并根据不同的环境类型（CHOLESKY、LU、QR）生成相应的任务数据和相关信息。
- **神经网络模型构建：**构建神经网络模型，该模型包括图卷积网络（GCN）处理任务图的部分和其他感知机层，输入为任务图的信息，如节点数、边信息、节点类型等。输出为任务调度的概率分布，表示模型对每个节点选择的动作概率。
- **强化学习代理训练：**通过代理进行强化学习训练，在每个训练步骤中，模型从环境中获取状态信息，通过 GCN 处理得到任务图表示，生成任务调度的概率分布，代理根据分布选择任务分配给计算资源。通过 Actor-Critic（A2C）算法计算奖励，优化模型参数，训练过程迭代进行直至达到停止条件。

3.2 强化学习环境构建

构建强化学习环境涉及多个关键要素，包括状态空间定义、动作空间设定、奖励函数设计以及环境动力学建模。

首先，在状态空间的定义方面，环境包含可见任务图部分、节点数量以及就绪任务列表等。这种状态表示允许代理观察当前任务图的局部结构和调度信息，为其提供关键的环境状态信息。

其次，动作空间包括可执行的一系列调度操作，例如选择将任务分配给特定处理器或选择不执行任务，代理需要在其中进行决策。

奖励函数基于调度的性能进行计算。奖励的计算涉及当前调度完成时间与参考任务路径的比较，这里指 HEFT 算法。具体而言，奖励计算公式如下：

$$\text{Reward} = \frac{\text{makespan(HEFT)} - \text{makespan(READYS)}}{\text{makespan(HEFT)}} \tag{1}$$

这种设计通过比较当前完成时间与参考任务路径完成时间的差异，为代理提供了关于其决策质量的明确反馈。

最后是环境动力学的建模，通过接收代理的动作作为输入，更新环境的状态，计算奖励，并判断是否达到终止条件。其实现使得代理能够与环境进行交互，并在学习过程中不断调整其策略以获得更好的奖励。

3.3 神经网络模型构建

在神经网络模型构建阶段，使用一个用于异构图学习的模型，在后续训练中充当策略网络和价值网络。该模型包含了图卷积层和感知机层，并具有可调参数，如图卷积层数量、感知机层数量等。该模型接收任务图的表示（包含图数据、节点数和就绪状态的字典）作为输入，经过图卷积层和感知机层的多层处理，输出路径选择的概率和特征表示。在这一过程中，考虑了任务的就绪状态和集群特征，以更好地指导路径选择。

此外，该模型支持批归一化和残差连接的可选使用。批归一化有助于加速训练过程，提高模型的泛化性能，而残差连接则有助于解决深层网络训练中的梯度消失问题，进一步增强模型的学习能力。

3.4 强化学习代理训练

每个训练步骤中，代理通过对环境进行采样，根据当前策略从环境中获取观测，并选择相应的动作。神经网络模型负责输出策略和值函数，其中策略用于在每个训练步骤中选择动作，而值函数用于评估状态的价值。基于采样得到的环境反馈，计算累积奖励和优势。

其中，奖励表示代理执行动作后在环境中获得的即时反馈，而优势则是当前状态估值与基准值的差异，用于衡量动作的优劣。通过计算获得的奖励和优势，进行神经网络模型的优化。这个优化过程持续进行，直到达到指定的训练步数为止

模型优化过程包括计算值函数均方误差和策略损失。值函数均方误差用于评估值函数的预测误差，策略损失则用于更新策略网络参数。通过反向传播和梯度裁剪，更新模型参数。

4 复现细节

4.1 代码参考说明

1. **Convolutional Neural Network:** 在神经网络模型构建阶段，我充分借鉴了经典卷积神经网络结构，如 LeNet、ResNet 等，以获取对图像处理和特征提取的有效方法。然而，为了更好地适应本工作的任务和数据特性，我对这些经典结构进行了深入调整和改进。

   首先，我考虑到异构图学习的独特性质，特别是任务图的动态变化。因此，如图 1所示，在 ModelHeterogene 模块中，我对图卷积层和感知机层的组合进行了改进，调整了图卷积层数量和感知机层数量等关键参数，以提高模型的表达能力和学习能力。

```
self.ngcn = ngcn   # 图卷积层的数量
self.nmlp = nmlp   # MLP 层的数量
self.withbn = withbn   # 批归一化
self.res = res   # 残差连接
self.listgcn = nn.ModuleList()
self.listmlp = nn.ModuleList()
self.listmlp_pass = nn.ModuleList()
self.listmlp_value = nn.ModuleList()
self.listgcn.append(BaseConvHeterogene(input_dim, hidden_dim, 'gcn', withbn=withbn))
for _ in range(ngcn-1):
    self.listgcn.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'gcn', res=res, withbn=withbn))
for _ in range(nmlp-1):
    self.listmlp.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'mlp', res=res, withbn=withbn))
self.listmlp.append(Linear(hidden_dim, 1))
for _ in range(nmlp_value-1):
    self.listmlp_value.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'mlp', res=res, withbn=withbn))
self.listmlp_value.append(Linear(hidden_dim, 1))

self.listmlp_pass.append(BaseConvHeterogene(hidden_dim+3, hidden_dim, 'mlp', withbn=withbn))
for _ in range(nmlp-2):
    self.listmlp_pass.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'mlp', res=res, withbn=withbn))
self.listmlp_pass.append(Linear(hidden_dim, 1))
```

