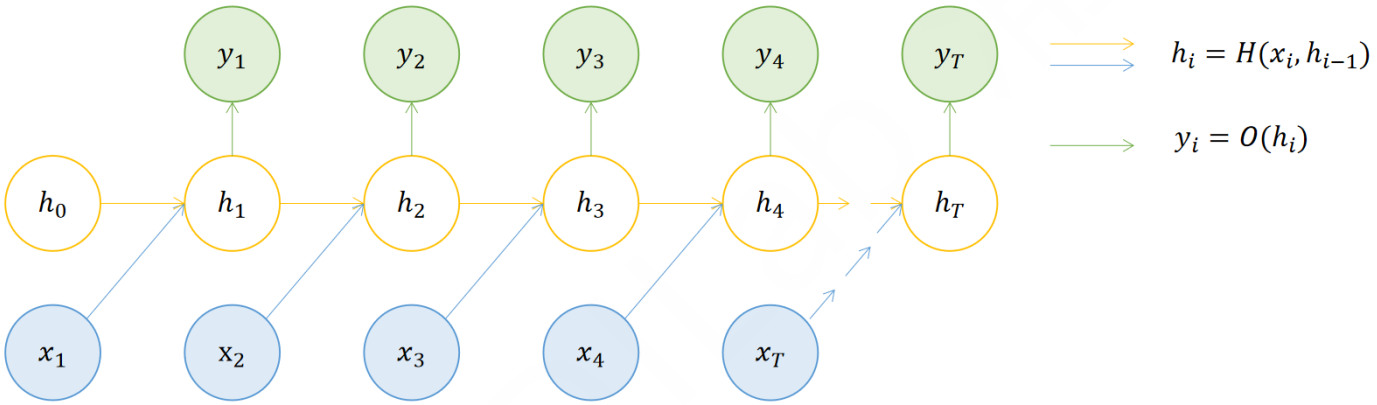
PPO × Family 第五讲习题作业答案

算法理论题

题目一（RNN 梯度计算分析）

序列模型（Sequential model）在机器学习领域有非常广泛的应用，尤其是在时序信息处理，自然语言处理等领域。下图是一个由 Recurrent Neural Network （RNN）组成的序列到序列（Seq2Seq）模型的示意图，它的输入 x_t 是一个序列，$x_t \in \mathbb{R}^{d_x}$，输出也是一个序列，$o_t \in \mathbb{R}^{d_y}$，序列最大长度为 T：



每一个时刻输入的信息 x_t，都会与上一个时刻的隐藏状态 h_{t-1}，一起通过转移函数 H，生成当前时刻的隐藏状态：

$$h_t = H(x_t, h_{t-1} \mid \theta)$$

$$H : \mathbb{R}^{d_h \times d_x} \to \mathbb{R}^{d_h}$$

而每个时刻的隐藏状态 h_t，通过一个输出函数 $o_t = O(h_t \mid \theta)$，输出当前时刻隐藏状态 h_t 的对应输出值 o_t。从上图中，以及转移函数的定义式中，我们可以发现，各个时刻之间的信息是循环展开和传递的，在每一个时刻隐藏状态都会被当前时刻的输入所更新，并将这种变化的信息通过输出值传递出来。通过定义一些优化目标函数（比如各类损失函数），可以用于度量模型的输出 $o_{1:T}$ 和真实数据 $y_{1:T}$ 之间的差别。具体来说，将单个时刻的损失函数记为 $l(o_t, y_t)$，则所有时刻的平均损失为：

$$L = \frac{1}{T} \sum_{i=0}^{T} l(o_i(x_{1:i}), y_i)$$

在第 $t = i$ 时刻的损失关于模型参数的梯度，会通过时刻 $t = i - 1$ 的隐藏状态向过去传递。

问题1：请推导对于一组序列数据，$(x_i, y_i) \in \mathbb{R}^{(d_x+d_y)\times T}$，当 $d_x = 1$，$d_y = 1$，$T = 3$ 时这个 RNN 模型对其参数 θ 的梯度表达式。

