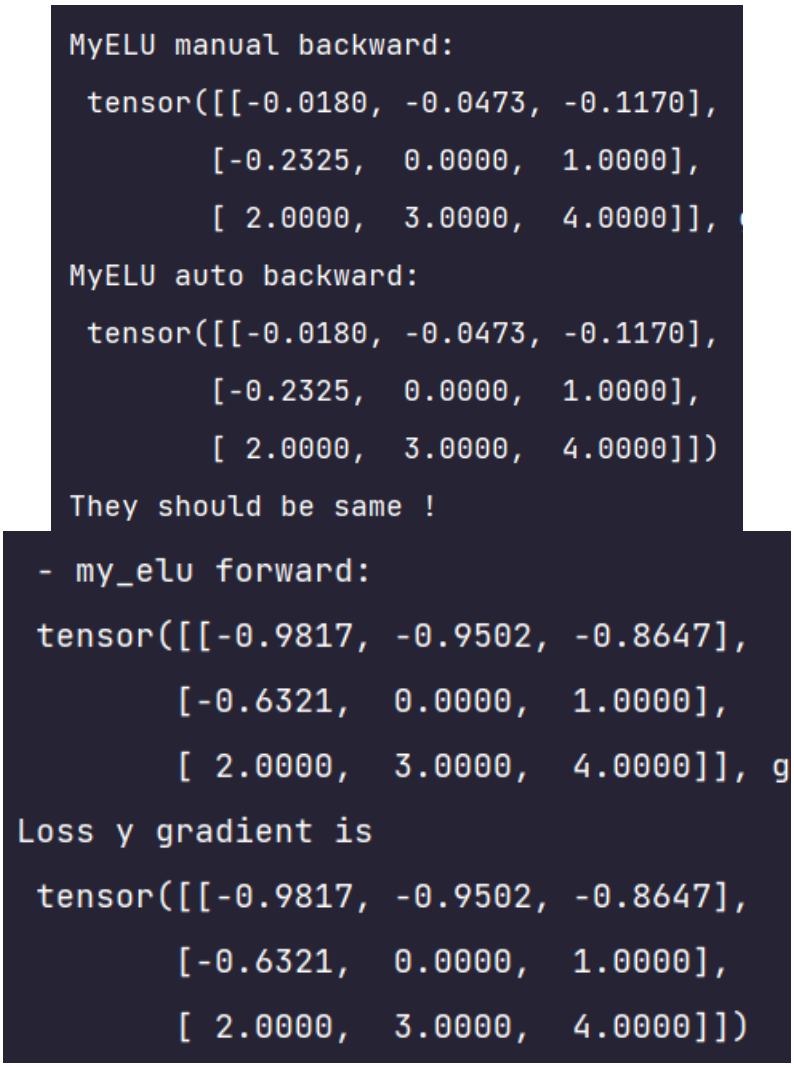
HW-4-陈宸

Q1.ELU 层反向传播实现与验证

1.1 backward_manual()的实现

```
def backward_manual(self, delta_X_top):
    delta_X_bottom = torch.zeros_like(delta_X_top)
    # >0的部分对top的偏导为top
    delta_X_bottom[self.mask] = delta_X_top[self.mask]
    # <=0的部分对top的偏导为alpha*top*e^(top)
    delta_X_bottom[~self.mask] = delta_X_top[~self.mask] * self.alpha * self.X_exp[~self.mask]
    return delta_X_bottom
```

output:



1.2 TODO-Explain行代码的解释

TODO1 in forward():Explain in hw why this is important ?

X_exp计算输入X_bottom每个元素的指数。X_neg计算ELU函数在负值部分的输出。在类中存储这些值可以方便后续反向传播的实现。

TODO2 in backward_manual():re-use the recorded mask in forward() function, why this is important? Explain.

mask作为掩码，用于记录输入X_bottom中大于0的元素位置，在实现反向传播过程中便于计算>0和<0的部分。

TODO3 in main():explain the result, what is dloss/dy

dloss/dy表示损失函数对于输出y值的梯度，在y=elu(x)函数下，损失函数为loss = 0.5*(y-1)^2，对y求导后结果为y-1。

TODO4 in forward():explain the result, calculate the gradient with manual backward function you implemented

结合elu()函数$\frac{dy}{dx}=\begin{cases}1 & if\ x>0\\ \alpha*e^x & if\ x\le 0\end{cases}$则$dx$为:

$$\frac{dLoss}{dx}=\begin{cases}\delta & if\ mask=True\\ \delta*\alpha*e^x & if\ mask=False\end{cases}$$