图 1. GCN 调整实现

其次，如图 2所示，针对异构性的特殊需求，我在 BaseConvHeterogene 模块中引入了批归一化和残差连接。批归一化有助于加速模型的收敛过程并提高模型的稳定性，特别是在异构图学习的场景下。残差连接则通过跨层的信息传递，有效缓解了梯度消失问题，进一步增强了模型的学习能力。

```
# 是否启用批归一化
if self.withbn:
    x = self.bn(x)
# 是否启用残差连接
if self.res:
    return F.relu(x) + input_x
return F.relu(x)
```

图 2. 批归一化和残差连接实现

经典卷积神经网络的样例实现可在 PyTorch 官网中找到 [1]。

2. **HEFT 算法：**在任务调度环境构建模块中，我设计了奖励函数，以更好地引导强化学习代理学习有效的任务调度策略。如图 3所示，奖励函数的设计中，我选择将 HEFT（Heterogeneous Earliest Finish Time）算法作为参考任务路径的基准。

```
reward = (self.heft_time - self.time) / self.heft_time if done else 0
```

图 3. 奖励函数实现

HEFT 算法是一种静态调度启发式算法，旨在在异构计算环境中高效执行任务调度。我将其用作评估任务调度性能的标准，具体而言，通过计算当前调度完成时间与 HEFT 算法参考任务路径完成时间的差异来构建奖励信号。这一差异性奖励的设计能够直观地反映出代理决策的优劣，使代理更倾向于学习能够在整个任务图上更接近 HEFT 算法性能的调度策略。

HEFT 算法的实现可在 github 上找到 [4]。

4.2 复现内容说明

在本次复现工作中，使用了 Python 作为编程语言，并依赖于 PyTorch 框架完成了整个强化学习部分的实现任务，包括环境构建、模型构建、代理训练以及任务图构建等工具类。

4.2.1 环境构建

在这一部分中，我实现了对任务调度环境的建模以及与代理交互的核心逻辑。

首先，通过创建环境类 DAGEnv，对任务调度问题进行了建模。在初始化环境时，可以选择不同类型的任务图，这些任务图是通过专用的工具类生成的，将任务图的文本数据转换为可操作的数据结构。解析函数负责提取任务类型和任务之间的边，从而构建了任务图的结构。生成的任务图被封装为 TaskGraph 类，包括节点的特征、图的边连接关系以及每个节点的任务信息。此外，对任务图的节点添加了附加特征，这些特征表示节点的后继任务的汇总信息，包括后继任务的数量和任务类型等。

```
self.observation_space = Dict
self.action_space = "Graph"

self.noise = noise
self.time = 0
self.num_steps = 0
self.p = p
self.n = n
self.window = window
self.env_type = env_type
if self.env_type == 'LU':
    self.max_duration_cpu = max(durations_cpu_lu)
    self.max_duration_gpu = max(durations_gpu_lu)
    self.task_data = ggen_denselu(self.n, self.noise)
elif self.env_type == 'QR':
    self.max_duration_cpu = max(durations_cpu_qr)
    self.max_duration_gpu = max(durations_gpu_qr)
    self.task_data = ggen_QR(self.n, self.noise)
elif self.env_type == 'chol':
    self.max_duration_cpu = max(durations_cpu)
    self.max_duration_gpu = max(durations_gpu)
    self.task_data = ggen_cholesky(self.n, self.noise)
else:
    raise EnvironmentError('not implemented')
```

图 4. DAGEnv 初始化部分实现

在环境初始化过程中，还计算了 HEFT 调度算法的时间，用于后续奖励函数 reward 的计算。

```
# 计算 heft 时间
string_cluster = string.printable[:self.p]
dic_heft = {}
for edge in np.array(self.task_data.edge_index.t()):
    dic_heft[edge[0]] = dic_heft.get(edge[0], ()) + (edge[1],)

def compcost(job, agent):
    idx = string_cluster.find(agent)
    expected_duration = self.task_data.task_list[job].durations[self.cluster.node_types[idx]] + self.noise

    return expected_duration

def commcost(ni, nj, A, B):
    return 0

orders, jobson = heft.schedule(dic_heft, string_cluster, compcost, commcost)
try:
    self.heft_time = orders[jobson[self.num_nodes - 1]][-1].end
except:
    self.heft_time = max([v[-1] for v in orders.values() if len(v) > 0])
```

图 5. HEFT 调度算法时间计算实现

接着，我实现了 _compute_state 方法，用于计算环境的状态表示。在这个过程中，调用了 _compute_embeddings 方法，生成了用于神经网络模型输入的嵌入表示，并构造了包含图信息、节点数量和就绪列表等关键信息的状态表示。

```python
def _compute_state(self):
    """
        计算当前环境状态的表示。
    """
    visible_graph, node_num = compute_sub_graph(self.task_data,
                                                torch.tensor(np.concatenate((self.running[self.running > -1],
                                                                             self.ready_tasks)), dtype=torch.long),
                                                self.window)
    visible_graph.x, ready = self._compute_embeddings(node_num)
    return {'graph': visible_graph, 'node_num': node_num, 'ready': ready}
```