（提示：可以将损失函数的梯度按照以下式子展开，并以此为基础续写和归纳至 $T = N$）

$$
\begin{aligned}
\frac{\partial L}{\partial \theta} &= \frac{1}{T}\sum_{i=0}^{T}\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial \theta} \\
&= \frac{1}{T}\sum_{i=0}^{T}[\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}\frac{\partial h_i}{\partial \theta}] \\
&= \frac{1}{T}\sum_{i=0}^{T}[\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}(\frac{\partial H(x_i, h_{i-1})}{\partial \theta} + \frac{\partial H(x_i, h_{i-1})}{\partial h_{i-1}}\frac{\partial h_{i-1}}{\partial \theta})] \\
&= \frac{1}{T}\sum_{i=0}^{T}[\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}(\frac{\partial H(x_i, h_{i-1})}{\partial \theta} + \frac{\partial H(x_i, h_{i-1})}{\partial h_{i-1}}(\frac{\partial H(x_{i-1}, h_{i-2})}{\partial \theta} + \frac{\partial H(x_{i-1}, h_{i-2})}{\partial h_{i-2}}\frac{\partial h_{i-2}}{\partial \theta}))]
\end{aligned}
$$

答案：

根据第一问提示以及梯度的计算方法，当 $d_x = 1$，$d_y = 1$，$T = 3$ 时，有：

$$
\begin{aligned}
\frac{\partial L}{\partial \theta} &= \frac{1}{3}\sum_{i=0}^{3}\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial \theta} \\
&= \frac{1}{3}\sum_{i=0}^{3}[\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}\frac{\partial h_i}{\partial \theta}] \\
&= \frac{1}{3}\sum_{i=0}^{3}[\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}(\frac{\partial H(x_i, h_{i-1})}{\partial \theta} + \frac{\partial H(x_i, h_{i-1})}{\partial h_{i-1}}\frac{\partial h_{i-1}}{\partial \theta})] \\
&= \frac{1}{3}\sum_{i=0}^{3}[\frac{\partial l(o_i(x_{1:i}), y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}(\frac{\partial H(x_i, h_{i-1})}{\partial \theta} + \frac{\partial H(x_i, h_{i-1})}{\partial h_{i-1}}(\frac{\partial H(x_{i-1}, h_{i-2})}{\partial \theta} + \frac{\partial H(x_{i-1}, h_{i-2})}{\partial h_{i-2}}\frac{\partial h_{i-2}}{\partial \theta}))] \\
&= \frac{1}{3}[\frac{\partial l(o_3, y_3)}{\partial o_3}\frac{\partial O(h_3)}{\partial h_3}(\frac{\partial H(x_3, h_2)}{\partial \theta} + \frac{\partial H(x_3, h_2)}{\partial h_2}(\frac{\partial H(x_2, h_1)}{\partial \theta} + \frac{\partial H(x_2, h_1)}{\partial h_1}(\frac{\partial H(x_1, h_0)}{\partial \theta} + \frac{\partial H(x_1, h_0)}{\partial h_0})))] \\
&+ \frac{1}{3}[\frac{\partial l(o_2, y_2)}{\partial o_2}\frac{\partial O(h_2)}{\partial h_2}(\frac{\partial H(x_2, h_1)}{\partial \theta} + \frac{\partial H(x_2, h_1)}{\partial h_1}(\frac{\partial H(x_1, h_0)}{\partial \theta} + \frac{\partial H(x_1, h_0)}{\partial h_0}))] \\
&+ \frac{1}{3}[\frac{\partial l(o_1, y_1)}{\partial o_1}\frac{\partial O(h1)}{\partial h_1}(\frac{\partial H(x_1, h_0)}{\partial \theta} + \frac{\partial H(x_1, h_0)}{\partial h_0})]
\end{aligned}
$$

