【完成】HW1-P1

2024年8月8日 16:24

前言：

- 要求使用Python3+numpy，不使用任何自动微分工具箱（如PyTorch, TensorFlow, Keras等）。
- 建议先浏览所有问题再尝试解决第一个问题。建议按顺序完成问题，难度逐渐增加，后面的问题通常依赖于前面问题的结果。
- **代码结构**：

```
HW1P1                              // 已包含bonus作业（adam, adamW, dropout）
├─ mytorch
│  ├─ models
│  │  ├─ __init__.py
│  │  └─ mlp.py
│  ├─ nn
│  │  ├─ __init__.py
│  │  ├─ activation.py
│  │  ├─ batchnorm.py
│  │  ├─ dropout.py
│  │  ├─ linear.py
│  │  └─ loss.py
│  ├─ optim
│  │  ├─ __init__.py
│  │  ├─ adam.py
│  │  ├─ adamW.py
│  │  └─ sgd.py
│  └─ __init__.py
├─ test_adam_dropout              // 验证文件
│  ├─ adamW_sol_W.pkl
│  ├─ adamW_sol_b.pkl
│  ├─ adam_sol_W.pkl
│  ├─ adam_sol_b.pkl
│  ├─ dropout_sol_backward.pkl
│  └─ dropout_sol_forward.pkl
├─ HW1P1_S24_Writeup.pdf
├─ S24_HW1_Bonus.pdf              //验证bonus作业代码
├─ bonus_autograder.py
├─ hw1p1_autograder.py            //验证HW1P1作业代码
├─ hw1p1_autograder_flags.py
├─ readme.md
├─ HW1P1.pdf                      // 个人笔记
└─ requirements.txt

验证HW1P1作业代码：python hw1p1_autograder.py
验证bonus作业代码：python bonus_autograder.py
```

-

TEST	STATUS	POINTS	DESCRIPTION
Test 0	PASSED	15	Linear Layer
Test 1	PASSED	10	Activation
Test 2	PASSED	10	MLP0
Test 3	PASSED	10	MLP1
Test 4	PASSED	15	MLP4
Test 5	PASSED	10	Loss
Test 6	PASSED	10	SGD
Test 7	PASSED	20	Batch Norm

-

TEST	STATUS	SCORE	DESCRIPTION
Test 00	PASSED	5.0	Adam
Test 01	PASSED	5.0	AdamW
Test 02	PASSED	5.0	Dropout Forward
Test 03	PASSED	5.0	Dropout Backward

[6.2.3 代码]