图 6. 环境状态表示实现

在 step 方法中，我实现了任务调度的逻辑。根据代理选择的动作进行相应的任务调度，奖励的计算基于代理执行动作后获得的即时反馈，以及当前状态的估值与基准值的差异，从而衡量动作的优劣。

```python
def step(self, action, render_before=False, render_after=False, enforce=True, speed=False):
    self.num_steps += 1
    self._find_available_proc()
    if action == -1 and enforce:
        if len(self.running_task2proc) == 0:
            action = self.ready_tasks[0]
    if action != -1:
        self.compeur_task += 1
    self._choose_task_processor(action, self.current_proc)
    if render_before:
        self.render()
    done = self._go_to_next_action(action, enforce)
    if render_after and not speed:
        self.render()
    reward = (self.heft_time - self.time) / self.heft_time if done else 0
    if self.time != 0:
        improvement = self.heft_time / self.time
    else:
        improvement = 0
    if done:
        print("1. heft_time: ", self.heft_time, ', readys_time: ', self.time)
        print('2. reward: ', reward)
        print('3. improve (heft_time / readys_time): ', improvement)
    info = {'episode': {'r': reward, 'length': self.num_steps, 'time': self.time}, 'bad_transition': False}
    if speed:
        return 0, reward, done, info, improvement
    return self._compute_state(), reward, done, info, improvement
```

图 7. 任务调度逻辑实现

4.2.2 神经网络模型构建

在这一部分中，我采用了混合模型的设计，结合图卷积网络（GCN）和多层感知机（MLP），以解决任务调度问题。在模型的初始化中，通过指定输入维度、隐藏维度以及网络层数等参数，创建了一个灵活的结构。这个模型被称为 ModelHeterogene，它继承自 PyTorch 的 torch.nn.Module 类。

```python
class ModelHeterogene(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim=128, ngcn=2, nmlp=1, nmlp_value=1, res=False, withbn=False):
        super(ModelHeterogene, self).__init__()
        self.ngcn = ngcn   # 图卷积层的数量
        self.nmlp = nmlp   # MLP 层的数量
        self.withbn = withbn   # 批归一化
        self.res = res   # 残差连接
        self.listgcn = nn.ModuleList()
        self.listmlp = nn.ModuleList()
        self.listmlp_pass = nn.ModuleList()
        self.listmlp_value = nn.ModuleList()
        self.listgcn.append(BaseConvHeterogene(input_dim, hidden_dim, 'gcn', withbn=withbn))
        for _ in range(ngcn-1):
            self.listgcn.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'gcn', res=res, withbn=withbn))
        for _ in range(nmlp-1):
            self.listmlp.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'mlp', res=res, withbn=withbn))
        self.listmlp.append(Linear(hidden_dim, 1))
        for _ in range(nmlp_value-1):
            self.listmlp_value.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'mlp', res=res, withbn=withbn))
        self.listmlp_value.append(Linear(hidden_dim, 1))

        self.listmlp_pass.append(BaseConvHeterogene(hidden_dim+3, hidden_dim, 'mlp', withbn=withbn))
        for _ in range(nmlp-2):
            self.listmlp_pass.append(BaseConvHeterogene(hidden_dim, hidden_dim, 'mlp', res=res, withbn=withbn))
        self.listmlp_pass.append(Linear(hidden_dim, 1))
```

图 8. ModelHeterogene 模块实现

首先，模型初始化时会根据设定的图卷积层数（ngcn）构建图卷积层（BaseConvHeterogene）的列表。每个图卷积层在初始化时根据类型参数选择是使用图卷积（GCNConv）还是线性层（Linear），并且可以选择是否启用批归一化（withbn）和残差连接（res）。

接下来，根据设定的 MLP 层数（nmlp 和 nmlp_value），构建了用于任务调度和节点价值估计的两组 MLP 层。这些 MLP 层通过 BaseConvHeterogene 模块实现，同样支持批归一化和残差连接的选项。

在前向传播过程中，输入的图数据经过图卷积层的处理，然后通过求取平均值和最大值，生成两个不同的节点表示。这些表示分别送入任务调度 MLP 和节点价值估计 MLP，生成最终的任务调度概率分布和节点价值的估计。

最后，任务调度概率由任务调度 MLP 的输出和特殊节点特征拼接而成，经过 Softmax 函数得到最终的任务调度概率。

```python
class BaseConvHeterogene(torch.nn.Module):
    def __init__(self, input_dim, output_dim, type='gcn', res=False, withbn=False):
        super(BaseConvHeterogene, self).__init__()
        self.res =res
        self.net_type = type
        # 根据网络类型选择是使用 GCN 还是线性层
        if type == 'gcn':
            self.layer = GCNConv(input_dim, output_dim, flow='target_to_source')
        else:
            self.layer = Linear(input_dim, output_dim)
        self.withbn = withbn
        if withbn:
            self.bn = torch.nn.BatchNorm1d(output_dim)

    def forward(self, input_x, input_e=None):
        if self.net_type == 'gcn':
            x = self.layer(input_x, input_e)
        else:
            x = self.layer(input_x)
        # 是否启用批归一化
        if self.withbn:
            x = self.bn(x)
        # 是否启用残差连接
        if self.res:
            return F.relu(x) + input_x
        return F.relu(x)
```