问题2：在实践中，简单的 RNN 模型比较容易因为出现梯度消失 [1] 或梯度爆炸 [2] 的现象而使得训练变得困难。请通过分析问题1中得到的 RNN 模型的梯度表达式，讨论梯度消失现象的原因。（此外，可以回顾课程内容，了解处理这类梯度问题的一些方法。）

答案：

从上一问的解答中可见，RNN 模型的目标表达式中出现了类似如下形式的项：

$$[\frac{\partial l(o_i,y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}\frac{\partial H(x_i,h_{i-1})}{\partial h_{i-1}}\frac{\partial H(x_{i-1},h_{i-2})}{\partial h_{i-2}}\frac{\partial H(x_{i-2},h_{i-3})}{\partial h_{i-3}}\frac{\partial H(x_{i-3},h_{i-4})}{\partial h_{i-4}}...]$$

这意味着，对于第 i 时刻的损失 $l(o_i,y_i)$，它的存在所带来的对于模型参数的梯度变化值，是损失函数局部的导数，$\frac{\partial l(o_i,y_i)}{\partial o_i}\frac{\partial O(h_i)}{\partial h_i}$，乘以一个因为梯度传导带来的连乘的系数，$\prod[\frac{\partial H(x_i,h_{i-1})}{\partial h_{i-1}}]$，其长度取决于两个时刻间的距离。由于神经网络中参数初始化、激活函数类型等等因素的选取的差异，$\prod[\frac{\partial H(x_i,h_{i-1})}{\partial h_{i-1}}]$ 中可能出现多数项都小于 1，即 $\prod(\leq 1)\approx 0$，此时为梯度消失现象，即各个时刻之间的梯度关系无法进行远距离传播。而当 $\prod[\frac{\partial H(x_i,h_{i-1})}{\partial h_{i-1}}]$ 中可能出现多数项都大于 1 时，即 $\prod(\geq 1)\approx \infty$，此时为梯度爆炸现象，表现为较远时刻的损失导致很大的梯度波动。

题目二（Belief MDP）

马尔可夫决策过程（Markov decision process，MDP）是一个可获得完全信息的决策过程，智能体在每个时刻可以观测到其真实的状态，$s_t \in S$，在经历行动 $a_t \in A$ 之后，可以观测到真实的状态转移

$$T(s_t,a_t,s_{t+1})=P(s_{t+1}|s_t,a_t):S\times A\times \mathcal{S}\to \mathbb{R}^+,$$

并可以在每个时刻获得奖励的具体数值

$$r_t(s_t,a_t):S\times A\to \mathbb{R}。$$

但对于那些由于各种客观原因的限制，**无法获取完全信息**的场景和其中的系统，假定其真实状态的动力学依然由一个真实不变的 MDP 所决定，只是智能体不能直接观察底层真实状态，仅能观察到其中一部分的信息 $o_t \in O$（这里的记号 O 代表观测，即 $Observation$）。那么，则可以称这个决策过程为一个部分观测马尔可夫决策过程（Partially observable Markov decision process，POMDP）。

相比于马尔可夫决策过程，部分观测马尔可夫决策过程可以额外引入观测信息 o_t 和真实状态 s_t 之间的函数关系，称为观测函数：

$$\Pi(o_{t+1},s_t,a_t)=P(o_{t+1}|s_t,a_t):S\times A\times \mathcal{O}\to \mathbb{R}^+。$$

虽然智能体无法获知自己的真实状态，但是可以建立一种对真实状态的某种估计，称之为信念（Belief State）：

$$b(s_t)=P(s_t):\mathcal{S}\to \mathbb{R}^+$$

也就是说，对于某一个时刻，智能体对当前时刻的真实状态的具体数值有一个实际的分布作为其估计。

通过更多次与环境的交互，可以利用交互带来的信息，结合贝叶斯定理逐步更新对当前真实状态信念的估计，一般将更新后的信念记为 $b'(s_t)$。比如对于离散的场景下，其利用贝叶斯定理的更新形式为：

$$b'(s_{t+1})=P(s_{t+1}|o_{t+1},a_t,b(s_t))=\frac{P(o_{t+1}|s_{t+1},a_t,b(s_t))P(s_{t+1}|a_t,b(s_t))}{\sum_{s_{t+1}\in S}[P(o_{t+1}|s_{t+1},a_t,b(s_t))P(s_{t+1}|a_t,b(s_t))]}$$