0 概览

我们可以将神经网络（NN）视为一个数学函数，它接受输入数据 x 并计算输出 y：

$$y = f_{NN}(x)$$

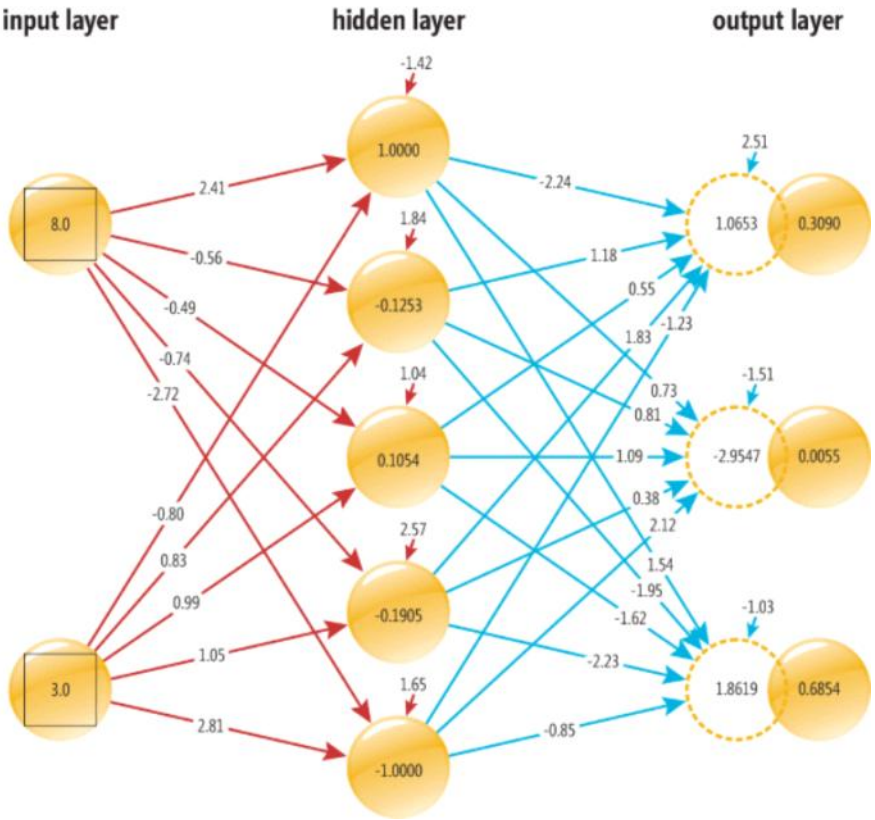
例如，一个训练用来识别垃圾邮件的模型接受一封电子邮件作为输入数据 x，并输出 0 或 1 来指示该邮件是否为垃圾邮件。函数 f_{NN} 有一个特定的形式：它是一个嵌套函数。对于一个输出为标量的三层神经网络，f_{NN} 看起来像这样：

$$y = f_{NN}(x) = f_3(f_2(f_1(x)))$$

在上述方程中，f_1 和 f_2 是以下形式的向量函数：

$$f_l(z) = g_l(W_l \cdot z + b_l)$$

其中 l 为层索引。函数 g_l 为激活函数（例如 ReLU、Sigmoid）。



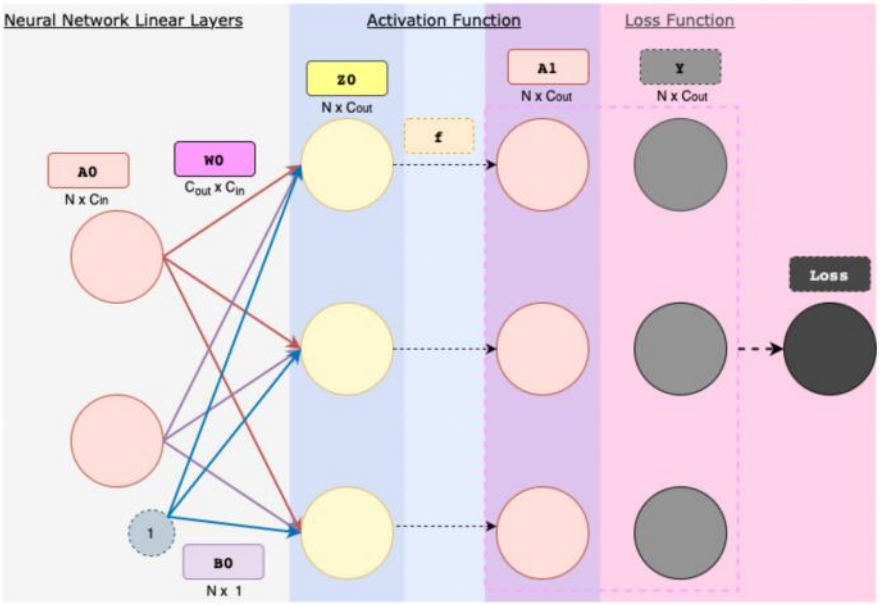


Figure A: End-to-End Topology.

