Decision Transformer: Reinforcement Learning via Sequence Modeling

摘要

如今，Transformer 模型已经在各种领域都大放异彩，例如图像分类、自然语言处理等等，在原版的基础上也已经衍生出了很多优秀的改版模型。但是在强化学习领域，依然没有很好的能将 Transformer 与强化学习任务所结合的模型，本篇文章即是填补了这一块工作上的空白。作者引入了一个框架，其将强化学习抽象为一个序列建模问题，这使我们能够利用 Transformer 架构的简洁性和可扩展性。并且与传统强化学习的通过拟合值函数或计算策略梯度的方法不同，Decision Transformer 通过利用一个 Transformer 模型输出最优动作，简单直接。虽然其简单，但作者在 OpenAI Gym 环境任务里的测试效果超过了原先最好的无模型离线 RL 基线的性能

关键词：强化学习；Transformer；OpenAI Gym

1 引言

随着 Transformer 模型 [1] 在各大领域成功的运用，我们越来越意识到其在大规模上建模语义概念的高维分布的作用，所以作者试图研究将 Transformer 应用到强化学习领域的顺序决策问题的可能性。与传统的将 Transformer 作为 RL 算法的组件的架构不同，作者采用生成轨迹建模——即建模状态 (state)、动作 (action) 和奖励 (reward) 序列的联合分布，尝试这种方法是否能替代传统的 RL 算法。

在传统的 RL 算法中 (例如 TD 算法)，通常是通过训练值函数和计算策略梯度而达到动作预测的目的，而在这篇文章中，作者使用了顺序序列训练 Transformer 模型，这将允许我们绕过长期信用分配的需求，从而避免 RL 中的 “deadly triad”，它还避免了像 TD 学习中的需要对未来奖励进行折现对行为，这可能会导致模型的短视现象。此外，我们还可以利用已经在语言和视觉领域中广泛使用并且易于扩展的现有 Transformer 框架，利用大量已经稳定训练的 Transformer 模型进行工作。

作者考虑通过离线强化学习 (offline RL) 来验证和探索这个假设，在这里我们将让 agent 学习来自次优算法在环境中跑出的序列集，从而测试模型在固定的有限经验中产生最大效能行为的能力。由于错误传播和价值过度估计的问题 [2]，这个任务在传统 RL 算法中是具有挑战性的。但在 Decision Transformer 这样的以序列建模目标进行训练时，它是个很自然的任务，所以作者认为这个模型会很擅长于处理离线强化学习的任务，并且拥有优秀的性能。

在接下来的部分我会详细阐述这个模型的细节、我做的改进工作以及关于这个模型的优缺点等等。

2 相关工作

在原文中，作者一共在两个环境中测试了模型，分别是 OpenAI Gym 以及 Atari [3] 环境。此次复现任务中只复现了第一个 OpenAI Gym 环境，故以下会对 OpenAI Gym 环境做一个简短介绍。由于目前 Decision Transformer 只能运用于离线强化学习 (Offline RL)，故以下也会对强化学习以及离线强化学习做出介绍。

2.1 OpenAI Gym

在强化学习中，通常需要让算法在一个环境中不断的与环境交互，通过试错来学习。但是有一种情况，我们事先已经有了一个固定的数据集，想要从这个数据集中学到一个好的策略，这就是离线强化学习。这个方法在处理以前收集的大量数据时很有用，就像在监督学习中利用大型数据集一样。OpenAI Gym 中的 d4rl [4] 基准则是一种新的方法来评估离线强化学习算法的性能。为了更好地理解这个方法，研究人员创建了一些特定设计用于离线学习的测试任务以及相应的数据集, 如图 1所示。这些任务和数据集包括了一些实际应用中常见的情况，例如由专业控制器和人类示范者生成的数据、代理在同一环境中执行不同任务的多任务数据等。