上式中，由于真实状态具有观测信息的完备统计量，故有：

$$P(o_{t+1}|s_{t+1},a_t,b(s_t))=P(o_{t+1}|s_{t+1})$$

而对于下一时刻状态对于上一时刻信念和动作的条件先验，可以使用转移概率展开求解：

$$P(s_{t+1}|a_t,b(s_t))=\sum_{s_t\in S}P(s_{t+1}|s_t,a_t)b(s_t)$$

这样一来，对于每一个时刻，我们都可以对当前的奖励做一个基于信念估计的函数计算：

$$R(a_t,b(s_t))=\sum_{s_t\in S}r(a_t,s_t)b(s_t)$$

这样我们就让一个POMDP问题出现了一些熟悉的元素，比如我们用对于状态的信念，来替代 MDP 中的状态。用奖励函数来替代 MDP 中的奖励。然后我们就可以使用一些适用于 MDP 的算法来近似解决 POMDP 问题。

为了让大家熟悉 POMDP 中的信念更新，本次作业题将涉及一个相关入门小例子：

胡图图同学对 OpenDILab 开设的 PPO × Family 课程非常感兴趣，想要观看直播和回放，不过因为忘记了登录电子设备的密码，遇到了一点麻烦。由于多次尝试错误，系统需要每隔100分钟才能再次尝试（尝试出错的惩罚为 -100，成功打开设备的奖励为 $+10$）。他大概记得密码应该是 A 和 B 中的一个，都有可能。不过由于直播马上就要开始了，他现在比较苦恼。假设胡图图同学仔细冷静思考1分钟（思考的成本为 -1），可以大概确定个八九不离十，想起来具体是 A 还是 B。假如标记想起来的密码为 o_t，真实的密码为 s_t，思考的动作为 $a_t=a_0$，输入密码的动作为 $a_t=a_A$ 或 $a_t=a_B$，即在这里我们可以认为：

$$P(o_{t+1}=A|s_t=A,a_t=a_0)=0.85\text{，}\ P(o_{t+1}=B|s_t=B,a_t=a_0)=0.85$$

问题1：假如刚开始胡图图对真实密码是 A 还是 B 的信念**相同**，那么假如他冷静思考之后，信念如何更新？请根据贝叶斯定理，计算思考后的真实状态的信念的分布。

答案：

刚开始胡图图的信念状态为：

$b(s_t=A)=0.5$，$b(s_t=B)=0.5$

根据题干中贝叶斯定理的更新公式，有：

$$\begin{aligned}b'(s_{t+1}=A)&=P(s_{t+1}=A|o_{t+1}=A,a_t=a_0,b(s_t))\\&=\frac{P(o_{t+1}=A|s_{t+1}=A,a_t=a_0,b(s_t))P(s_{t+1}=A|a_t=a_0,b(s_t))}{P(o_{t+1}=A|s_{t+1}=A,a_t=a_0,b(s_t))P(s_{t+1}=A|a_t=a_0,b(s_t))+P(o_{t+1}=A|s_{t+1}=B,a_t=a_0,b(s_t))P(s_{t+1}=B|a_t=a_0,b(s_t))}\end{aligned}$$