1 全连接层

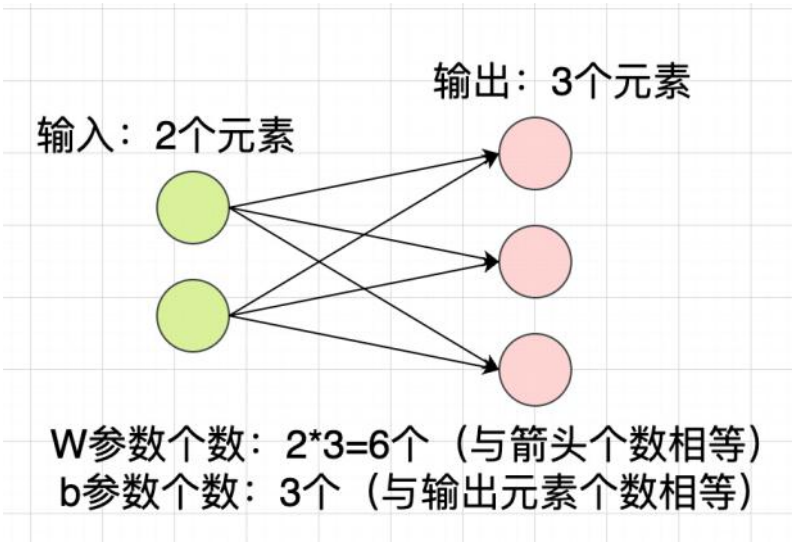
```
class Linear:

    def __init__(self, in_features, out_features):
        self.W    = # TODO
        self.b    = # TODO

    def forward(self, A):
        self.A    = # TODO
        self.N    = # TODO: store the batch size
        Z         = # TODO

        return Z

    def backward(self, dLdZ):
        dLdA      = # TODO
        dLdW      = # TODO
        dLdb      = # TODO
        self.dLdW = dLdW
        self.dLdb = dLdb

        return dLdA
```

Table 1: Linear Layer Components

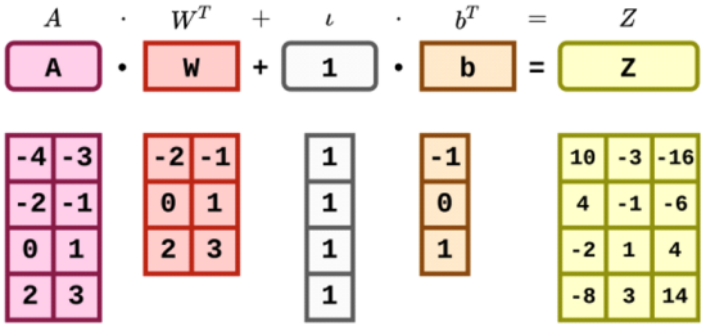
Code Name	Math	Type	Shape	Meaning
N	N	scalar	-	batch size
in_features	C_{in}	scalar	-	number of input features
out_features	C_{out}	scalar	-	number of output features
A	A	matrix	$N \times C_{in}$	batch of N inputs each represented by C_{in} features
Z	Z	matrix	$N \times C_{out}$	batch of N outputs each represented by C_{out} features
W	W	matrix	$C_{out} \times C_{in}$	weight parameters
b	b	matrix	$C_{out} \times 1$	bias parameters
dLdZ	$\partial L/\partial Z$	matrix	$N \times C_{out}$	how changes in outputs affect loss
dLdA	$\partial L/\partial A$	matrix	$N \times C_{in}$	how changes in inputs affect loss
dLdW	$\partial L/\partial W$	matrix	$C_{out} \times C_{in}$	how changes in weights affect loss
dLdb	$\partial L/\partial b$	matrix	$C_{out} \times 1$	how changes in bias affect loss

1.1 前向传播

当batchsize=1时，全连接层前向传播如下：



当 batchsize=4 时，我们用矩阵计算来表达前向传播，如下：



前向传播公式：

$$Z = A \cdot W + \iota_N \cdot b^T$$

- A 是输入数据矩阵。
- W 是权重矩阵。
- b 是偏置向量。
- ι_N 是一个全 1 的 N 维列向量。
- b^T 表示偏置向量 b 的转置，确保能够和 ι_N 相乘进行广播，以便为每个数据点加上偏置。