图 9. BaseConvHeterogene 模块实现

4.2.3 代理训练

在这一部分中，我使用 A2C 算法实现了代理训练的过程。

首先，代理通过与环境进行交互，执行策略并收集环境的观测。神经网络模型使用的是上一部分中实现的 ModelHeterogene，负责输出当前状态的策略和值函数估计。在代理的训练步骤中，这一过程通过神经网络的前向传播来计算当前状态的策略（policy）和值函数（value）。

```python
for i in range(batch_size):
    # Collect observations from the environment
    observations.append(observation['graph'])
    policy, value = self.network(observation)
    values[i] = value.detach().cpu().numpy()
    vals[i] = value
    probs_entropy[i] = - (policy * policy.log()).sum(-1)
    try:
        action_raw = torch.multinomial(policy, 1).detach().cpu().numpy()
    except:
        print("chelou")
    probs[i] = policy[action_raw]
    ready_nodes = observation['ready'].squeeze(1).to(torch.bool)
    actions[i] = -1 if action_raw == policy.shape[-1] -1 else observation['node_num'][ready_nodes][action_raw]
    observation, rewards[i], dones[i], info, improvement = self.env.step(actions[i])  # 改

    observation['graph'] = observation['graph'].to(device)
    n_step += 1

    if dones[i]:
        observation = self.env.reset()
        observation['graph'] = observation['graph'].to(device)
        reward_log.append(rewards[i])
        time_log.append(info['episode']['time'])
        max_improvement = max(max_improvement, improvement)
```

图 10. 交互步骤部分实现

接下来，根据代理与环境的交互计算累积奖励（returns）和优势（advantages）。累积奖励的计算采用时间差分的方式，通过回溯计算未来奖励的累积总和。优势的计算则是通过当前状态的估值与基准值的差异来衡量。

```python
def _returns_advantages(self, rewards, dones, values, next_value):
    returns = np.append(np.zeros_like(rewards), [next_value], axis=0)

    for t in reversed(range(rewards.shape[0])):
        returns[t] = rewards[t] + self.gamma * returns[t + 1] * (1 - dones[t])

    returns = returns[:-1]
    advantages = returns - values
    return returns, advantages
```

图 11. 累积奖励和优势计算实现

最后，代理采用 Actor-Critic 方法，通过优化值函数均方误差和策略损失来更新神经网络模型参数。其中，值函数均方误差通过比较估值和实际返回的累积奖励计算得到，而策略损失则通过最大化动作概率乘以优势的期望来实现。

```python
def optimize_model(self, observations, actions, probs, entropies, vals, returns, advantages, step=None):

    returns = torch.tensor(returns[:, None], dtype=torch.float, device=device)
    advantages = torch.tensor(advantages, dtype=torch.float, device=device)

    loss_value = 1 * F.mse_loss(vals.unsqueeze(-1), returns)
    if self.writer:
        self.writer.add_scalar('critic_loss', loss_value.data.item(), step)

    # Actor loss
    loss_policy = ((probs.log()) * advantages).mean()
    loss_entropy = entropies.mean()
    loss_actor = - loss_policy - self.entropy_cost * loss_entropy
    if self.writer:
        self.writer.add_scalar('actor_loss', loss_actor.data.item(), step)

    total_loss = self.config["loss_ratio"] * loss_value + loss_actor
    total_loss.backward()
    torch.nn.utils.clip_grad_norm_(self.network.parameters(), 10)
    self.optimizer.step()
    return loss_value.data.item(), loss_actor.data.item(), loss_entropy.data.item()
```

图 12. 模型优化实现

在这个过程中，通过定期评估当前策略的性能，记录关键指标，如 actor_loss、critic_loss、entropy、improvement、reward、test_time、time 等，并通过 TensorBoard 进行可视化。在训练过程中，还保存了在测试集上表现最佳的模型，以便后续使用。

```python
if self.writer is not None and log_ratio * self.config['log_interval'] < n_step:
    log_ratio += 1
    self.writer.add_scalar('reward', np.mean(reward_log), n_step)
    self.writer.add_scalar('time', np.mean(time_log), n_step)
    self.writer.add_scalar('critic_loss', loss_value, n_step)
    self.writer.add_scalar('actor_loss', loss_actor, n_step)
    self.writer.add_scalar('entropy', loss_entropy, n_step)
    self.writer.add_scalar('improvement', max_improvement, n_step)
    current_time = self.evaluate()
    self.writer.add_scalar('test time', current_time, n_step)
```

图 13. 指标记录实现

4.2.4 任务图构建

在这一部分中，我实现了构建任务图的工具类，其功能为根据生成的图数据进行解析，构建任务图的表示。

首先，使用 ggen 方法生成分解任务的数据流图，并将其存储在文件中。然后，通过解析生成的数据流图，提取节点和边的信息，得到任务列表 tasks 和边的信息 edges。根据节点信息构建节点特征矩阵 x，然后根据边的信息构建任务图 task_graph。每个节点特征表示任务类型，其中，p、s、t、g 分别对应于不同的任务类型。任务图的边通过 edge_index 表示。

```python
class TaskGraph(Data):

    def __init__(self, x, edge_index, task_list):
        Data.__init__(self, x, edge_index.to(torch.long))
        self.task_list = np.array(task_list)
        self.task_to_num = {v: k for (k, v) in enumerate(self.task_list)}
        self.n = len(self.x)

    def add_features_descendant(self):
        n = self.n
        x = self.x
        succ_features = torch.zeros((n, 4))
        succ_features_norm = torch.zeros((n, 4))
        edges = self.edge_index
        for i in reversed(range(n)):
            succ_i = edges[1][edges[0] == i]
            feat_i = x[i] + torch.sum(succ_features[succ_i], dim=0)
            n_pred_i = torch.FloatTensor([torch.sum(edges[1] == j) for j in succ_i])
            if len(n_pred_i) == 0:
                feat_i_norm = x[i]
            else:
                feat_i_norm = x[i] + torch.sum(succ_features_norm[succ_i] / n_pred_i.unsqueeze(1).repeat((1, 4)), dim=0)
            succ_features[i] = feat_i
            succ_features_norm[i] = feat_i_norm
        return succ_features_norm, succ_features
```