根据转移概率可以计算获得：

$$
\begin{aligned}
P(s_{t+1}=A|a_t=a_0,b(s_t)) &= \sum_{s_t\in S} P(s_{t+1}=A|s_t,a_t=a_0)b(s_t) \\
&= P(s_{t+1}=A|s_t=A,a_t=a_0)b(s_t=A)+P(s_{t+1}=A|s_t=B,a_t=a_0)b(s_t=B) \\
&= 1\times 0.5+0\times 0.5=0.5
\end{aligned}
$$

$$
\begin{aligned}
P(s_{t+1}=B|a_t=a_0,b(s_t)) &= \sum_{s_t\in S} P(s_{t+1}=B|s_t,a_t=a_0)b(s_t) \\
&= P(s_{t+1}=B|s_t=A,a_t=a_0)b(s_t=A)+P(s_{t+1}=B|s_t=B,a_t=a_0)b(s_t=B) \\
&= 0\times 0.5+1\times 0.5=0.5
\end{aligned}
$$

由于动作 a_0 并不改变真实状态，即 $s_t=s_{t+1}$ ，因此有：

$$
P(o_{t+1}=A|s_{t+1}=A,a_t=a_0,b(s_t))=P(o_{t+1}=A|s_t=A,a_t=a_0,b(s_t))
$$

此外由于真实状态具有观测信息的完备统计量，所以有：

$$
P(o_{t+1}=A|s_t=A,a_t=a_0,b(s_t))=P(o_{t+1}=A|s_t=A,a_t=a_0)=0.85
$$

同理可得：

$$
\begin{aligned}
P(o_{t+1}=A|s_{t+1}=B,a_t=a_0,b(s_t)) &= P(o_{t+1}=A|s_t=B,a_t=a_0,b(s_t)) \\
&= P(o_{t+1}=A|s_t=B,a_t=a_0) \\
&= 0.15
\end{aligned}
$$

所以胡图图思考后的真实状态的信念更新为：

$$
\begin{aligned}
b'(s_{t+1}=A) &= P(s_{t+1}=A|o_{t+1}=A,a_t=a_0,b(s_t)) \\
&= \frac{P(o_{t+1}=A|s_{t+1}=A,a_t=a_0,b(s_t))P(s_{t+1}=A|a_t=a_0,b(s_t))}{P(o_{t+1}=A|s_{t+1}=A,a_t=a_0,b(s_t))P(s_{t+1}=A|a_t=a_0,b(s_t))+P(o_{t+1}=A|s_{t+1}=B,a_t=a_0,b(s_t))P(s_{t+1}=B|a_t=a_0,b(s_t))} \\
&= \frac{0.85\times 0.5}{0.85\times 0.5+0.15\times 0.5}=0.85
\end{aligned}
$$

$$
b'(s_{t+1}=B)=0.15
$$

问题2：假如刚开始胡图图对真实密码是 A 还是 B 的信念**不相同**，为了让奖励期望最大，胡图图需要拥有多少程度的信念，才应该进行密码输入？

$$
R(a_t=a_0,b(s_t))=\sum_{s_t\in S} r(a_t=a_0,s_t)b(s_t)=-1
$$

$$
\begin{aligned}
R(a_t=a_A,b(s_t)) &= \sum_{s_t\in S} r(a_t=a_A,s_t)b(s_t) \\
&= r(a_t=a_A,s_t=A)b(s_t=A)+r(a_t=a_A,s_t=B)b(s_t=B) \\
&= 10\times\ b(s_t=A)-100\times\ b(s_t=B) \\
&= 10\times\ b(s_t=A)-100\times(1-b(s_t=A)) \\
&= -100+110\times\ b(s_t=A)
\end{aligned}
$$