1.2 反向传播

反向传播的目标是计算损失 L 关于权重矩阵 W、偏置 b 和输入 A 的导数，即 dLdW、dLdb 和 dLdA。

我们可以应用链式法则来计算 A、W、b 的变化如何影响损失 L：

$$\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial Z}\right) \cdot \left(\frac{\partial Z}{\partial A}\right)^T \in \mathbb{R}^{N \times C_{in}}$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \left(\frac{\partial Z}{\partial W}\right)^T \in \mathbb{R}^{C_{out} \times C_{in}}$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \left(\frac{\partial Z}{\partial b}\right)^T \in \mathbb{R}^{C_{out} \times 1}$$

在上述方程中，dZdA、dZdW 和 dZdb 分别代表输入、权重矩阵和偏置如何影响全连接层的输出。

关于dLdZ：

1. 在MLP网络中，如果当前层是输出层，那么 `dLdZ` 就是通过对损失函数求导得到的，它反映了当前网络预测值与真实标签之间的差异如何影响损失值。
2. 如果当前层不是输出层，那么 `dLdZ` 是通过从后面的层传递回来的梯度计算得到的。这些梯度表示损失函数对于该层输出的敏感度，通常是通过链式法则计算的。这些梯度会继续向前传播，以更新该层的权重和偏置。

上面给出的导数方程可以简化为以下形式：（因为 Z=AW+b 中，Z 关于A 的偏导数就是W，其他类似 ）

$$\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial Z}\right) \cdot W \in \mathbb{R}^{N \times C_{in}}$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot A \in \mathbb{R}^{C_{out} \times C_{in}}$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \iota_N \in \mathbb{R}^{C_{out} \times 1}$$

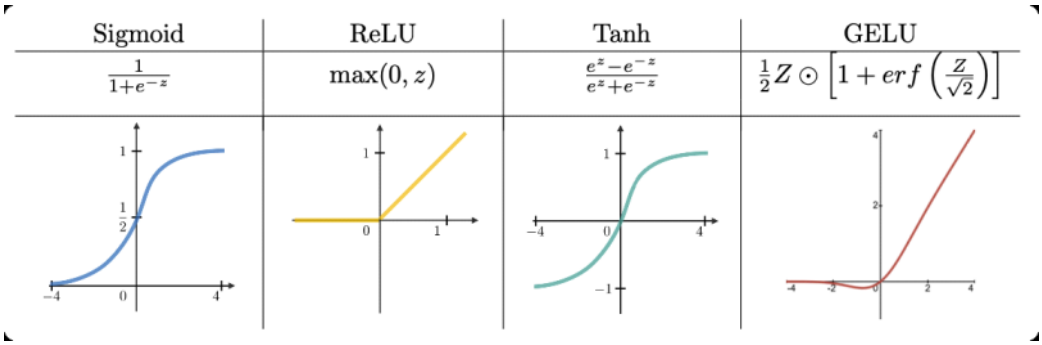
1.3 代码

类的实现如下：

```
import numpy as np

class Linear:
    def __init__(self, in_features, out_features, debug=False):
        """
        初始化权重和偏置为零。
        :param in_features: 输入特征的数量
        :param out_features: 输出特征的数量
        """
        self.W = np.zeros((out_features, in_features))
        self.b = np.zeros((out_features, 1))

    def forward(self, A):
        """
        前向传播
        :param A: 输入层数据，形状为 (N, C0)
        :return: 输出层数据，形状为 (N, C1)
        """
        self.A = A
```

```
        self.N = A.shape[0]
        self.Ones = np.ones((self.N, 1))
        Z = np.dot(A, self.W.T) + np.dot(self.Ones, self.b.T)
        return Z

    def backward(self, dLdZ):
        """
        反向传播
        :param dLdZ: 关于输出的梯度
        :return: 关于输入的梯度
        """
        dLdA = np.dot(dLdZ, self.W)
        self.dLdW = np.dot(dLdZ.T, self.A)
        self.dLdb = np.dot(dLdZ.T, self.Ones)

        return dLdA
```