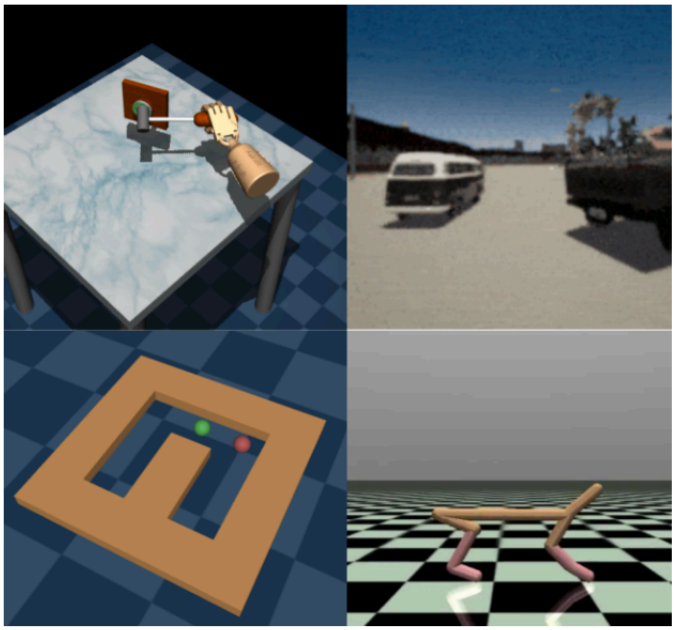

图 1. d4rl 游戏环境

通过引入这些新的测试任务和数据集，研究人员发现了一些现有算法的问题。他们还提供了这些任务和数据集的详细评估结果，以及一些开源示例，希望能够为整个研究社区提供一个共同的起点，促进离线强化学习领域的进一步发展。

在本篇文章中作者一共测试了该环境下的 12 个游戏以及轨迹集，由于运行时间和机能限制，本次复现工作只复现了其中三个游戏轨迹集，分别是 Hopper-medium、Hopper-medium-replay 以及 Walker2d-medium-replay。

2.2 强化学习

强化学习（Reinforcement Learning，简称 RL）是机器学习的一个分支，关注如何通过智能体（Agent）与环境（Environment）的交互来实现目标。在强化学习中，智能体通过观察环境的状态（State）并采取行动（Action），然后根据环境的反馈获得奖励（Reward）。智能体的目标是学习一个策略，使得在不同的环境状态下选择最优的行动，以最大化长期累积的奖励。

强化学习的基本要素包括：

- 智能体（Agent）：算法或机器学习模型，负责在环境中执行动作。
- 环境（Environment）：智能体所处的外部系统，它会对智能体的动作做出响应，并提供反馈 (奖励或惩罚)。
- 状态（State）：描述环境的特定瞬时情况，对智能体来说是可观察的信息。
- 动作（Action）：智能体可以在每个时间步骤中采取的行动。
- 奖励（Reward）：在执行动作后，环境提供的一个信号，用于评估该动作的好坏。
- 策略（Policy）：从状态到动作的映射规则，定义了智能体在特定状态下应该采取的动作。