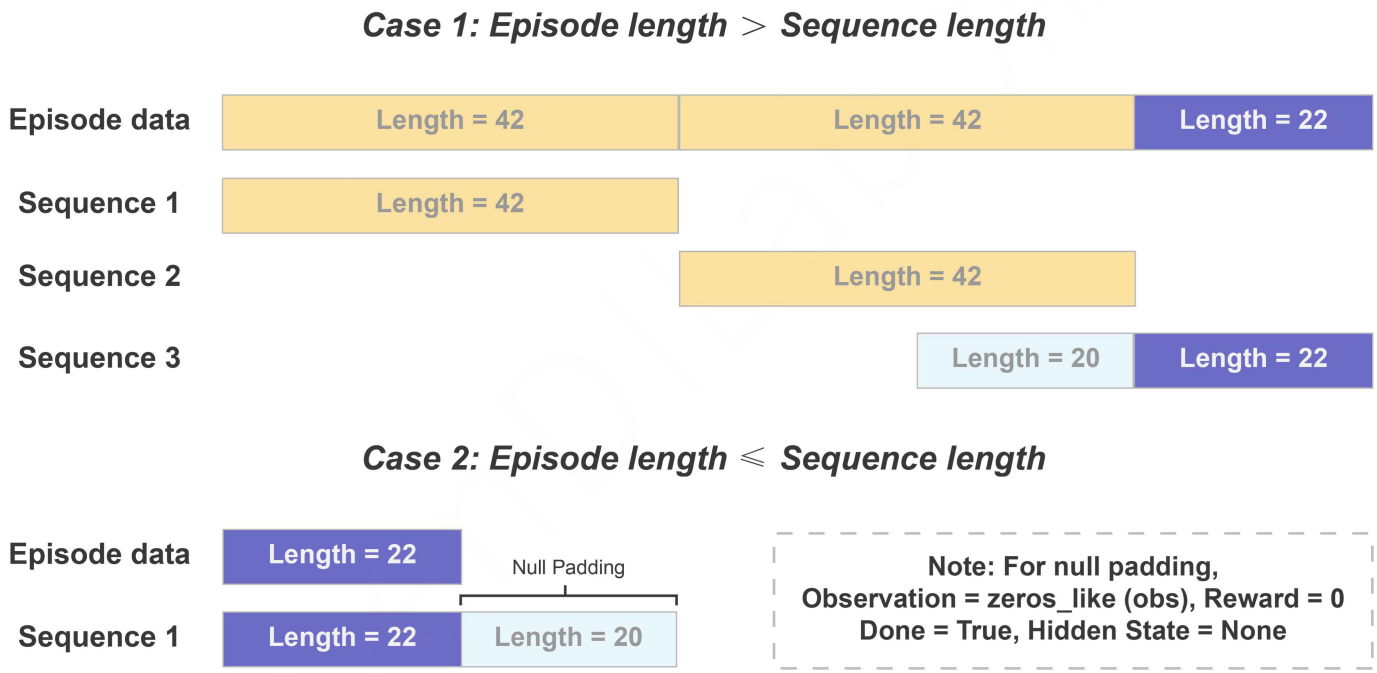
经过比较可以看到，当 $b(s_t = A) > \frac{99}{110} = 0.9$ 时，有 $R(a_t = a_A, b(s_t) > R(a_t = a_0, b(s_t)$，即，胡图图需要拥有至少为 0.9 的信念 $b(s_t = A)$，才应该进行密码 A 的输入。同理，胡图图需要拥有至少为 0.9 的信念 $b(s_t = B)$，才应该进行密码 B 的输入。

代码实践题

题目一（变长序列处理）

请根据课程第五讲 LSTM 部分讲到的变长序列处理方法，填补实现完成下方示例代码中的 `pack_data` 函数，并运行相应的测试函数，通过所有断言（Assertion）语句。

