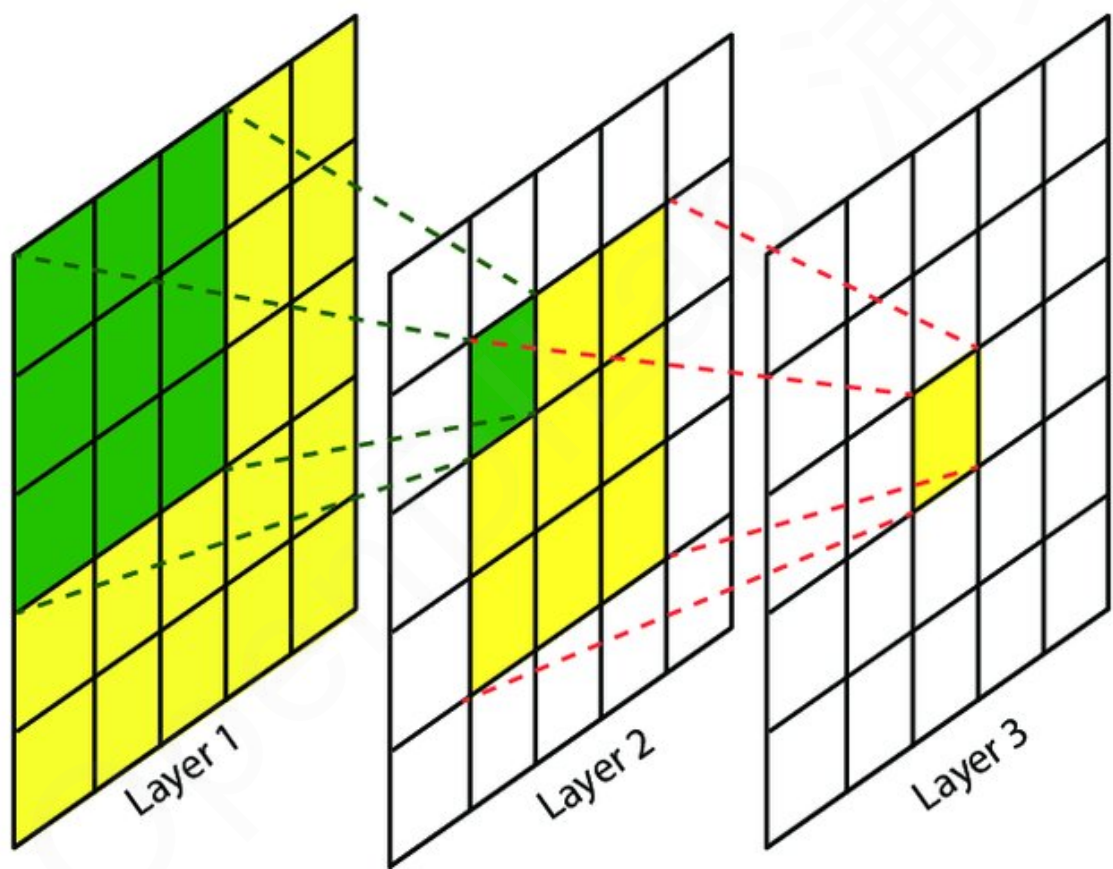


PPO × Family 第三讲习题与答案

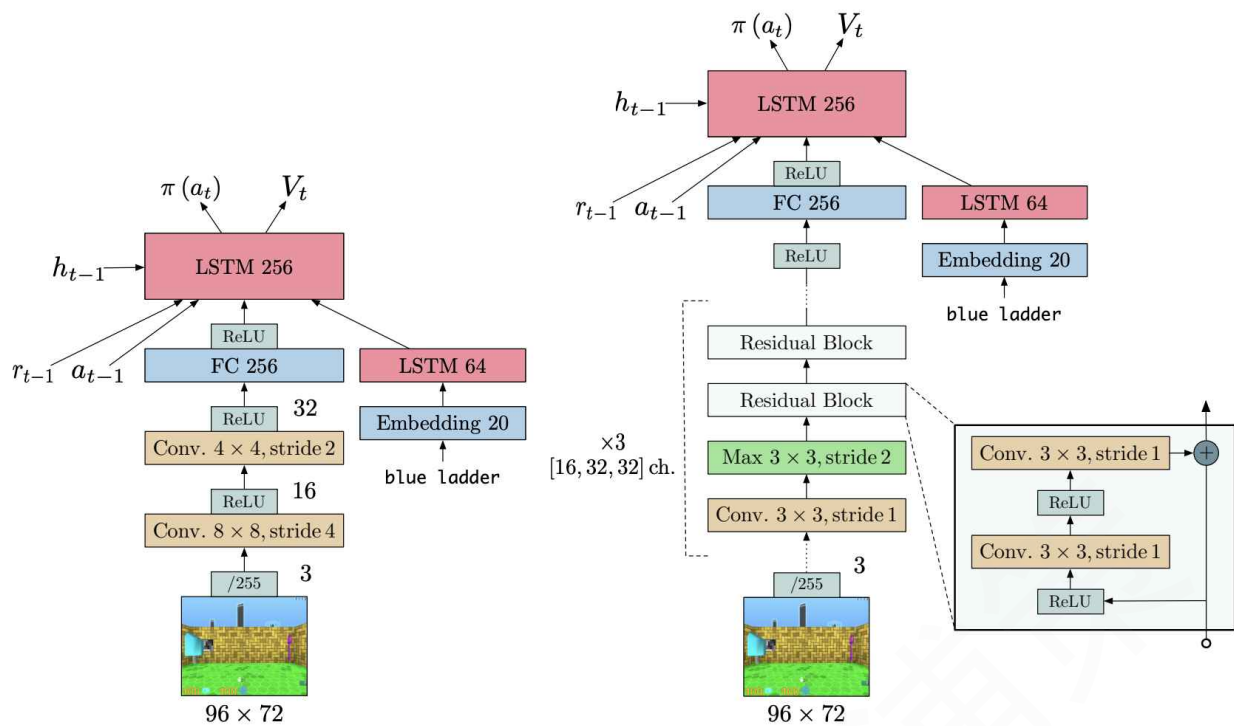
算法理论题

题目1（卷积感受野计算）

在卷积神经网络中，感受野是衡量网络结构设计的重要参考指标之一，大致可以判断某一层网络能够关联到的原始输入的大小，具体的定义为：卷积神经网络（CNN）每一层输出的特征图（feature map）上的像素点在原始图像上映射的区域大小，示例图如下（也可参阅更多关于感受野计算的资料）：



而在经典的深度强化学习 + 例如 Atari 这样的视频游戏实践中，一般常用如下类型的卷积神经网络，现在请简单计算下图中左半部分给出的神经网络，最后一层卷积的激活值对应的感受野，给出计算过程和最终答案。



(如有兴趣，也可以下计算右半部分，最后一个 Residual Block 的激活值对应的感受野)

题解

1.

首先给出感受野计算的递推公式：

$$l_k = l_{k-1} + [(K_k - 1) \prod_{i=1}^{k-1} s_i]$$

其中 l_k 指的是第 k 层激活值的感受野， K_k 指的是第 k 层的卷积核大小， s_i 指的是第 i 层的步长大小 (stride)。

接下来我们开始进行计算，首先对于第一层而言，感受野显然为第一层的卷积核大小，即：

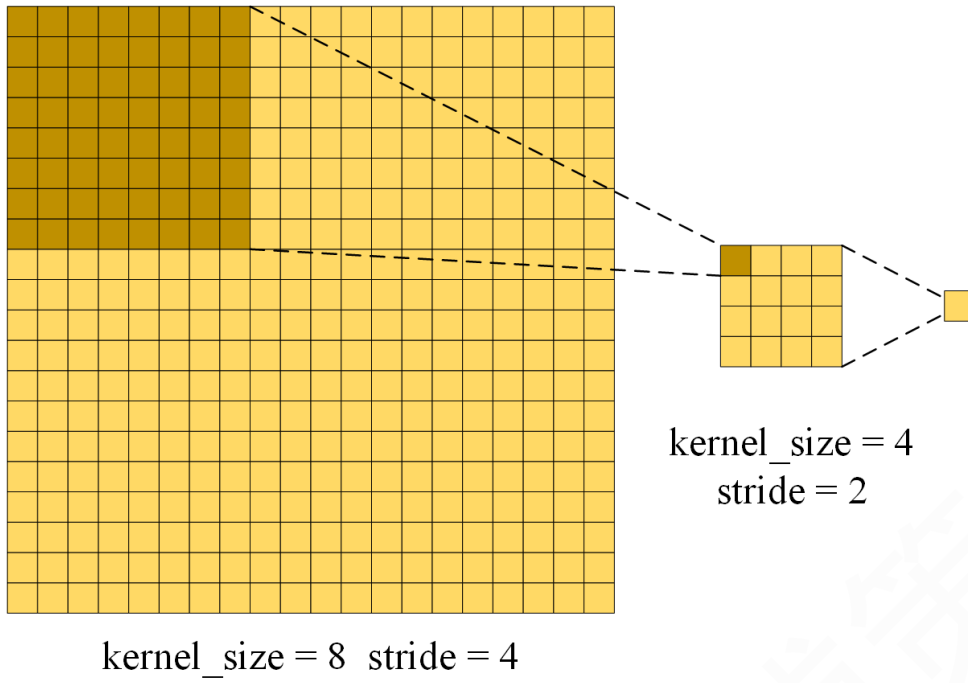
$$l_1 = 8$$

接下来进行递推：

$$l_2 = l_1 + (K_2 - 1) \prod_{i=1}^1 s_i = 20$$