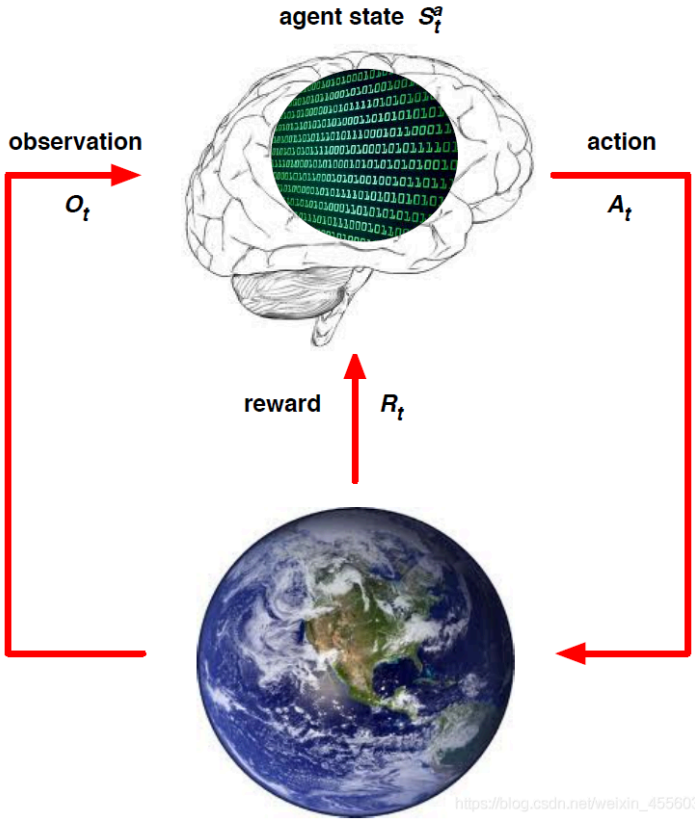

图 2. 强化学习

强化学习在许多领域取得了显著的成就，包括机器人控制、游戏玩法、金融交易等。著名的强化学习应用包括 AlphaGo 在围棋中的表现以及深度 Q 网络（Deep Q-Network，DQN）在玩电子游戏中取得的成功。本文则聚焦于接下来要介绍的强化学习的分支之一——离线强化学习

2.3 离线强化学习

离线强化学习（Offline Reinforcement Learning，简称 Offline RL）是强化学习的一种设置，其中智能体的训练和学习是基于预先收集好的静态数据集，而不是通过与环境的实时交互获取数据。这与传统的在线强化学习不同，后者需要智能体在环境中实时地进行交互并收集数据。

关键特点和术语：

- 静态数据集：在离线强化学习中，智能体依赖于一个事先收集好的数据集，这个数据集可能是由专家策略、人类演示或者其他方法生成的。
- 全批次学习：在离线设置中，智能体不再实时地与环境交互，而是使用整个数据集进行离线训练。这与在线强化学习中的逐步迭代训练不同。
- 目标：离线强化学习的目标是从静态数据集中学习出一个高性能的策略，使得该策略在实际应用中能够获得较好的性能。
- 评估和改进：通常需要使用离线评估方法来估计学到的策略的性能，并可能采用一些特殊的算法来进行离线策略改进。

然而，离线强化学习也面临一些挑战，例如数据分布偏移、评估和改进的准确性等。近年来，研究人员一直在努力提出新的算法和技术，以解决这些挑战，推动离线强化学习在实际问题中的应用。本文所提出的模型则是离线强化学习领域的新工作，拥有超越之前 SOTA 的优秀性能。