2 激活函数



下图 Z 是前一个线性层的输出，A 是输入到下一个线性层的输入，设 f_l 为第 L 层的激活函数，则 $A_{l+1} = f_l(Z_l)$。

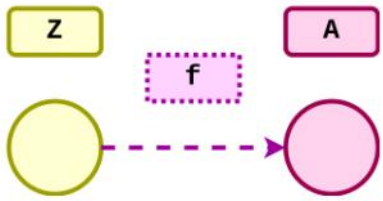
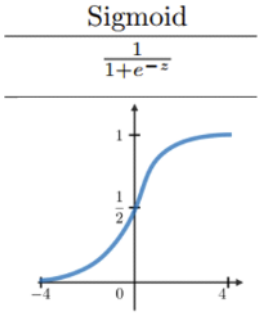


Figure C: Activation Function Topology

2.1 sigmoid



2.1.1 前向传播

$$
\begin{aligned}
A &= \texttt{sigmoid.forward}(Z) \\
&= \varsigma(Z) \\
&= \frac{1}{1+e^{-Z}}
\end{aligned}
$$

f	(	Z	)	=	A
f	(	Z	)	=	A

-4	-3
-2	-1
0	1
2	3

0.01	0.04
0.11	0.26
0.50	0.73
0.88	0.95

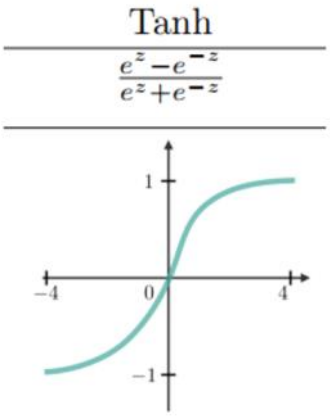
2.1.2 反向传播

$$
\begin{aligned}
\frac{dL}{dZ} &= \texttt{sigmoid.backward}(\texttt{dLdA}) \\
&= \texttt{dLdA} \odot \frac{\partial A}{\partial Z} \\
&= \texttt{dLdA} \odot (\varsigma(Z) - \varsigma^2(Z)) \\
&= \texttt{dLdA} \odot (A - A \odot A)
\end{aligned}
$$

2.1.3 代码

```python
class Sigmoid:
    def forward(self, Z):
        # 缓存当前层的激活输出
        self.A = 1 / (1 + np.exp(-Z))
        return self.A;

    def backward(self, dLdA):
        dLdZ = dLdA * (self.A - self.A * self.A)
        return dLdZ
```

2.2 Tanh



2.2.1 前向传播

$$
\begin{aligned}
A &= \texttt{Tanh.forward}(Z) \\
&= \tanh(Z) \\
&= \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}}
\end{aligned}
$$

f	(	Z	)	=	A
f	(	Z	)	=	A

-4	-3
-2	-1
0	1
2	3

-0.99	-0.99
-0.96	-0.76
0.00	0.76
0.96	0.99

2.2.2 反向传播

首先，根据tanh的前向传播，有：

$$A = tanh(Z)$$

又因为：

$$tanh'(x) = 1 - tanh^2(x)$$

我们可以将 $tanh(x)$ 替换为 A，那么 $tanh'(Z) = 1 - A^2$，即 $\frac{dA}{dZ} = 1 - A^2$

现在，我们可以根据链式法则计算 $\frac{dL}{dZ}$。

$$\frac{dL}{dZ} = \frac{dL}{dA} \cdot \frac{dA}{dZ}$$

我们知道 $\frac{dL}{dA}$ 是 $dLdA$（这是损失 L 关于激活 A 的导数），并且 $\frac{dA}{dZ}$ 就是我们刚刚推导出的 $tanh'(Z)$。