图 14. 任务图构建实现

```python
def ggen_cholesky(n_vertex, noise=0):
    dic_task_ch = {'p': 0, 's': 1, 't': 2, 'g': 3}
    def parcours_and_purge(s):
        reg = re.compile('\[kernel=[\D]*\]')
        x = reg.findall(s)
        return np.array([dic_task_ch[subx[8:9]] for subx in x])
    def parcours_and_purge_edges(s):
        reg = re.compile('\\t[\d]+ -> [\d]+\\t')
        x = reg.findall(s)
        out = np.array([[int(subx.split(' -> ')[0][1:]), int(subx.split(' -> ')[1][:-1])] for subx in x])
        return out.transpose()
    file_path = 'graphs/cholesky_{}.txt'.format(n_vertex)
    if os.path.exists(file_path):
        with open(file_path, 'r') as f:
            graph = f.read()
    else:
        stream = os.popen("ggen dataflow-graph cholesky {:d}".format(n_vertex))
        graph = stream.read()
    edges = parcours_and_purge_edges(graph)
    tasks = parcours_and_purge(graph)
    n = len(tasks)
    x = np.zeros((n, 4), dtype=int)
    x[np.arange(n), tasks.astype(int)] = 1
    task_list = []
    for i, t in enumerate(tasks):
        task_list.append(Task((t, i), noise=noise, task_type='chol'))
    return TaskGraph(x=torch.tensor(x, dtype=torch.float), edge_index=torch.tensor(edges), task_list=task_list)
```

图 15. 图数据解析实现，以 CHOLESKY 为例

4.3 实验环境搭建

4.3.1 实验环境

本工作使用的环境配置如下：

- 操作系统：Windows 11
- 开发语言：Python 3.11.4

- 框架: PyTorch 2.1.0
- CPU: 12th Gen Intel(R) Core(TM) i7-12700
- 内存: 32G

4.3.2 数据集

考虑了三种类型的有向无环图（DAG），分别对应 CHOLESKY，LU 和 QR 分解。这些分解涉及大量任务，复杂的依赖关系和少量不同的内核。本实验中，选择使用任务数较少的任务图进行训练，并在任务较多的任务图上进行测试。

4.3.3 基准算法

选择了 HEFT 作为参考的静态算法，该算法是一种静态的列表调度启发式，与 READYS 不同，它使用整个 DAG 来计算调度。

4.3.4 参数设置

执行对模型的超参数的随机搜索，包括窗口大小 $w \in [0, 2]$ 和 GCN 层数 $g \in [1, 3]$。使用 Adam 优化器进行网络训练，学习率为 0.01。对于 Actor-Critic 算法，选择了折扣因子 $\gamma = 0.99$，基线损失缩放为 0.5，并在 $[20, 40, 60, 80]$ 和 $[10^{-3}, 5 \times 10^{-3}, 10^{-2}]$ 中进行了展开长度和熵损失比例的网格搜索。这里选择 T = 4 和 8 的三种类型的任务图进行训练，其中，T 表示矩阵每个维度上的瓦片数。

4.3.5 迁移学习

探讨一个在特定 DAG 上训练的代理是否能够调度其他不同大小的 DAG。这里选择 CHOLESKY 任务图，并将在 T = 4 和 8 个瓦片上训练的 READYS 直接应用于大小为 10 和 12 个瓦片的 DAG。

4.4 监控与分析平台

本工作使用了 TensorBoard 记录了多个关键指标，以便于对模型训练进行监控和分析。下面是记录的指标及其含义：

- **actor_loss:** 演员网络的损失。
- **critic_loss:** 评论家网络的损失。
- **entropy:** 动作概率分布的熵，表示模型的不确定性。
- **improvement:** 模型相对于参考任务路径（HEFT 算法）的性能改进。
- **reward:** 每个训练步骤的奖励值。
- **test_time:** 每个训练步骤的调度时间。