3 本文方法

3.1 本文方法概述

此部分会详细介绍原文所提到的 Decision Transformer 模型，模型整体架构图如下图 3

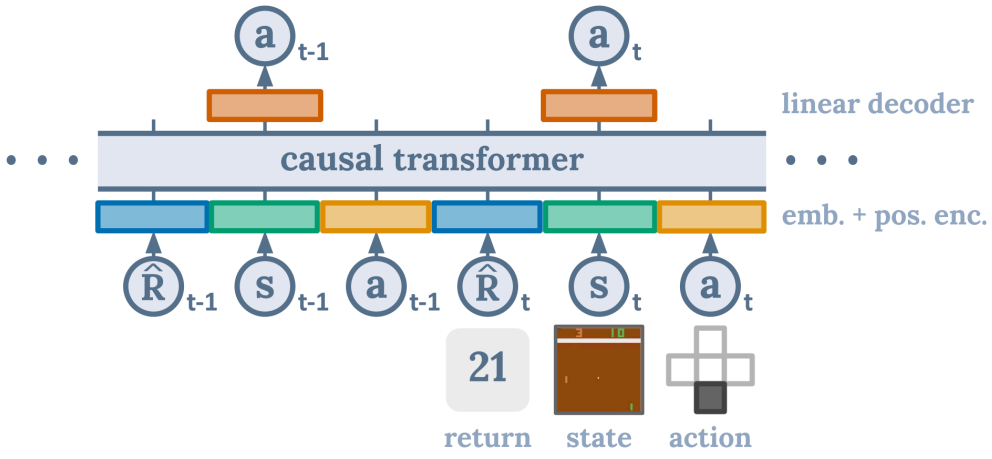

图 3. Decision Transformer

可以看出此模型可分为三部分，分别为序列输入部分、中间的 Transformer 部分以及最后的输出部分。由于 Transformer 部分仅为原版 Transformer，故不在此介绍，接下来主要介绍序列输入部分呢、输出部分以及整个模型的训练和预测过程。

3.2 序列输入模块

首先因为 transformer 是处理序列的，所以第一个问题就是输入一个怎样的序列给模型去学习，本篇文章中作者采用的输入序列就是下面这一行，分别是 t-1 时刻的 R（return，所采取行动之后环境给予的反馈奖励），t-1 时刻的 s（state，当前状态，比如如果是走迷宫游戏的话，状态就是所处的坐标），t-1 时刻的 a（action，也就是采取的动作和行动），之后便是 t 时刻的 R，s，a。将这样一个序列轮流输入进去让 transformer 去学习，但需要强调的一点是，这里的 R 在一般的强化学习模型当中所代表的都是此时此刻获得的 return 值，但本篇文章中，作者将 R 修改为了剩余收益，也就是说在模型开始之前会提前定一个目标收益，而这个 R 就是当前收益距离目标收益还有多远。这样做的目的是作者希望模型根据未来期望回报生成动作，而不是根据过去回报生成动作。

然后模型会为每个输入学习一个线性层，该层将原始输入投影到嵌入维度，从而更好输入到 transformer 里，当然，对于具有视觉输入的环境，状态会被送到卷积层而不是线性层。另外，模型还学习了位置编码（Positional Encoding），但与原始 transformer 中的位置编码不同的是，这里一个位置编码对应于三个输入，因为一个 R，s，a 是属于一个时间点的。

这便是对整个模型输入模块的介绍，经过以上处理的数据会被输入到 Transformer 模型中进行训练。

3.3 输出模块

输出模块相较于输入模块更加简单，模型会在每个时间节点的 s 输入时刻同时输出一个预测动作 a。Transformer 模型首先会将经过处理的数据输入给输出全连接层，之后再交由分类函数进行分类，根据环境性质的不同最后分类的类别数也不相同，有些连续决策动作环境最后也是连续的分类值。最后得到的概率最大的值便是此刻预测的下一个时刻的动作 a。

3.4 训练和预测过程

训练的时候会每次在离线轨迹集里抽取 k 个时间点的这个 R，s，a 组，输入进去，在每个时间点的 s 输入位置都会跟随一个输出 a，这个输出 a 就会跟轨迹集的 a 进行对比校验，计算损失函数，进行梯度的反向传播。最后预测部分则会根据训练好的模型，在每个 s 状态时间点给予一个预测动作 a，再把这个 a 作为输入，这个动作 a 与环境交互又会得到新的 R 以及 s，再将这个 R 和 s 输入到模型当中，一直循环重复，使整个模型的 R 向着一开始设定的 R 目标移动。这便是整个模型完整的训练以及预测过程的介绍。