因此最终的答案即为：20.

或者也可用画图的方式，更加直观地理解：



也可看出，第二层卷积的激活值的感受野为 20.

2.

首先，我们可以发现，由于后层卷积激活值的感受野一定大于前层卷积的激活值感受野，因此 Residual Connection 并不会影响对激活值感受野的计算。递推公式仍然成立。

同时，也不难发现，Maxpooling 层具有和卷积层相同的递推公式。

因此，可以进行如下递推：

$$\begin{aligned}
 l_1 &= 3 \\
 l_2 &= l_1 + (K_2 - 1) \prod_{i=1}^1 s_i = 3 + (3 - 1) \times (1) = 5 \\
 l_3 &= l_2 + (K_3 - 1) \prod_{i=1}^2 s_i = 5 + (3 - 1) \times (1 \times 2) = 9 \\
 l_4 &= l_3 + (K_4 - 1) \prod_{i=1}^3 s_i = 9 + (3 - 1) \times (1 \times 2 \times 1) = 13 \\
 l_5 &= l_4 + (K_5 - 1) \prod_{i=1}^4 s_i = 13 + (3 - 1) \times (1 \times 2 \times 1 \times 1) = 17 \\
 l_6 &= l_5 + (K_6 - 1) \prod_{i=1}^5 s_i = 17 + (3 - 1) \times (1 \times 2 \times 1 \times 1 \times 1) = 21
 \end{aligned}$$

综上所述，最后一个残差层输出的激活值感受野为 21.