- **time:** 训练过程中的时间信息。

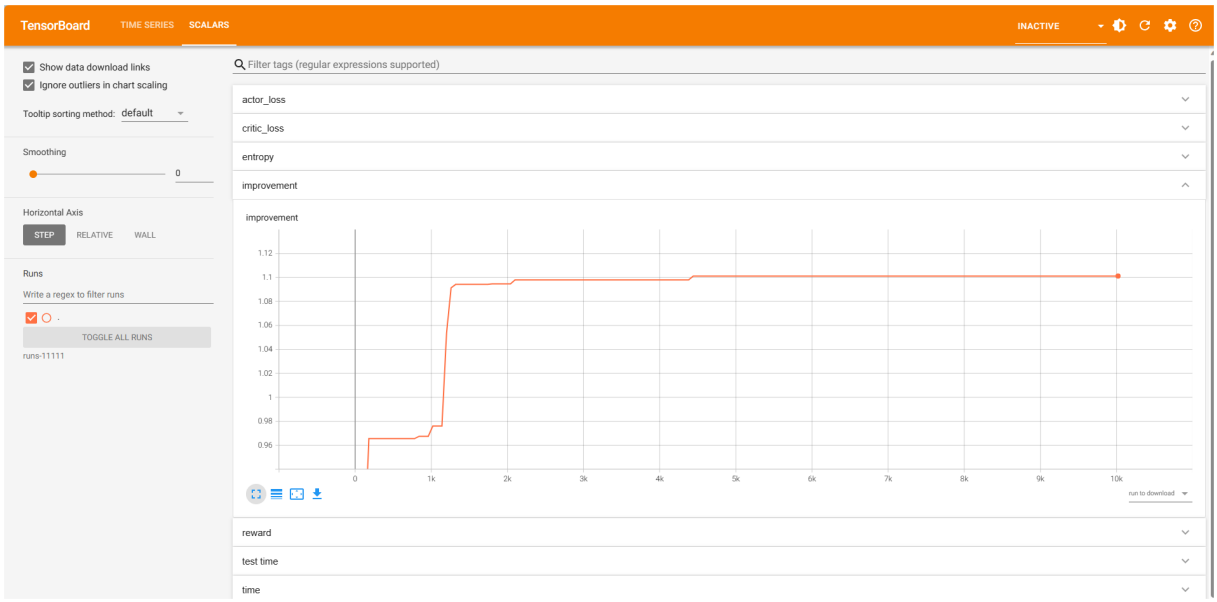


图 16. TensorBoard 界面演示图，以 improvement 指标为例

通过监控这些指标，可以更好地理解模型的训练过程，优化超参数，并评估模型的性能。同时，TensorBoard 提供了直观的可视化界面，有助于深入分析模型的学习过程。

5 实验结果分析

5.1 训练结果

选择 $T = 4$ 的三种类型的任务图进行训练，分别为 20、30 和 30 个任务。在 σ（噪声水平）大于 0 时，任务持续时间具有随机性。从图17-图19可以看出，对比静态启发式算法 HEFT，观察当 σ 较小时，READYS 的性能类似于 HEFT，但随着 σ 的增加，READYS 在自主发现任务持续时间的能力上超越了 HEFT。