4 复现细节

4.1 与已有开源代码对比

此次复现工作中，原 Decision Transformer 模型的代码均用自原文章的开源代码，模型主体代码如下图4和5所示：

```python
class DecisionTransformer(TrajectoryModel):

    """
    This model uses GPT to model (Return_1, state_1, action_1, Return_2, state_2, ...)
    """

    def __init__(
            self,
            state_dim,
            act_dim,
            hidden_size,
            max_length=None,
            max_ep_len=4096,
            action_tanh=True,
            **kwargs
    ):
        super().__init__(state_dim, act_dim, max_length=max_length)

        self.hidden_size = hidden_size
        config = transformers.GPT2Config(
            vocab_size=1,  # doesn't matter -- we don't use the vocab
            n_embd=hidden_size,
            **kwargs
        )

        # note: the only difference between this GPT2Model and the default Huggingface version
        # is that the positional embeddings are removed (since we'll add those ourselves)
        self.transformer = GPT2Model(config)

        self.embed_timestep = nn.Embedding(max_ep_len, hidden_size)
        self.embed_return = torch.nn.Linear( in_features: 1, hidden_size)
        self.embed_state = torch.nn.Linear(self.state_dim, hidden_size)
        self.embed_action = torch.nn.Linear(self.act_dim, hidden_size)

        self.embed_ln = nn.LayerNorm(hidden_size)

        # note: we don't predict states or returns for the paper
        self.predict_state = torch.nn.Linear(hidden_size, self.state_dim)
        self.predict_action = nn.Sequential(
            *([nn.Linear(hidden_size, self.act_dim)] + ([nn.Tanh()] if action_tanh else []))
        )
        self.predict_return = torch.nn.Linear(hidden_size,  out_features: 1)

    def forward(self, states, actions, rewards, returns_to_go, timesteps, attention_mask=None):

        batch_size, seq_length = states.shape[0], states.shape[1]

        if attention_mask is None:
            # attention mask for GPT: 1 if can be attended to, 0 if not
            attention_mask = torch.ones((batch_size, seq_length), dtype=torch.long)

        # embed each modality with a different head
        state_embeddings = self.embed_state(states)
        action_embeddings = self.embed_action(actions)
        returns_embeddings = self.embed_return(returns_to_go)
        time_embeddings = self.embed_timestep(timesteps)

        # time embeddings are treated similar to positional embeddings
        state_embeddings = state_embeddings + time_embeddings
        action_embeddings = action_embeddings + time_embeddings
        returns_embeddings = returns_embeddings + time_embeddings

        # this makes the sequence look like (R_1, s_1, a_1, R_2, s_2, a_2, ...)
        # which works nice in an autoregressive sense since states predict actions
        stacked_inputs = torch.stack(
            tensors: (returns_embeddings, state_embeddings, action_embeddings), dim=1
        ).permute( *dims: 0, 2, 1, 3).reshape(batch_size, 3*seq_length, self.hidden_size)
        stacked_inputs = self.embed_ln(stacked_inputs)

        # to make the attention mask fit the stacked inputs, have to stack it as well
        stacked_attention_mask = torch.stack(
            tensors: (attention_mask, attention_mask, attention_mask), dim=1
        ).permute( *dims: 0, 2, 1).reshape(batch_size, 3*seq_length)

        # we feed in the input embeddings (not word indices as in NLP) to the model
        transformer_outputs = self.transformer(
            inputs_embeds=stacked_inputs,
            attention_mask=stacked_attention_mask,
        )
```