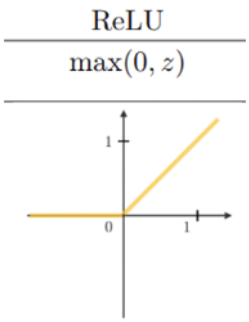
所以，将这些值带入上述公式中，得到：

$$\frac{dL}{dZ} = dLdA \cdot (1 - A^2)$$

2.2.3 代码

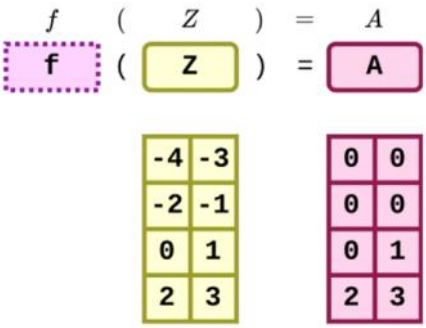
```
class Tanh:
    def forward(self, Z):
        # 缓存当前层的激活输出
        self.A = np.tanh(Z)
        return self.A;

    def backward(self, dLdA):
        dLdZ = dLdA * (1 - np.square(self.A))
        return dLdZ
```

2.3 ReLU



2.3.1 前向传播

$$A = \text{relu.forward}(Z) = max(0, Z)$$



2.3.2 反向传播

ReLU 函数的反向传播需要计算关于其输入 Z 的梯度。这个梯度取决于在前向传播中 Z 的值。

反向传播的计算规则是：

- 如果 Z 大于0，Z 对应的梯度应该传递下去，因此梯度是 1。
- 如果 Z 小于或等于0，Z 对应的梯度被阻断，因此梯度是 0。

所以，ReLU 的反向传播方程可以表示为：

$$\frac{dL}{dZ} = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{if } Z \leq 0 \end{cases}$$

这里，$\frac{dL}{dZ}$ 是损失函数 L 对 Z 的梯度，$dLdA$ 是从后一层传回的梯度。最终，$\frac{dL}{dZ}$ 的值由 Z 的正负决定。

总结：

- **前向传播**：$A = max(0, Z)$
- **反向传播**：

$$\frac{dL}{dZ} = dLdA \times \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{if } Z \leq 0 \end{cases}$$

2.3.3 代码

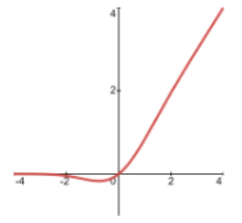
```
class ReLU:
    def forward(self, Z):
        # 如果 Z 中的元素小于0，则取0；如果大于等于0，则取元素本身的值
        self.A = np.maximum(0, Z)
        return self.A

    def backward(self, dLdA):
        # 如果激活值 A 大于 0，梯度为 dLdA；否则，梯度为 0
        dLdZ = np.where(self.A > 0, dLdA, 0)
        return dLdZ
```

因为Z 的维度[N, C]，代表 N 个样本，每个样本又有 C 个特征，**需要对Z中每一个元素与0进行max比较**，所以需要用np.maximum()函数。

2.4 GELU

GELU (Gaussian Error Linear Unit)：高斯误差线性单元。**在transformer、bert、GPT中用的很多。**

GELU
$z\Phi(z)$



2.4.1 前向传播

GELU激活函数根据标准高斯分布的累积分布函数Φ(Z) = P(X ≤ Z)来定义，其中X~N(0,1)：

$$\begin{aligned} A &= \texttt{gelu.forward}(Z) \\ &= Z\Phi(Z) \\ &= Z\int_{-\infty}^{Z} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx \\ &= \frac{1}{2}Z\left[1 + erf\left(\frac{Z}{\sqrt{2}}\right)\right] \end{aligned}$$

这里，`erf`指的是误差函数，经常出现在概率和统计学中。它也可以处理复数参数，但在这里我们只取实数参数。

$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$

python的 math 和 scipy 库有其实现。

2.4.2 反向传播

先计算关于Z的导数：

$$\begin{aligned}\frac{dA}{dZ} &= \frac{d}{dZ} Z\Phi(Z) \\ &= \Phi(Z) + Z\Phi'(Z) \\ &= \Phi(Z) + Z\mathbb{P}(X = Z) \\ &= \frac{1}{2}\left[1 + erf\left(\frac{Z}{\sqrt{2}}\right)\right] + \frac{Z}{\sqrt{2\pi}} \exp\left(-\frac{Z^2}{2}\right)\end{aligned}$$