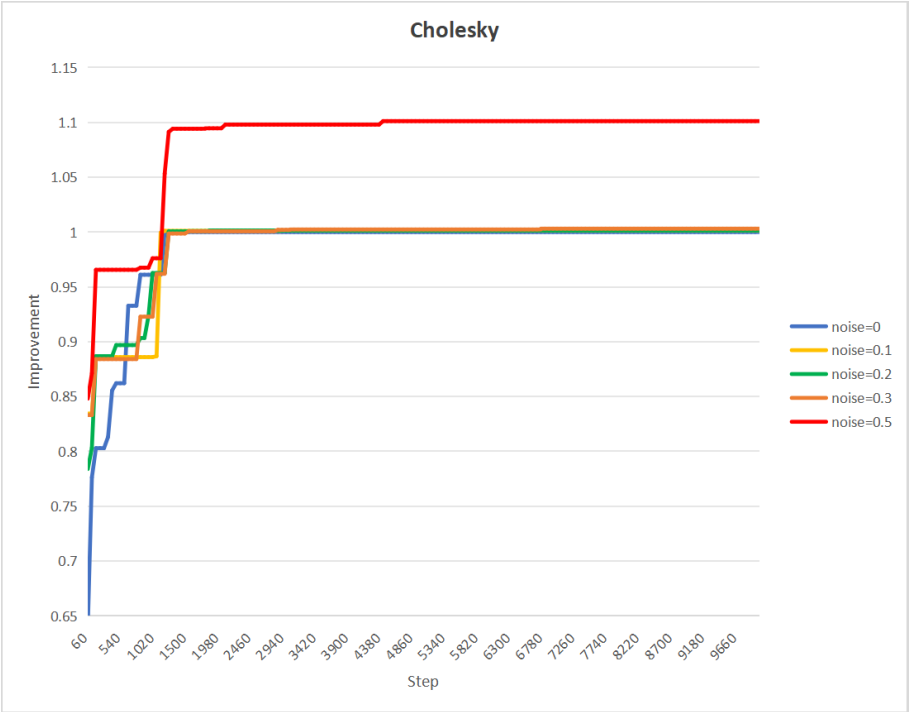


图 17. 在不同 σ 条件下，T=4 时 CHOLESKY 任务图的 READYS 改进率

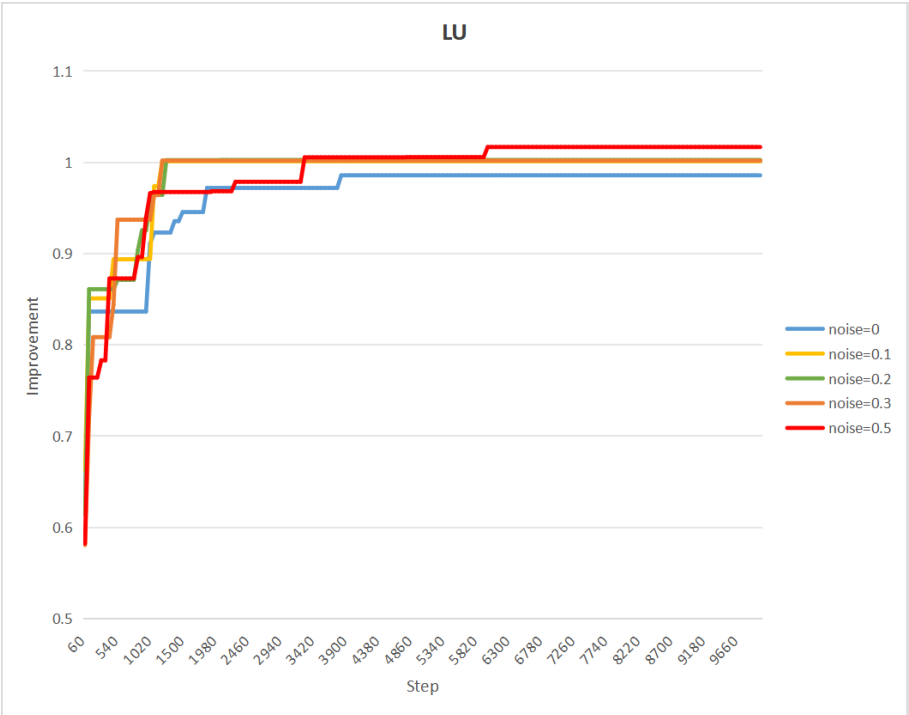


图 18. 在不同 σ 条件下，T=4 时 LU 任务图的 READYS 改进率

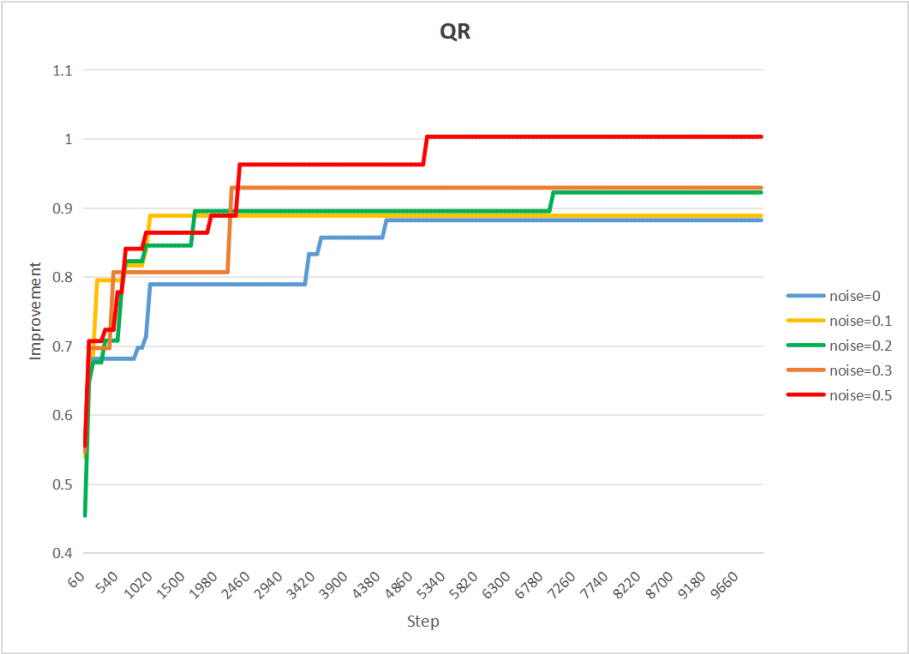


图 19. 在不同 σ 条件下，T=4 时 QR 任务图的 READYS 改进率

以 T = 8 的 CHOLESKY 任务图进行训练，测试较大任务图的 makespan。因为 READYS 对任务持续时间的不确定性不敏感，所以在安排任务时考虑了系统的实际状态和任务的意外持续时间。相反，HEFT 在执行之前计算调度，并且不太能够应对持续时间的变化。这也意味着，为了处理不确定性，一旦输入图足够大，做出动态决策比获得图的完整拓扑更好。从图20可以看出，不确定性较大，即噪声较大时，READYS 改进率较好。

