线性层(Linear Layer)

概念与定义

线性层对输入张量执行线性变换 ’公式为 ：

$$y = xW^T + b$$

其中 ’x 是输入张量 ’W 是权重矩阵 ’b 是偏置项 ’y 是输出张量 。

实现与使用

nn.Linear 是 PyTorch 中的一个模块 ’用于实现线性层 。

我们也可以通过代码简单实现一个线性层(对原理不感兴趣的 ’直接跳过这里 ’看“线性层使用”即可) 。

线性层实现

一个简单的线性层实现如下 ：

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class MyLinear(nn.Module):
    def __init__(self,
                 in_features: int,         # 输入特征数
                 out_features: int,        # 输出特征数
                 bias: bool = True):       # 是否使用偏置向量
        super(MyLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features

        # 初始化权重矩阵, 大小为(out_features, in_features), 可训练
        self.weight = nn.Parameter(torch.Tensor(out_features, in_features))

        # 初始化偏置向量, 大小为(out_features, 1), 可训练
        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)

        # 调用重置参数函数, 初始化权重和偏置
        self.reset_parameters()

    # 重置参数函数
    def reset_parameters(self):
        # 使用凯明初始化(均匀分布)初始化权重(权重的取值范围是(-bound,bound), 其中bound = sqrt(6/(f
        # 其中, fan_in 是输入单元的数量. 它是一种专门针对激活函数为 ReLU 或其变体 (如 Leaky ReLU) 的
        nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))

        # 如果有偏置, 使用均匀分布初始化偏置
        if self.bias is not None:
            # 调用了nn.init._calculate_fan_in_and_fan_out函数, 传入self.weight作为参数.
            # 这个函数用于计算神经网络层的接收的输入单元数量和输出单元数量. 但这里只关心输入单元数量, 所
            fan_in, _ = nn.init._calculate_fan_in_and_fan_out(self.weight)
            # 计算偏置的初始化范围边界
            bound = 1 / math.sqrt(fan_in)
```

```
            # 使用均匀分布初始化偏置
            nn.init.uniform_(self.bias, -bound, bound)

    # 前向传播函数
    def forward(self, input):
        # 执行线性变换: y = xA^T + b
        return F.linear(input, self.weight, self.bias)


if __name__ == '__main__':

    torch.manual_seed(0)

    x = torch.arange(4).reshape(2, 2).float()
    print(f'x: {x}')
    # 打印结果:
    # x: tensor([[0., 1.],
    #            [2., 3.]])

    linear = MyLinear(2, 3)
    print(f'weight: {linear.weight}')
    # 打印结果:
    # weight: Parameter containing:
    # tensor([[-0.0053,  0.3793],
    #         [-0.5820, -0.5204],
    #         [-0.2723,  0.1896]], requires_grad=True)
    print(f'bias: {linear.bias}')
    # 打印结果:
    # bias: Parameter containing:
    # tensor([-0.0140,  0.5607, -0.0628], requires_grad=True)

    y = linear(x)
    print(f'y: {y}')
    # 打印结果:
    # y: tensor([[ 0.3653,  0.0403,  0.1269],
    #            [ 1.1134, -2.1645, -0.0386]], grad_fn=<AddmmBackward0>)
```

由上面的实现代码可以看出 ’在线性层中 ’权重和偏置被视为可学习的参数 ’在训练过程中会被优化器更新 。

线性层使用

正如前面所说的 ’nn.Linear 是 PyTorch 中的一个模块 ’可以直接使用 °它接受 3 个参数 ：

- in_features (int): 输入特征的维度(输入张量的最后一维的维度) °
- out_features (int): 输出特征的维度(输出张量的最后一维的维度) °
- bias (bool, optional): 是否使用偏置项 °默认值为 True °如果设置为 False ’则不添加偏置 °

继续沿用上面的例子 ’但将上面的自定义线性层替换为 nn.Linear ：

```python
import torch
import torch.nn as nn

torch.manual_seed(0)

x = torch.arange(4).reshape(2, 2).float()
print(f'x: {x}')
# 打印结果:
# x: tensor([[0., 1.],
#            [2., 3.]])

linear = nn.Linear(in_features=2, out_features=3, bias=True)
print(f'weight: {linear.weight}')
# 打印结果:
# weight: Parameter containing:
# tensor([[-0.0053,  0.3793],
#         [-0.5820, -0.5204],
#         [-0.2723,  0.1896]], requires_grad=True)
print(f'bias: {linear.bias}')
# 打印结果:
# bias: Parameter containing:
# tensor([-0.0140,  0.5607, -0.0628], requires_grad=True)