题目2（表征学习网络中的 LayerNorm）

背景

Layer Normalization [1] 是一种神经网络模型中常用的归一化方法，它由多伦多大学的 Jimmy Lei Ba 等人于2016年提出（其它常用的归一化方法还包括 Batch Normalization [2] 等）。这些归一化方法的目的是将当前神经网络中激活值的数值大小缩放到一个**合适的范围**内，从而让网络的输出始终处于神经网络激活函数的有效数值范围内（例如 tanh 的线性段），并保持网络激活值的数值量级，一定程度上避免使用深层网络时常遇到的梯度爆炸或消失问题，从而让整个模型在训练中更快更好地收敛。

LN 的定义

Layer Normalization 具体实现中的定义为：假设神经网络模型某层的输出激活值为 v ，它的所有元素的均值为 μ ，方差为 σ ， v 中的元素数量为 d ：

$$\mu = \frac{1}{d} \sum_{k=1}^d v_k$$
$$\sigma^2 = \frac{1}{d} \sum_{k=1}^d (v_k - \mu)^2$$

那么可以将 v 归一化为符合正态分布的情况，即：

$$\bar{v} = \frac{v - \mu}{\sigma}$$

并且，将归一化后的结果进行数值缩放和偏置（仿射变换），引入缩放的尺寸参数 γ 与偏置平移的参数 β ，就可以得到 Layer Normalization 后的最终结果：

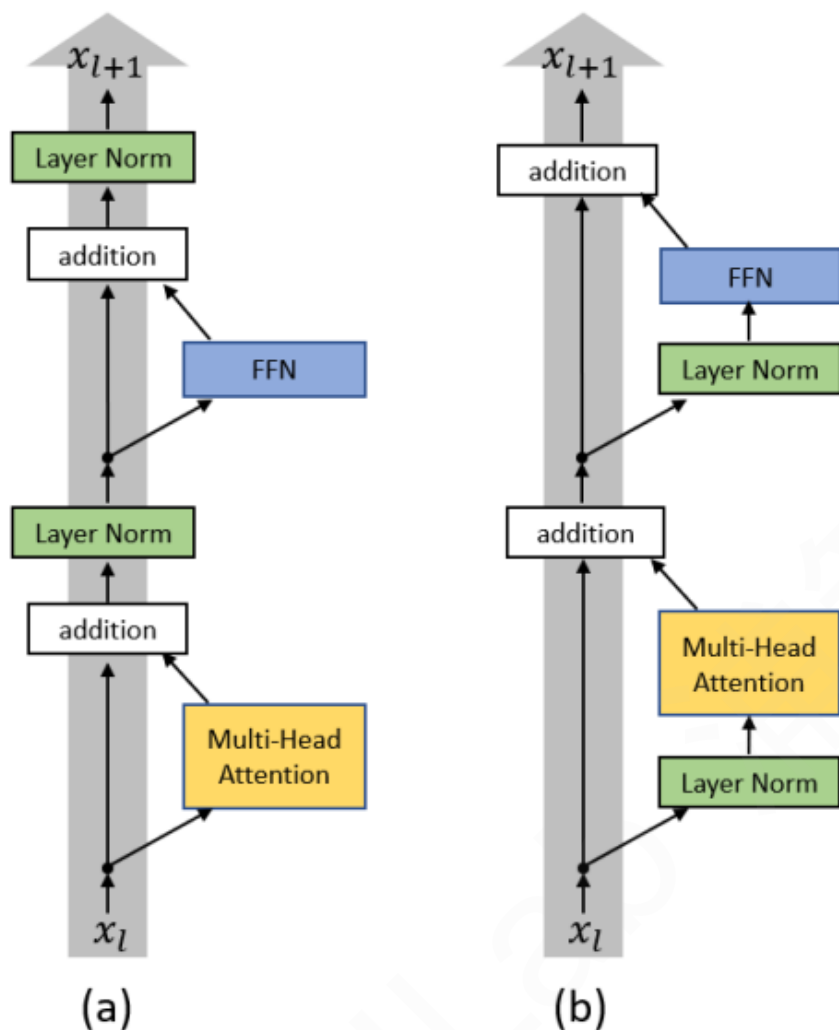
$$\text{LayerNorm}(v) = \gamma \frac{v - \mu}{\sigma} + \beta$$

Tips:

比如在CNN架构的网络中，某层通道数 C ，长宽为 $H \times W$ ，则共计需要对 $d = C \times H \times W$ 这些所有的数值做Layer Normalization。

Post-LN 与 Pre-LN

在具体的神经网络设计，例如 Transformer 结构中，有两种使用 LN 的方式（位置），即 Post-LN 和 Pre-LN，整体设计对比如下图所示（左图 Post-LN，右图 Pre-LN）：



更具体地，对于 L 层 Transformer Layers 构成的模型，我们补充一些数学定义：

- 假设前馈网络的参数 W 从高斯分布 $\mathcal{N}(0, 1/d)$ 中初始化，偏置系数 b 初始化为 0，并将自注意力层的 MultiheadAtt 函数简化为 $\frac{1}{n} \sum_{j=1}^n x_{l,j} W^{V,l}$ ，其中 $W^{V,l}$ 从 $\mathcal{N}(0, 1/d)$ 中采样获得。
- Layer Normalization 的尺寸参数 γ 初始化为 1，偏置平移系数 β 初始化为 0
- 假设输入变量 $x \sim X := \mathcal{N}(0, \sigma^2 I_d)$ 也从相同高斯分布中采样获得
- 假设使用 ReLU 作为激活函数