其中，$\delta=\frac{dLoss}{dy}$

TODO5 in forward():explain the result, use torch autograd to get x's gradient

在这一步中，源代码使用 torch.autograd函数自动反向传播求偏导计算输入x的梯度 dx2。此处的dx2输出值应与backward_mannual中的输出值相同。

TODO6 in forward():explain why dx=dx2, use chain rule to compute, then compare

$$dx(variable)=\frac{dLoss}{dx}=\frac{dLoss}{dy}*\frac{dy}{dx}$$

代入 $\frac{dy}{dx}=\begin{cases}1 & if\ x>0\\ \alpha*e^x & if\ x\le 0\end{cases}$，$\delta=\frac{dLoss}{dy}$即可得到todo4中的式子，利用autograd函数所计算的过程与结果与链式法则也相同。

TODO7 in forward():here we directly use Pytorch ELU. Explain, Should be y==y3? dx==dx3? Explain

y，y3分别为自定义的elu()函数与pytorch内置的elu()函数前向传播的输出值，应为一致。

dx，dx3分别为自定义的backward_mannual()函数与pytorch内置的autograd()函数反向传播输出的梯度值，由于求导法则与变量一致，输出值应为一致。

Q2.Dropout 层反向传播实现与验证

2.1 backward_manual()的实现

```
def backward_manual(self, delta_X_top):
    if self.training:
        delta_X_bottom = delta_X_top * self.mask * self.scale
    else:
        delta_X_bottom = delta_X_top
        return delta_X_bottom
```

output:

```
- dropout forward:
tensor([[ 0.,  0.,  4.],
        [ 0.,  8., 10.],
        [ 0., 14.,  0.]],
Loss y gradient is
 tensor([[ 0.,  0.,  4.],
        [ 0.,  8., 10.],
        [ 0., 14.,  0.]])
```

```
Dropout manual backward:
 tensor([[ 0.,  0.,  8.],
        [ 0., 16., 20.],
        [ 0., 28.,  0.]])
Dropout auto backward:
 tensor([[ 0.,  0.,  8.],
        [ 0., 16., 20.],
        [ 0., 28.,  0.]])
```

2.2 TODO-Explain行代码的解释

TODO1 in forward():Read forward function, explain what dropout does

Dropout的作用是随机将输入的一部分元素设置为零，并对剩余未设置为零的元素进行缩放，根据课堂内容，目的是为防止过拟合，提高模型的泛化能力。结合源代码注释，在训练期间，Dropout操作具体分为以下步骤。

1.构建一个与输入 x 形状相同的二项分布掩码 mask，例如如果 p=0.5，则50%的mask值为1，50%的 mask 值为0。

2.使用该掩码与输入 x 元素相乘，目的是随机部分神经元的输出值被乘以0，视为被丢弃。

3.对结果进行缩放，使得输出的期望值与原输入相同，具体为乘以 `1/(1-p)`。

4.在验证或测试期间，不执行Dropout操作，直接返回输入。

TODO2 in main():explain the result, what is dloss/dy

同Q1一样，在这一步中，利用了autograd.grad函数模拟反向传播，dloss/dy表示损失函数对输出 y 的梯度。假设模型的输出 y = dropout(x)，损失函数为 L = 0.5 * y ^2。根据链式法则，损失函数对输出 y 的梯度 dloss/dy 为：

$$\frac{dLoss}{dy} = \frac{d(0.5 * y^2)}{dy} = y$$

计算得到的 y_diff 就是 $\frac{dLoss}{dy}$。

TODO3 in main():explain why dropout manual backward function you implemented is to compute dy/dx (here use variable dx)

将y_diff作为delta_X_top传入backward_manual函数，通过在前向传播中记录的 mask 和缩放系数 scale，利用链式法则计算每个元素的梯度。对于被丢弃的元素（mask为0），梯度为0；对于未被丢弃的元素（mask为1），梯度乘以 scale，最后的出的值即为x经过dropout的梯度。

TODO4 in main():explain the result, use torch autograd to get x's gradient

使用 torch.autograd计算输入 x 的梯度 dx2，即为$\frac{dy}{dx}$。

TODO5 in main():explain why dx=dx2, use chain rule to explain

利用链式法则，dy/dx 的结果为：

$$dx(variable) = \frac{dLoss}{dy} * \frac{dy}{dx} = \begin{cases} scale & if\ mask = 1 \\ 0 & if\ mask = 0 \end{cases}$$

结合损失函数的梯度，最终得到的 dx 应该与 torch.autograd的计算结果一致，因为两者的计算方法遵循相同的微分规则和公式。