y = linear(x)
print(f'y: {y}')
# 打印结果:
# y: tensor([[ 0.3653,  0.0403,  0.1269],
#            [ 1.1134, -2.1645, -0.0386]], grad_fn=<AddmmBackward0>)
```

可以看到 ’使用 nn.Linear 模块和使用自定义的线性层在功能上是一致的(计算结果也完全一致) 。

nn.Linear 通常用于构建神经网络的全连接层 。它可以单独使用 ’也可以与其他层﹀如激活函数 `归一化层等﹀组合在一起 。它广泛应用于各种深度学习任务中 ’包括但不限于 ：

- 神经网络的最后几层(如分类任务中的全连接层) 。
- 特征提取后的降维或升维 。
- 自编码器(Autoencoder)的编码器和解码器部分 。

- 多层感知机(MLP)的核心组件 。

作为输出层

为什么不使用 **softmax** 激活函数 ？

在Transformer的解码器中 ，框架设计的最后一层要使用 softmax 激活函数 ，而线性层的输出 $logits$ 作为 softmax 的输入 。但正如我们在2-模型构建中所展示的模型构架一样 ，绝大多数的大语言模型并不会在最后使用 softmax 激活函数 ，而是直接输出 $logits$ 。这样做主要原因有 3 个 ：

(1) softmax 已经是损失函数的一部分 ，因此模型训练时 ，不需要在模型中再添加 softmax 层 。

在语言模型中 ，最常用的损失函数是交叉熵损失(CrossEntropyLoss) 。而在Pytorch和TensorFlow等深度学习框架中 ，交叉熵损失函数内部已经包含了 softmax 操作 。即将线性层 nn.Linear 的输出结果 $logits$ 直接作为损失函数的输入 ，然后在损失函数内部对 $logits$ 进行 softmax 操作后再计算损失值 。

即 ，如果线性层的输出 $logits = z$ ，目标是计算概率分布 p :

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

而交叉熵损失 ：

$$Loss = -\sum_i y_i \log(p_i)$$

这个计算过程在交叉熵损失函数内部优化实现 ，避免了在模型中显式添加 softmax 层 。

(2) 提高计算效率

包含在损失函数中的 softmax 操作通过数值优化技术(Log-Sum-Exp Trick)来避免直接计算指数值 ，从而提升效率并减少数值不稳定的风险 。

(3) 推理时不需要显式计算 softmax

在推理时 ，我们通常只需要知道概率最高的类别 ，而不需要所有类别的概率(完整的概率分布) 。因此 ，在推理时直接使用线性层的输出 logits 并通过 argmax 来选择最高概率的类别即可 ，无需计算 softmax 。

是否权重共享 ？

在 NLP 任务中 ，作为输出层的线性层和嵌入层之间权重共享是一种常见的优化策略 。这种技术被称为“Tied Embeddings”或“Coupled Embeddings” 。其核心思想是将输入的词嵌入矩阵 E 与输出层用于计算 $logits$ 的权重矩阵 W 设置为相同 。即 $W = E^T$ 。

假设输入的文本为 $X_{s \times v}$ 。其中 s 是序列长度 ，v 是词汇表大小 。嵌入矩阵大小为 $E_{v \times d}$ ，其中 v 是词汇表大小 ，d 是嵌入向量维度 。则 nn.Embedding 嵌入层的输出结果为 ：

$$H_{s \times d} = X_{s \times v} \cdot E_{v \times d}$$

而对于 nn.Linear 输出层 ，由于权重共享 ，$W = E^T_{v \times d}$ ，则有 ：

$$Y_{s \times v} = H_{s \times d} \cdot E^T_{v \times d}$$

其中 ，s 是序列长度 ，v 是词汇表大小 。

设置权重共享时 ，由于 nn.Linear 中 X 与 W 相乘的方法是 X 与 W^T 相乘 ，因此 $W = E^T$ 正好满足这一要求 。即直接设置 embeddings.weight = output.weight 即可 。

(1) 权重共享的优点

- 提升参数效率 ：减少了模型参数的数量 ，降低了内存占用和训练成本 。
- 理论一致性 ：嵌入层和输出层之间的映射关系一致 ，理论上可以提高模型的学习效率 。
- 训练稳定性 ：一定程度上缓解梯度消失和梯度爆炸问题 。

(2) 权重共享的缺点

- 表达能力受限 ：嵌入层和输出层的任务不同 。嵌入层学习将词元映射到语义空间 ，而输出层是要根据上下文预测下一个词元的概率分布 。因此 ，共享权重会限制模型的表达能力 。
- 不适合某些任务 ：对于某些任务 ，如分类任务或序列标注任务 ，输出层可能需要更复杂的结构(如多层感知机) ，共享权重此时不合适 。
- 可能导致性能下降 ：在大规模预训练模型中 ，解耦嵌入层和输出层的权重可能会带来更好的效果 。

(3) 选用权重共享的策略

- 模型规模 ：小规模模型共享权重 ，大规模模型解耦权重 。
- 任务需求 ：参数效率高(嵌入式推理)共享权重 ，输出精度高(生成任务)解耦权重 。
- 根据实验调整 ：根据具体任务和模型的表现来决定是否共享权重 。