这个 Transformer 模型在 Post-LN 和 Pre-LN 模式下完整的数据流如下表中所示：

Post-LN Transformer	Pre-LN Transformer
第 l 层，输入 x 的第 i 个元素为 $x_{l,i}^{\text{post}}$	第 l 层，输入 x 的第 i 个元素为 $x_{l,i}^{\text{pre}}$
$x_{l,i}^{\text{post},1} = \text{MultiheadAtt}(x_{l,i}^{\text{post}}, [x_{l,1}^{\text{post}}, \dots, x_{l,n}^{\text{post}}])$	$x_{l,i}^{\text{pre},1} = \text{LayerNorm}(x_{l,i}^{\text{pre}})$
$x_{l,i}^{\text{post},2} = x_{l,i}^{\text{post}} + x_{l,i}^{\text{post},1}$	$x_{l,i}^{\text{pre},2} = \text{MultiheadAtt}(x_{l,i}^{\text{pre},1}, [x_{l,1}^{\text{pre},1}, \dots, x_{l,n}^{\text{pre},1}])$
$x_{l,i}^{\text{post},3} = \text{LayerNorm}(x_{l,i}^{\text{post},2})$	$x_{l,i}^{\text{pre},3} = x_{l,i}^{\text{pre}} + x_{l,i}^{\text{pre},2}$

$x_{l,i}^{\text{post},4} = \text{ReLU}(x_{l,i}^{\text{post},3}W^{1,l} + b^{1,l})W^{2,l} + b^{2,l}$	$x_{l,i}^{\text{pre},4} = \text{LayerNorm}(x_{l,i}^{\text{pre},3})$
$x_{l,i}^{\text{post},5} = x_{l,i}^{\text{post},3} + x_{l,i}^{\text{post},4}$	$x_{l,i}^{\text{pre},5} = \text{ReLU}(x_{l,i}^{\text{pre},3}W^{1,l} + b^{1,l})W^{2,l} + b^{2,l}$
$x_{l,i}^{\text{post}} = \text{LayerNorm}(x_{l,i}^{\text{post},5})$	$x_{l+1,i}^{\text{pre}} = x_{l,i}^{\text{pre},5} + x_{l,i}^{\text{pre},3}$
	末层 LayerNorm: $x_{\text{Final},i}^{\text{pre}} = \text{LayerNorm}(x_{L+1,i}^{\text{pre}})$

根据上述题干信息，请证明 LN 和网络层数与每层元素数量之间的关系：

在模型网络参数初始化时，对于 Post-LN Transformer 的任意第 l 层的第二个残差网络的输出的第 i 个元素， $x_{l,i}^{\text{post},5}$ ，有如下公式成立：

$$\mathbb{E}(\|x_{l,i}^{\text{post},5}\|^2) = \frac{3}{2}d$$

在模型网络参数初始化时，对于 Pre-LN Transformer 的任意第 l 层的输入的第 i 个元素， $x_{l,i}^{\text{pre}}$ ，有如下公式成立：

$$(1 + \frac{l}{2})d \leq \mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) \leq (1 + \frac{3l}{2})d$$

提示：

•

$$\mathbb{E}(\|\text{ReLU}(X)\|^2) = \sum_{i=1}^d \mathbb{E}(\|\text{ReLU}(X_i)\|^2) = \sum_{i=1}^d \mathbb{E}(\|\text{ReLU}(X_i)\|^2 | X_i \geq 0) \mathbb{P}(X_i \geq 0) = \frac{1}{2}d\sigma^2$$

$$\bullet \quad \mathbb{E}(\|\text{LayerNorm}(v)\|^2) = \mathbb{E}(\|\frac{v - \mu}{\sigma}\|^2) = \mathbb{E}(\frac{\sum_{k=1}^d (v_k - \mu)^2}{\sigma^2}) = d$$

- 如果你已经成功证明上述二式，就可以将其用于计算 Transformer 各层梯度的相对大小，可以进一步思考 Post-LN Transformer 与 Pre-LN Transformer 各层梯度的变化趋势，以及这个结论对于神经网络结构和训练方法设计的意义。

题解

1.

根据提示，于是有：

$$\mathbb{E}(\|x_{l,i}^{\text{post}}\|^2) = \mathbb{E}(\|\text{LayerNorm}(v)\|^2) = d$$

$$\begin{aligned}
\mathbb{E}(\|x_{l,i}^{\text{post},2}\|^2) &= \mathbb{E}(\|x_{l,i}^{\text{post}} + x_{l,i}^{\text{post},1}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{post}}\|^2) + \mathbb{E}(\|x_{l,i}^{\text{post},1}\|^2) + 2\mathbb{E}(x_{l,i}^{\text{post}} x_{l,i}^{\text{post},1}) \\
&= \mathbb{E}(\|x_{l,i}^{\text{post}}\|^2) + \mathbb{E}(\|x_{l,i}^{\text{post},1}\|^2) + 0 \\
&= \mathbb{E}(\|x_{l,i}^{\text{post}}\|^2) + \mathbb{E}(\|\frac{1}{n} \sum_{i=1}^n x_{l,i}^{\text{post}}\|^2)
\end{aligned}$$

由于，

$$d = \mathbb{E}(\|x_{l,i}^{\text{post}}\|^2) \geq \mathbb{E}(\|\frac{1}{n} \sum_{i=1}^n x_{l,i}^{\text{post}}\|^2)$$