图 4. 源代码 01

```
    transformer_outputs = self.transformer(
        inputs_embeds=stacked_inputs,
        attention_mask=stacked_attention_mask,
    )
    x = transformer_outputs['last_hidden_state']

    # reshape x so that the second dimension corresponds to the original
    # returns (0), states (1), or actions (2); i.e. x[:,1,t] is the token for s_t
    x = x.reshape(batch_size, seq_length, 3, self.hidden_size).permute(0, 2, 1, 3)

    # get predictions
    return_preds = self.predict_return(x[:,2])  # predict next return given state and action
    state_preds = self.predict_state(x[:,2])    # predict next state given state and action
    action_preds = self.predict_action(x[:,1])  # predict next action given state

    return state_preds, action_preds, return_preds

2 usages (2 dynamic)
def get_action(self, states, actions, rewards, returns_to_go, timesteps, **kwargs):
    # we don't care about the past rewards in this model

    states = states.reshape(1, -1, self.state_dim)
    actions = actions.reshape(1, -1, self.act_dim)
    returns_to_go = returns_to_go.reshape(1, -1, 1)
    timesteps = timesteps.reshape(1, -1)

    if self.max_length is not None:
        states = states[:,-self.max_length:]
        actions = actions[:,-self.max_length:]
        returns_to_go = returns_to_go[:,-self.max_length:]
        timesteps = timesteps[:,-self.max_length:]

        # pad all tokens to sequence length
        attention_mask = torch.cat([torch.zeros(self.max_length-states.shape[1]), torch.ones(states.shape[1])])
        attention_mask = attention_mask.to(dtype=torch.long, device=states.device).reshape(1, -1)
        states = torch.cat(
            tensors: [torch.
                      self.max_length-states.shape[1],
                      self.state_dim), device=states.device), states],
                dim=1).to(dtype=torch.float32)
        actions = torch.cat(
            tensors: [torch.zeros((actions.shape[0], self.max_length - actions.shape[1], self.act_dim),
                                  device=actions.device), actions],
                dim=1).to(dtype=torch.float32)
        returns_to_go = torch.cat(
            tensors: [torch.zeros((returns_to_go.shape[0],
                      self.max_length-returns_to_go.shape[1], 1), device=returns_to_go.device), returns_to_go],
                dim=1).to(dtype=torch.float32)
        timesteps = torch.cat(
            tensors: [torch.zeros((timesteps.shape[0],
                      self.max_length-timesteps.shape[1]), device=timesteps.device), timesteps],
                dim=1
            ).to(dtype=torch.long)
    else:
        attention_mask = None

    _, action_preds, return_preds = self.forward(
        states, actions, rewards: None, returns_to_go, timesteps, attention_mask=attention_mask, **kwargs)

    return action_preds[0,-1]
```

Multiple implementations

图 5. 源代码 02

在此基础上，我将原模型的全连接层替换为了 GLU 模块，以下为我编写的 GLU 模块代码，将其替换到原代码的全连接部分即可构成这次复现工作的全部完整代码：

```
class GPT2WithGLU(nn.Module):
    def __init__(self, config):
        super(GPT2WithGLU, self).__init__()
        self.transformer = GPT2Model(config)
        self.glu_layer = nn.Linear(config.n_embd, config.n_embd * 2)  # Assuming hidden size is config.n_embd

    def forward(self, inputs_embeds, attention_mask=None):
        transformer_outputs = self.transformer(inputs_embeds=inputs_embeds, attention_mask=attention_mask)
        hidden_states = transformer_outputs['last_hidden_state']

        # Apply GLU layer
        glu_output = self.glu_layer(hidden_states)
        glu_output = F.glu(glu_output, dim=-1)

        # Further processing or downstream tasks...

        return glu_output
```

图 6. GLU

4.2 实验环境搭建

安装 d4rl 环境需要首先安装 mujoco，MuJoCo（Multi-Joint dynamics with Contact）是一种物理模拟引擎，用于模拟多关节动态系统的运动和物理交互。MuJoCo 最初是为仿真生物运动而设计的，但它也被广泛用于强化学习领域，特别是深度强化学习。

在强化学习中，MuJoCo 通常用于创建仿真环境，供智能体（代理）在其中学习和执行任务。这些任务可以涉及控制机器人、学习复杂动作策略或其他需要对物理世界进行建模和交互的问题。

安装好 mujoco 环境后即可运行下图 yml 文件，配置所有此模型运行所需环境：

```yaml
name: d4rl_env1
channels:
- pytorch
dependencies:
- python=3.8.5
- anaconda
- cudatoolkit=10.
- numpy
- pip
- pip:
  - gym==0.18.3
  - mujoco-py==2.0.2.13
  - numpy==1.ß20.3
  - torch==1.8.1
  - transformers==4.5.1
  - wandb==0.9.1
```