变长序列处理方法示意图如下：



完整代码如下（也可从官网链接下载）：

```python
"""
Long Short Term Memory (LSTM) <link https://ieeexplore.ieee.org/abstract/document/6795963 link> is a kind of recurrent neural network that can capture long-short term information.
This document mainly includes:
- Pytorch implementation for LSTM.
- An example to test LSTM.
For beginners, you can refer to <link https://zhuanlan.zhihu.com/p/32085405 link> to learn the basics about how LSTM works.
"""
from typing import Optional, Union, Tuple, List, Dict
```

```python
import math
import torch
import torch.nn as nn
from ding.torch_utils import build_normalization

class LSTM(nn.Module):
    """
    **Overview:**
        Implementation of LSTM cell with layer norm.
    """

    def __init__(
            self,
            input_size: int,
            hidden_size: int,
            num_layers: int,
            norm_type: Optional[str] = 'LN',
            dropout: float = 0.
    ) -> None:
        # Initialize arguments.
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        # Initialize normalization functions.
        norm_func = build_normalization(norm_type)
        self.norm = nn.ModuleList([norm_func(hidden_size * 4) for _ in range(2 * num_layers)])
        # Initialize LSTM parameters.
        self.wx = nn.ParameterList()
        self.wh = nn.ParameterList()
        dims = [input_size] + [hidden_size] * num_layers
        for l in range(num_layers):
            self.wx.append(nn.Parameter(torch.zeros(dims[l], dims[l + 1] * 4)))
            self.wh.append(nn.Parameter(torch.zeros(hidden_size, hidden_size * 4)))
        self.bias = nn.Parameter(torch.zeros(num_layers, hidden_size * 4))
        # Initialize the Dropout Layer.
        self.use_dropout = dropout > 0.
        if self.use_dropout:
            self.dropout = nn.Dropout(dropout)
        self._init()

    # Dealing with different types of input and return preprocessed prev_state.
    def _before_forward(self, inputs: torch.Tensor, prev_state: Union[None, List[Dict]]) -> torch.Tensor:
        seq_len, batch_size = inputs.shape[:2]
```

```
        # If prev_state is None, it indicates that this is the beginning of a sequence. In this case, prev_state will be initialized as zero.
        if prev_state is None:
            zeros = torch.zeros(self.num_layers, batch_size, self.hidden_size, dtype=inputs.dtype, device=inputs.device)
            prev_state = (zeros, zeros)
        # If prev_state is not None, then preprocess it into one batch.
        else:
            assert len(prev_state) == batch_size
            state = [[v for v in prev.values()] for prev in prev_state]
            state = list(zip(*state))
            prev_state = [torch.cat(t, dim=1) for t in state]

        return prev_state

    def _init(self):
        # Initialize parameters. Each parameter is initialized using a uniform distribution of: $$U(-\sqrt {\frac 1 {HiddenSize}}, -\sqrt {\frac 1 {HiddenSize}})$$
        gain = math.sqrt(1. / self.hidden_size)
        for l in range(self.num_layers):
            torch.nn.init.uniform_(self.wx[l], -gain, gain)
            torch.nn.init.uniform_(self.wh[l], -gain, gain)
            if self.bias is not None:
                torch.nn.init.uniform_(self.bias[l], -gain, gain)

    def forward(
            self,
            inputs: torch.Tensor,
            prev_state: torch.Tensor,
    ) -> Tuple[torch.Tensor, Union[torch.Tensor, list]]:
        # The shape of input is: [sequence length, batch size, input size]
        seq_len, batch_size = inputs.shape[:2]
        prev_state = self._before_forward(inputs, prev_state)

        H, C = prev_state
        x = inputs
        next_state = []
        for l in range(self.num_layers):
            h, c = H[l], C[l]
            new_x = []
            for s in range(seq_len):
                # Calculate $$z, z^i, z^f, z^o$$ simultaneously.
                gate = self.norm[l * 2](torch.matmul(x[s], self.wx[l])
                                        ) + self.norm[l * 2 + 1](torch.matmul(h, self.wh[l]))
                if self.bias is not None:
```

```
                    gate += self.bias[l]