故有，

$$d \leq \mathbb{E}(\|x_{l,i}^{\text{post},2}\|^2) \leq 2d$$

同理可得，

$$\mathbb{E}(\|x_{l,i}^{\text{post},3}\|^2) = \mathbb{E}(\|\text{LayerNorm}(v)\|^2) = d$$

$$\begin{aligned}
\mathbb{E}(\|x_{l,i}^{\text{post},4}\|^2) &= \mathbb{E}(\|\text{ReLU}(x_{l,i}^{\text{post},3} W^{1,l}) W^{2,l}\|^2) = \mathbb{E}(\mathbb{E}(\mathbb{E}(\|\text{ReLU}(x_{l,i}^{\text{post},3} W^{1,l}) W^{2,l}\|^2 | x_{l,i}^{\text{post},3}, W^{1,l}) | x_{l,i}^{\text{post},3})) \\
&= \mathbb{E}(\mathbb{E}(\|\text{ReLU}(x_{l,i}^{\text{post},3} W^{1,l})\|^2 | x_{l,i}^{\text{post},3})) \\
&= \frac{d}{2}
\end{aligned}$$

$$\mathbb{E}(\|x_{l,i}^{\text{post},5}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{post},3}\|^2) + \mathbb{E}(\|x_{l,i}^{\text{post},4}\|^2) = d + \frac{d}{2} = \frac{3d}{2}$$

2.

$$\mathbb{E}(\|x_{l,i}^{\text{pre},1}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{pre},4}\|^2) = \mathbb{E}(\|\text{LayerNorm}(v)\|^2) = d$$

$$\mathbb{E}(\|x_{l,i}^{\text{pre},3}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) + \mathbb{E}(\|x_{l,i}^{\text{pre},2}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) + \mathbb{E}(\frac{1}{n} \sum_{i=1}^n \|x_{l,i}^{\text{pre},1}\|^2)$$

由于，

$$d = \mathbb{E}(\|x_{l,i}^{\text{pre},1}\|^2) \geq \mathbb{E}(\|\frac{1}{n} \sum_{i=1}^n x_{l,i}^{\text{pre},1}\|^2)$$

故有，

$$\mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) \leq \mathbb{E}(\|x_{l,i}^{\text{pre},3}\|^2) \leq \mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) + d$$

随后计算：

$$\mathbb{E}(\|x_{l+1,i}^{\text{pre}}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{pre},3}\|^2) + \mathbb{E}(\|x_{l,i}^{\text{pre},5}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{pre},3}\|^2) + \mathbb{E}(\frac{1}{2}\|x_{l,i}^{\text{pre},4}\|^2) = \mathbb{E}(\|x_{l,i}^{\text{pre},3}\|^2) + \frac{1}{2}d$$

那么可以得到：

$$\mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) + \frac{1}{2}d \leq \mathbb{E}(\|x_{l+1,i}^{\text{pre}}\|^2) \leq \mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) + \frac{3}{2}d$$

因此，递归该式子，我们有：

$$(1 + \frac{l}{2})d \leq \mathbb{E}(\|x_{l,i}^{\text{pre}}\|^2) \leq (1 + \frac{3l}{2})d$$

3.

根据提示中的不等式，对于 Post-LN Transformer 有，

$$\frac{\partial L}{\partial W^{2,l}} = \frac{\partial L}{\partial x_{L+1}^{\text{post}}} \left(\prod_{j=l+1}^L \frac{\partial x_{j+1}^{\text{post}}}{\partial x_j^{\text{post}}} \right) \frac{\partial x_{l+1}^{\text{post}}}{\partial W^{2,l}} \approx \mathcal{O}((\frac{2}{3})^{(L-l)/2})$$

对于 Pre-LN Transformer 有，

$$\frac{\partial L}{\partial W^{2,l}} = \frac{\partial L}{\partial x_{\text{Final}}^{\text{pre}}} \frac{\partial x_{\text{Final}}^{\text{pre}}}{\partial x_{L+1}^{\text{pre}}} \left(\prod_{j=l+1}^L \frac{\partial x_{j+1}^{\text{pre}}}{\partial x_j^{\text{pre}}} \right) \frac{\partial x_{l+1}^{\text{pre}}}{\partial W^{2,l}} \approx \mathcal{O}(1)$$

这意味着在训练的开始，对于 Post-LN Transformer，其各层参数梯度是指数型增长的，前几层低而后几层高。因此，如果直接使用较大的学习率会很容易导致训练崩溃。为了控制训练过程的稳定性，对于 Post-LN Transformer 我们需要在训练的早期控制学习率在较低水平，虽然这样子会影响训练速度，但可以让训练稳定收敛。对于 Pre-LN Transformer，其各层的参数梯度幅值相似，因此可以使用一个合适的学习率即可。

更多关于 Post-LN 和 Pre-LN 的对比思考可以参考论文 [《On Layer Normalization in the Transformer Architecture》](#)

代码实践题

题目1（奇偶数预测问题）

这道题需要实现“**利用神经网络预测一个数字是奇数还是偶数**”的相关训练和测试代码。具体来说，这是一个使用神经网络进行二分类的经典问题。网络的输入是范围在 $[0, 999]$ 内的整数，网络的输出是这个数字属于奇数还是偶数的类别标签。

这里我们提供了三种实现这个二分类问题的思路，具体代码实现可以自定义：

- 直接将需要预测的数字输入神经网络
- 先此数字转化为二进制编码，然后将所有的二进制位输入网络
- 使用三角函数 (\sin , \cos) 对需要预测的数字进行处理，然后输入网络

请分别实现这三种方案，提交相关代码。并回答在这三种方案中，哪些方案是可行的（即网络可以收敛）；同时思考哪种方案是实践中最优的，给出相关分析？

题解

对于“**利用神经网络预测一个数字是奇数还是偶数**”这个问题，我们给出了三个思路，现在逐一进行提示和解答。最后也将附上实验结果和简要说明。

直接将需要预测的数字输入神经网络

对于这种思路，我们先来明确神经网络的输入和输出维度：

- 输入：一维向量，其中的元素是需要预测的数字
- 输出：二维向量，代表奇偶数的 one-hot 表示

针对输入的设计，我们就可以构造数据集了。根据题目的要求，数据是 $[0, 999]$ 之间的整数，因此我们可以写出下面的代码来构造数据集：

```
1 num_range=(0, 1000)
2 total_data = [(torch.tensor([float(i)]), i % 2) for i in range(*num_range)]
```