所以反向传播函数的最终表达式：

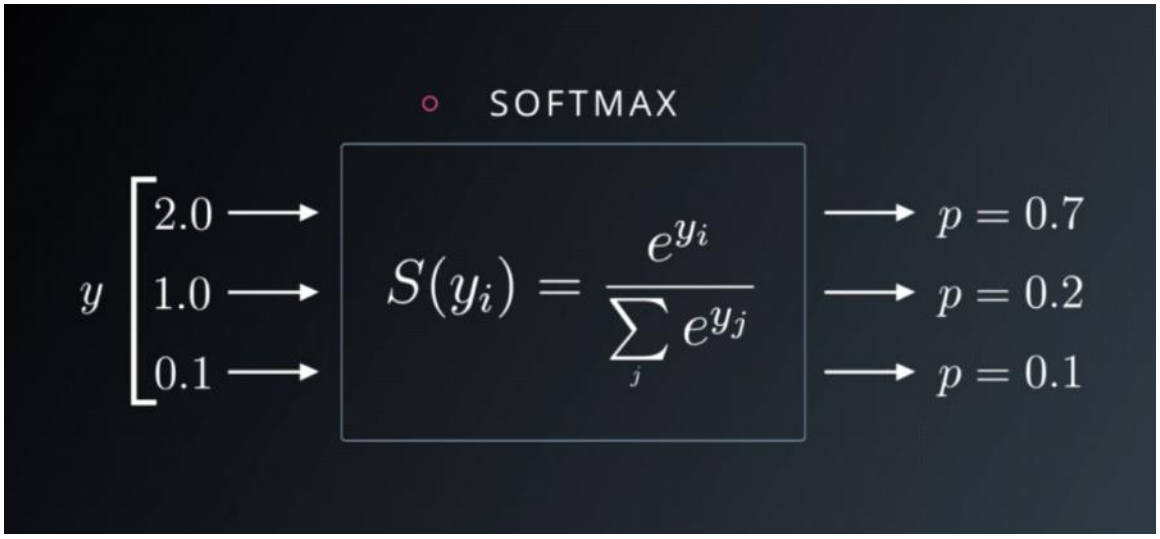
$$\begin{aligned}\frac{\partial L}{\partial Z} &= \texttt{gelu.backward(dLdA)} \\ &= \texttt{dLdA} \odot \frac{\partial A}{\partial Z} \\ &= \texttt{dLdA} \odot \left[\frac{1}{2}\left(1 + erf\left(\frac{Z}{\sqrt{2}}\right)\right) + \frac{Z}{\sqrt{2\pi}} \exp\left(-\frac{Z^2}{2}\right)\right]\end{aligned}$$

2.4.3 代码

```
import numpy as np
from scipy.special import erf

class GELU:
    def forward(self, Z):
        # 缓存Z以便在反向传播中使用
        self.Z = Z
        # 缓存当前层的激活输出
        self.A = 0.5 * Z * (1 + erf(Z / np.sqrt(2)))
        return self.A;

    def backward(self, dLdA):
        dAdZ = 0.5 * (1 + erf(self.Z / np.sqrt(2))) + self.Z * np.exp(-
np.square(self.Z) / 2) / (np.sqrt(2 * np.pi))
        dLdZ = dLdA * dAdZ
        return dLdZ
```

2.5 softmax

对于深度学习多分类问题，神经网络的最后一层常用softmax，将输出层所有元素转变为0~1的概率值。



2.5.1 前向传播

给定一个向量 $\mathbf{z} = [z_1, z_2, \ldots, z_C]$，Softmax函数定义为：

$$a_m = \frac{e^{z_m}}{\sum_{k=1}^{C} e^{z_k}}$$

- a_m 是Softmax函数的第 m 个输出。
- C 是输入元素的个数。

关于溢出：

由于指数运算 e^x 当 x 值较大时会迅速增长，这可能导致数值溢出。为了避免这种溢出，一个常用的技巧是在进行指数运算前从输入的每个分量中减去最大分量的值，即**规范化**（normalization），它保证了所有的指数项的值都不会大于1，从而确保计算过程的数值稳定性。