                gate = list(torch.chunk(gate, 4, dim=1))
                i, f, o, z = gate
                # $$z^i = \sigma (Wx^ix^t + Wh^ih^{t-1})$$
                i = torch.sigmoid(i)
                # $$z^f = \sigma (Wx^fx^t + Wh^fh^{t-1})$$
                f = torch.sigmoid(f)
                # $$z^o = \sigma (Wx^ox^t + Wh^oh^{t-1})$$
                o = torch.sigmoid(o)
                # $$z = tanh(Wxx^t + Whh^{t-1})$$
                z = torch.tanh(z)
                # $$c^t = z^f \odot c^{t-1}+z^i \odot z$$
                c = f * c + i * z
                # $$h^t = z^o \odot tanh(c^t)$$
                h = o * torch.tanh(c)
                new_x.append(h)
            next_state.append((h, c))
            x = torch.stack(new_x, dim=0)
            # Dropout layer.
            if self.use_dropout and l != self.num_layers - 1:
                x = self.dropout(x)
        next_state = [torch.stack(t, dim=0) for t in zip(*next_state)]
        # Return list type, split the next_state .
        h, c = next_state
        batch_size = h.shape[1]
        # Split h with shape [num_layers, batch_size, hidden_size] to a list with length batch_size and each element is a tensor with shape [num_layers, 1, hidden_size]. The same operation is performed on c.
        next_state = [torch.chunk(h, batch_size, dim=1), torch.chunk(c, batch_size, dim=1)]
        next_state = list(zip(*next_state))
        next_state = [{k: v for k, v in zip(['h', 'c'], item)} for item in next_state]
        return x, next_state

def pack_data(data: List[torch.Tensor], traj_len: int) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Overview:
        You need to pack variable-length data to regular tensor, return tensor and corresponding mask.
        If len(data_i) < traj_len, use `null_padding`,
        else split the whole sequences info different trajectories.
    Returns:
        - tensor (:obj:`torch.Tensor`): dtype (torch.float32), shape (traj_len, B, N)
```

```
        - mask (:obj:`torch.Tensor`): dtype (torch.float32), shape (traj_len, B)
    """
    new_data = []
    mask = []
    for item in data:
        D, N = item.shape
        if D < traj_len:
            null_padding = torch.zeros(traj_len - D, N)
            new_item = torch.cat([item, null_padding])
            new_data.append(new_item)
            item_mask = torch.ones(traj_len)
            item_mask[D:].zero_()
            mask.append(item_mask)
        else:
            for i in range(0, D, traj_len):
                item_mask = torch.ones(traj_len)
                new_item = item[i:i + traj_len]
                if new_item.shape[0] < traj_len:
                    new_item = item[-traj_len:]
                new_data.append(new_item)
                mask.append(torch.ones(traj_len))
    new_data = torch.stack(new_data, dim=1)
    mask = torch.stack(mask, dim=1)

    return new_data, mask

def test_lstm():
    seq_len_list = [32, 49, 24, 78, 45]
    traj_len = 32
    N = 10
    hidden_size = 32
    num_layers = 2

    variable_len_data = [torch.rand(s, N) for s in seq_len_list]
    input_, mask = pack_data(variable_len_data, traj_len)
    assert isinstance(input_, torch.Tensor), type(input_)
    batch_size = input_.shape[1]
    assert batch_size == 9, "packed data must have 9 trajectories"
    lstm = LSTM(N, hidden_size=hidden_size, num_layers=num_layers, norm_type='LN', dropout=0.1)

    prev_state = None
    for s in range(traj_len):
        input_step = input_[s:s + 1]
        output, prev_state = lstm(input_step, prev_state)

```

```
    assert output.shape == (1, batch_size, hidden_size)
    assert len(prev_state) == batch_size
    assert prev_state[0]['h'].shape == (num_layers, 1, hidden_size)
    loss = (output * mask.unsqueeze(-1)).mean()
    loss.backward()
    for _, m in lstm.named_parameters():
        assert isinstance(m.grad, torch.Tensor)
    print('finished')

if __name__ == '__main__':
    test_lstm()

```

题目二（应用实践）

在课程第五讲（解密稀疏奖励空间）几个应用中任选一个

- Pong（使用叠帧和 PPO + LSTM）
- Memory Len（使用 PPO + GTrXL）

根据课程组给出的示例代码，训练得到相应的智能体。最终提交需要上传相关训练代码、日志截图或最终所得的智能体效果视频（replay），具体样式可以参考第五讲的示例 ISSUE。