在神经网络的设计上，我们简单使用了一个三层 MLP，具体构造如下：

```
1 class NetWork(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.mlp = nn.Sequential(
5             nn.Linear(1, 32),
6             nn.ReLU(),
```

```

7         nn.Linear(32, 32),
8         nn.ReLU(),
9         nn.Linear(32, 2)
10    )
11
12    def forward(self, x):
13        return self.mlp(x)

```

对于后续的每一种情形，我们都将沿用同样的网络架构，就不再重复说明了。

先将此数字转化为二进制编码，然后将所有的二进制位输入网络

对于这种思路，神经网络的输入维度将发生改变。

首先，我们来分析一下新的输入的维度。发现， $1 = 0b1$ 只需要一维向量就可以表示， $3 = 0b11$ 需要二维向量， $4 = 0b100$ 需要三维向量。也就是说，对于不同的数字，输入的维度数可能是变化的，但这种变长输入不能直接用 MLP 处理。为了解决这个问题，我们将输入进行“填充”，把所有的数字均填充到10个比特位，即： $1 = 0b0000000001$ ， $3 = 0b0000000011$ ， $4 = 0b0000000100$ ，以此类推。这就保证了输入维度的统一。

数据集生成代码如下：

```

1 in_bit = 10
2 num_range=(0, 1000)
3 total_data = [(dec2bin(i, in_bit), i % 2) for i in range(*num_range)]

```

其中，核心的 dec2bin 函数实现为：

```

1 def dec2bin(num, in_bit):
2     str_format = bin(num).__repr__()[3:-1]
3
4     res = [float(c) for c in str_format]
5
6     if len(res) > in_bit: # Truncate
7         res = res[-in_bit:]
8     elif len(res) < in_bit: # Pad
9         num = in_bit - len(res)
10        for _ in range(num):
11            res.insert(0, 0.)
12    res = torch.tensor(res).float()
13    return res

```

由于我们输入数据的范围是 $[0, 999]$ ，而 2 的 10 次方为 1024，因此我们在代码中将所有的二进制数字都填充到了 10 位，这就是代码中 `in_bit` 参数的含义。

代码的其他部分基本和第一种思路相同，这里就不赘述了。

使用三角函数 (sin, cos) 对需要预测的数字进行处理，然后输入网络

对于这种思路，我们类似于需要对原始输入做特征工程，加入到输入中。具体而言，我们采用的输入是二维向量：

$$(|\sin(\frac{i\pi}{2})|, |\cos(\frac{i\pi}{2})|)$$

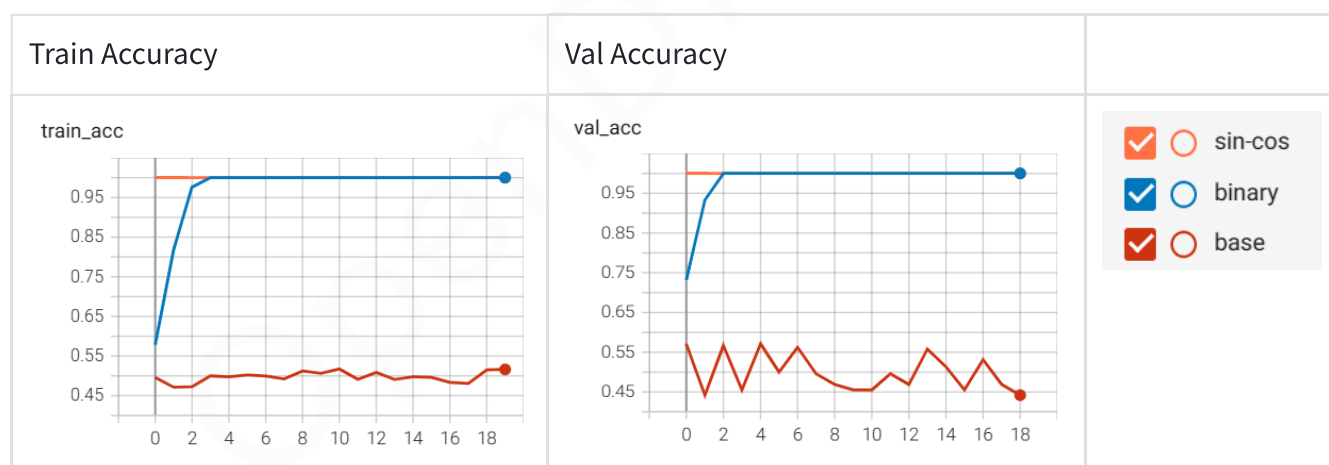
生成数据集的核心代码为：

```
1 from math import sin, cos, pi
2 num_range=(0, 1000)
3 total_data = [(torch.tensor([abs(sin(i * pi / 2)), abs(cos(i * pi / 2))]), i % 2)
```

其余代码部分和前面保持一致。

实验结果和分析

我们在相同的超参数设计下进行实验，得到了以下的结果：



从上图的结果，我们可以看出：

- 基础的将数字直接送入神经网络的实验思路并不收敛，最终 acc 在 0.5 附近震荡
- 使用二进制、三角函数编码的实验均可以收敛，但是使用三角函数的收敛速度更快