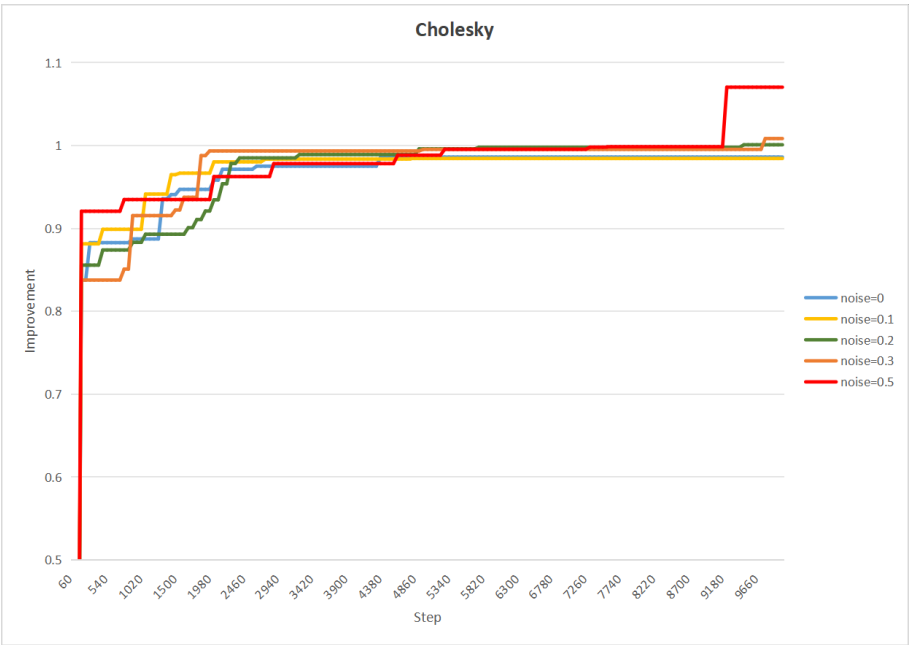


图 20. 在不同 σ 条件下，T=8 时 CHOLESKY 任务图的 READYS 改进率

考虑到原论文结果仅以图表形式呈现，无法提供具体数值，因此在此进行直观上的大致估计。与原论文结果相比，本次复现实验结果整体趋势相似，但存在略微差异。初步分析表

明，可能的原因包括所使用的 CPU 核心数不一致以及部分参数存在微小差异。尽管存在这些差异，本次复现仍然成功达到了预期目标，并呈现出优于基准算法的性能。

5.2 迁移学习结果

选择 CHOLESKY 任务图，并将在 T = 4 和 8 个瓦片上训练的 READYS 直接应用于大小为 10 和 12 个瓦片的 DAG，分别为 220 和 364 个任务。图22展示了迁移学习的效果，当用于调度大小为 10 或 12 的问题时，T = 4 训练的模型性能较弱，因为其训练使用的环境与测试环境差异太大，而 T = 8 的模型获得了较好的性能，尤其是在 $\sigma > 0.2$ 时变得有竞争力。

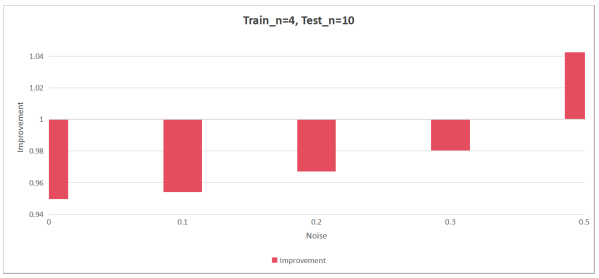
```
1. heft_time:  4563.865771949325 , readys_time:  4475.879263296427
2. reward:  0.019278943126172857
3. improve (heft_time / readys_time):  1.0196579271863775
num_nodes= 220
Final improvement: 1.0196579271863775

Process finished with exit code 0
```

图 21. 迁移学习代码运行结果

迁移学习实验遵循了原论文的实验步骤，对比结果仍采用直观估计。尽管与原论文相比，实验结果显示出相似的趋势，但提升率略有下降。这一差异可能源于硬件环境和资源配置的差异，以及超参数的微小调整。尽管存在这些差异，本次迁移学习实验仍然取得了显著的性能提升，为实际应用奠定了坚实基础。

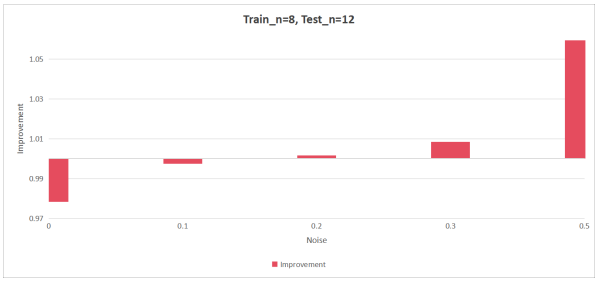
总体而言，尽管提升率略低，但迁移学习仍然能够在复现实验中取得相近的结果，表明其在任务调度领域的有效性。



(a) 训练 T=4，测试 T=10

(b) 训练 T=4，测试 T=12

(c) 训练 T=8，测试 T=10



(d) 训练 T=8，测试 T=12

图 22. 迁移学习中 READYS 改进率

6 总结与展望

本次工作复现了 READYS 算法，包括其与图卷积网络（GCN）和 Actor-Critic 算法（A2C）的结合，这使得 READYS 在学习和应用调度策略时更加灵活和高效。通过模拟实验，验证了 READYS 算法在不同任务图场景下的性能。

未来的研究可以深入探索 READYS 算法，着重在模型结构、超参数调整和算法融合策略等方面进行更深入的研究。其中，引入先进的图神经网络（GNN）结构，如 GAT（Graph Attention Networks）或 GraphSAGE（Graph Sample and Aggregation），有望更好地捕捉任务图的复杂关系和拓扑结构，从而提高 READYS 对任务图动态特性的建模能力，进一步提高调度决策的效率。

此外，采用自动调参技术，例如超参数优化算法或神经网络架构搜索（NAS），对 READYS 的超参数进行全面、智能的搜索和调整，有助于发现更优的配置，从而优化算法的性能，并增强其在不同场景下的通用性。

另一方面，可以考虑融合强化学习算法的混合策略，如结合深度 Q 网络（DQN）的经验回放机制。这种融合策略有望增强 READYS 算法对历史经验的学习和利用，提高算法对长期依赖关系的处理能力，从而在复杂任务图上表现更为出色。

参考文献

[1] Defining a neural network in pytorch. https://github.com/pytorch/tutorials/tree/main/recipes_source/recipes.

[2] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. JMLR.org, 2016.

[4] Matthew Rocklin. Heterogeneous earliest finish time, 2013. https://github.com/mrocklin/heft.

[5] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.