图 7. 环境 yml 文件

环境配置完成后即可远程通过指令下载离线轨迹集，下载完成后即完成所有环境的搭建，模型可以开始运行。

4.3 创新点

在此模型的基础上做出的创新点在于将原模型最后输出的全连接结构替换为了 GLU 结构 (如图8)，Gated Linear Unit [5]（门控线性单元，简称 GLU）是一种用于神经网络的激活函数，常用于序列建模任务，特别是在语言建模和自然语言处理（NLP）领域。GLU 最初用于 Transformer 模型的编码器部分，并在后来的深度学习架构中得到了广泛的应用。

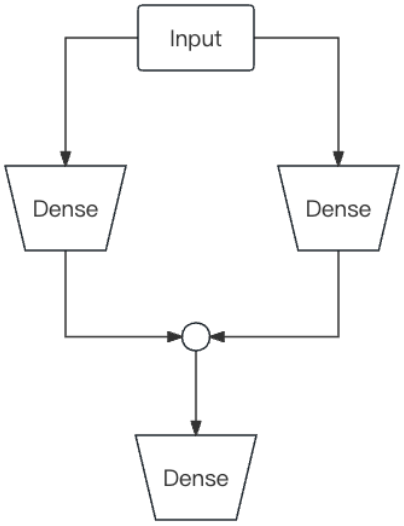

图 8. GLU 结构图

GLU 的门控机制允许模型在学习过程中选择性地保留或抑制输入信息，有助于提高模型对序列中重要部分的关注度。在 Transformer 编码器中，GLU 的引入使得模型能够更有效地捕捉序列中的局部特征和长程依赖关系，从而提高了其性能。

在这里加入 GLU 结构的思路是由于 GLU 结构能很好的捕捉局部特征并且更好的处理从模型中输出的数据，更好的能将其变为分类结果，所以我认为使用它替换掉原本的全连接层效果会更好，最后的实验结果也发现在一些数据集中，替换后的 DT 模型效果要好于替换之前的。

5 实验结果分析

本次复现任务中，我一共跑了两个模型，分别为原版 DT 模型以及改进过后的 DT 模型，在三个环境中运行 10 个 iteration 后的结果如下图所示：

Game	Improve-DT	DT	CQL	BEAR	BRAC-v	BC	AWR
Hopper-medium	2257.2	2196.8	**2557.3**	1674.5	990.4	923.5	1149.5
Hopper-medium-replay	**2610.3**	2450.0	1227.3	1076.8	-0.8	364.4	904.0
Walker2d-medium-replay	3148.9	**3524.9**	1227.3	833.8	44.5	518.6	712.5

图 9. 实验结果表格

其中，Improve-DT 是我改进后的模型，DT 是原模型，CQL 是 DT 模型提出前的最佳离线强化学习模型，BEAR、BEAR-v、BC 和 AWR 同样是以往的离线强化学习模型。

从表格中可以看出，DT 模型在后两个环境中的效果要远远高于 CQL，甚至在第三个环境中的效果达到了 CQL 的三倍。而改进后的 DT 在第一个和第二个环境中的效果要好于改进之前的 DT，虽然第一个环境的效果依然没有 CQL 好，不过在第二个环境中改进后的 DT 已经达到了目前的最佳效果。从这样的表格可以看出原版 DT 性能好于之前的 SOTA，改进后的 DT 在一些环境中效果也要好于改进之前的 DT。

6 总结与展望

这篇文章提出了 Decision Transformer，我认为其有如下优点以及缺点
优点：

- 拥有卓越的性能：在绝大多数环境中效果都要好于之前的 SOTA：CQL 模型，在个别环境中效果甚至要好于其两到三倍，这证明这种建模思路很有发展空间
- 首次将 Transformer 与强化学习结合：首次结合了两个热门领域的工作，之后在 Transformer 模型上做出的改良也可更方便的移用到强化学习领域，对于帮助强化学习领域发展有着很重要的意义

缺点：

- 只适用于离线强化学习：本次提出的模型暂时无法移用到在线强化学习任务当中，这是以后可以发展的方向

参考文献

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[2] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

[3] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[4] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

[5] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017.