简要分析一下内在的原因：

- 对于第二种方法的二进制表示而言，需要判断奇偶的思路是看最后一个二进制位的值。如果是 1 则说明是奇数，如果是 0 则说明是偶数。这种规律是可以被神经网络学习到的，因此最终网络可以收敛。

- 对于第三种方法而言，我们可以找到下面的规律

$i \% 2$	0	1
$ \sin(\frac{i\pi}{2}) $	0	1
$ \cos(\frac{i\pi}{2}) $	1	0
是否是奇数?	no	yes

上述的规律可以被神经网络学习到，因此也可以收敛

- 在实践中，我们发现第三种思路收敛较之第二种更快，这可能是因为第三种思路的输入维度更少，因此网络的训练更加容易，收敛速度也就更快了。

附录

基础实验（以原始数字作为输入）完整代码：

```
1 import torch
2 from torch.utils.data import DataLoader
3 import torch.nn as nn
4 from torch.utils.tensorboard import SummaryWriter
5 import random
6
7 class NetWork(nn.Module):
8     def __init__(self):
9         super().__init__()
10        self.fc = nn.Sequential(
11            nn.Linear(1, 32),
12            nn.ReLU(),
13            nn.Linear(32, 32),
14            nn.ReLU(),
15            nn.Linear(32, 2)
16        )
17
18    def forward(self, x):
19        return self.fc(x)
20
21 def generate_data(num_range=(0, 1000), val_frac=0.2):
22     total_data = [(torch.tensor([float(i)]), i % 2) for i in range(*num_range)]
23     random.shuffle(total_data)
24     val_number = int(len(total_data) * val_frac)
25     train_data = total_data[: -val_number]
26     val_data = total_data[-val_number:]
```

```

27     return train_data, val_data
28
29 if __name__ == '__main__':
30     tb_logger = SummaryWriter('base')
31     train_dataset, val_dataset = generate_data(num_range=(0, 1000), val_frac=0.2)
32     train_loader, val_loader = DataLoader(train_dataset, shuffle=True, batch_size=
33                                     DataLoader(val_dataset, shuffle=True, batch_size=
34
35     net = Network()
36     criterion = nn.CrossEntropyLoss()
37     optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
38
39     for epoch in range(20):
40         # train
41         train_losses = []
42         train_accs = []
43         for x, y in train_loader:
44             pred_y = net(x)
45             pred_res = torch.argmax(pred_y, dim=-1)
46             loss = criterion(pred_y, y)
47
48             # update
49             optimizer.zero_grad()
50             loss.backward()
51             optimizer.step()
52
53             # visualization
54             train_losses.append(loss.item())
55             train_accs.append(torch.sum(pred_res == y) / x.shape[0])
56         train_loss = sum(train_losses) / len(train_losses)
57         train_acc = sum(train_accs) / len(train_accs)
58         tb_logger.add_scalar('train_loss', train_loss, epoch)
59         tb_logger.add_scalar('train_acc', train_acc, epoch)
60         print('Epoch: {}    Train Acc: {}    Train Loss: {}'.format(epoch, train
61
62         # validation
63         val_losses = []
64         val_accs = []
65         with torch.no_grad():
66             for x, y in val_loader:
67                 pred_y = net(x)
68                 pred_res = torch.argmax(pred_y, dim=-1)
69                 loss = criterion(pred_y, y)
70
71                 # visualization
72                 val_losses.append(loss.item())
73                 val_accs.append(torch.sum(pred_res == y) / x.shape[0])

```

```

74     val_loss = sum(val_losses) / len(val_losses)
75     val_acc = sum(val_accs) / len(val_accs)
76     tb_logger.add_scalar('val_loss', val_loss, epoch)
77     tb_logger.add_scalar('val_acc', val_acc, epoch)
78     print('Epoch: {}    Val Acc: {}    Val Loss: {}'.format(epoch, val_acc,
79

```

使用二进制编码作为输入的实验，完整代码：

```

1  import torch
2  from torch.utils.data import DataLoader
3  import torch.nn as nn
4  from torch.utils.tensorboard import SummaryWriter
5  import random
6
7  class NetWork(nn.Module):
8      def __init__(self, in_bit):
9          super().__init__()
10         self.fc = nn.Sequential(
11             nn.Linear(in_bit, 32),
12             nn.ReLU(),
13             nn.Linear(32, 32),
14             nn.ReLU(),
15             nn.Linear(32, 2)
16         )
17
18     def forward(self, x):
19         return self.fc(x)
20
21 def dec2bin(num, in_bit):
22     str_format = bin(num).__repr__()[3:-1]
23
24     res = [float(c) for c in str_format]
25     # Truncate or padding
26     if len(res) > in_bit:
27         res = res[-in_bit:]
28     elif len(res) < in_bit:
29         num = in_bit - len(res)
30         for _ in range(num):
31             res.insert(0, 0.)
32     res = torch.tensor(res).float()
33     return res
34
35 def generate_data(num_range=(0, 1000), val_frac=0.2, in_bit=10):
36     total_data = [(dec2bin(i, in_bit), i % 2) for i in range(*num_range)]

```

```

37     random.shuffle(total_data)
38     val_number = int(len(total_data) * val_frac)
39     train_data = total_data[: -val_number]
40     val_data = total_data[-val_number:]
41     return train_data, val_data
42
43 if __name__ == '__main__':
44     tb_logger = SummaryWriter('binary')
45     input_bit = 10 # 1000 以内能用10个bit表示
46     train_dataset, val_dataset = generate_data(num_range=(0, 1000), val_frac=0.2)
47     train_loader, val_loader = DataLoader(train_dataset, shuffle=True, batch_size=
48                                     DataLoader(val_dataset, shuffle=True, batch_size=
49
50     net = Network(input_bit)
51     criterion = nn.CrossEntropyLoss()
52     optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
53
54     for epoch in range(20):
55         # train
56         train_losses = []
57         train_accs = []
58         for x, y in train_loader:
59             pred_y = net(x)
60             pred_res = torch.argmax(pred_y, dim=-1)
61             loss = criterion(pred_y, y)
62
63             # update
64             optimizer.zero_grad()
65             loss.backward()
66             optimizer.step()
67
68             # visualization
69             train_losses.append(loss.item())
70             train_accs.append(torch.sum(pred_res == y) / x.shape[0])
71         train_loss = sum(train_losses) / len(train_losses)
72         train_acc = sum(train_accs) / len(train_accs)
73         tb_logger.add_scalar('train_loss', train_loss, epoch)
74         tb_logger.add_scalar('train_acc', train_acc, epoch)
75         print('Epoch: {}    Train Acc: {}    Train Loss: {}'.format(epoch, train
76
77         # validation
78         val_losses = []
79         val_accs = []
80         with torch.no_grad():
81             for x, y in val_loader:
82                 pred_y = net(x)
83                 pred_res = torch.argmax(pred_y, dim=-1)

```

```

84         loss = criterion(pred_y, y)
85
86         # visualization
87         val_losses.append(loss.item())
88         val_accs.append(torch.sum(pred_res == y) / x.shape[0])
89     val_loss = sum(val_losses) / len(val_losses)
90     val_acc = sum(val_accs) / len(val_accs)
91     tb_logger.add_scalar('val_loss', val_loss, epoch)
92     tb_logger.add_scalar('val_acc', val_acc, epoch)
93     print('Epoch: {}    Val Acc: {}    Val Loss: {}'.format(epoch, val_acc,
94

```

使用 sin, cos 函数作为网络输入，完整代码：

```

1  import torch
2  from torch.utils.data import DataLoader
3  import torch.nn as nn
4  from torch.utils.tensorboard import SummaryWriter
5  import random
6  from math import sin, cos, pi
7
8  class NetWork(nn.Module):
9      def __init__(self):
10         super().__init__()
11         self.fc = nn.Sequential(
12             nn.Linear(2, 32),
13             nn.ReLU(),
14             nn.Linear(32, 32),
15             nn.ReLU(),
16             nn.Linear(32, 2)
17         )
18
19     def forward(self, x):
20         return self.fc(x)
21
22 def generate_data(num_range=(0, 1000), val_frac=0.2):
23     total_data = [(torch.tensor([abs(sin(i * pi / 2)), abs(cos(i * pi / 2))]), i
24     random.shuffle(total_data)
25     val_number = int(len(total_data) * val_frac)
26     train_data = total_data[: -val_number]
27     val_data = total_data[-val_number:]
28     return train_data, val_data
29
30 if __name__ == '__main__':
31     tb_logger = SummaryWriter('sin-cos')

```



```

32 train_dataset, val_dataset = generate_data(num_range=(0, 1000), val_frac=0.2
33 train_loader, val_loader = DataLoader(train_dataset, shuffle=True, batch_siz
34                                     DataLoader(val_dataset, shuffle=True, batch_size=
35
36 net = NetWork()
37 criterion = nn.CrossEntropyLoss()
38 optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
39
40 for epoch in range(20):
41     # train
42     train_losses = []
43     train_accs = []
44     for x, y in train_loader:
45         pred_y = net(x)
46         pred_res = torch.argmax(pred_y, dim=-1)
47         loss = criterion(pred_y, y)
48
49         # update
50         optimizer.zero_grad()
51         loss.backward()
52         optimizer.step()
53
54         # visualization
55         train_losses.append(loss.item())
56         train_accs.append(torch.sum(pred_res == y) / x.shape[0])
57     train_loss = sum(train_losses) / len(train_losses)
58     train_acc = sum(train_accs) / len(train_accs)
59     tb_logger.add_scalar('train_loss', train_loss, epoch)
60     tb_logger.add_scalar('train_acc', train_acc, epoch)
61     print('Epoch: {}    Train Acc: {}    Train Loss: {}'.format(epoch, train
62
63     # validation
64     val_losses = []
65     val_accs = []
66     with torch.no_grad():
67         for x, y in val_loader:
68             pred_y = net(x)
69             pred_res = torch.argmax(pred_y, dim=-1)
70             loss = criterion(pred_y, y)
71
72             # visualization
73             val_losses.append(loss.item())
74             val_accs.append(torch.sum(pred_res == y) / x.shape[0])
75     val_loss = sum(val_losses) / len(val_losses)
76     val_acc = sum(val_accs) / len(val_accs)
77     tb_logger.add_scalar('val_loss', val_loss, epoch)
78     tb_logger.add_scalar('val_acc', val_acc, epoch)

```

```
79         print('Epoch: {}      Val Acc: {}      Val Loss: {}'.format(epoch, val_acc,  
80
```

题目2（应用实践）


在课程第三讲（表征多模态观察空间）几个应用中任选一个

- 软体机器人（向量观察空间）
- 超级马里奥（图片观察空间）
- 羊了个羊（DI-sheep）（复杂结构化观察空间）

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三讲的[示例 ISSUE](#)。

参考文献

- [1] Ba J L, Kiros J R, Hinton G E. Layer normalization[J]. arXiv preprint arXiv:1607.06450, 2016.
- [2] Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[C]//International conference on machine learning. PMLR, 2015: 448-456.

 如果其他问题请添加官方课程小助手微信（vx: OpenDILab），备注「课程」，小助手将邀请您加入官方课程微信交流群；或发送邮件至 opendilab@pjlab.org.cn