为什么可以这么做：因为分子分母同乘以e^c常数，不改变结果。

$$(1) softmax(X + c) = softmax(X)$$

这里X是向量，c是一个常数。下面证明左右两边的每一个分量相等。

$$\begin{aligned} 证：softmax(X + c)_i &= \frac{e^{x_i + c}}{\sum_k e^{x_k + c}} \\ &= \frac{e^{x_i} \cdot e^c}{\sum_k e^{x_k} \cdot e^c} \\ &= \frac{e^{x_i}}{\sum_k e^{x_k}} \\ &= softmax(X)_i \end{aligned}$$

2.5.2 反向传播

在反向传播过程中，我们的目标是计算损失函数 L 关于输入 Z 的梯度 $\frac{\partial L}{\partial Z}$。这需要通过链式法则，结合输出 A 对 Z 的导数（即雅可比矩阵 J）来完成。

计算雅可比矩阵 J：

Softmax 的雅可比矩阵 J 中的元素 J_{mn} 定义为输出 a_m 对输入 z_n 的偏导数：

$$J_{mn} = \frac{\partial a_m}{\partial z_n} = \begin{cases} a_m(1 - a_m) & \text{if } m = n \\ -a_m a_n & \text{if } m \neq n \end{cases}$$

推导：

- 当 $m = n$ 时，使用商的法则对 a_m 求导，我们得到 $a_m(1 - a_m)$。
- 当 $m \neq n$ 时，z_n 只通过分母影响 a_m，计算得到 $-a_m a_n$。

损失函数对输入的梯度：

设 $dLdA = [\frac{\partial L}{\partial a_1}, \ldots, \frac{\partial L}{\partial a_C}]$ 是损失函数 L 关于输出 A 的梯度。那么，损失函数关于输入 Z 的梯度可以通过矩阵乘法 $dLdA$ 和雅可比矩阵 J 得到：

$$dLdZ = dLdA \cdot J$$

这个乘积 $dLdA \cdot J$ 将计算得到 $\frac{\partial L}{\partial Z}$，是一个向量，其中的每个元素是损失函数关于相应 z_i 的梯度。

备注：关于Jmn的推导过程

假设我们的 Softmax 函数输出 a_m 是这样计算的：

$$a_m = \frac{\exp(z_m)}{\sum_{k=1}^{C} \exp(z_k)}$$

我们想计算 $\frac{\partial a_m}{\partial z_n}$。根据定义，我们有两种情况：当 $m = n$ 和 $m \neq n$。

情况 1: 当 $m = n$

当我们对 a_m 相对于其自身的输入 z_m 进行求导时：

$$\frac{\partial a_m}{\partial z_m} = \frac{\partial}{\partial z_m} \left(\frac{\exp(z_m)}{\sum_{k=1}^{C} \exp(z_k)} \right)$$

应用商规则，得：

$$\frac{\partial a_m}{\partial z_m} = \frac{\exp(z_m) \sum_{k=1}^{C} \exp(z_k) - \exp(z_m) \exp(z_m)}{(\sum_{k=1}^{C} \exp(z_k))^2}$$

简化后为：

$$\frac{\partial a_m}{\partial z_m} = \frac{\exp(z_m)}{\sum_{k=1}^{C} \exp(z_k)} \left(1 - \frac{\exp(z_m)}{\sum_{k=1}^{C} \exp(z_k)} \right) = a_m(1 - a_m)$$

情况 2: 当 $m \neq n$

当 $m \neq n$ 时，我们需要计算 a_m 关于不同输入 z_n 的导数：

$$\frac{\partial a_m}{\partial z_n} = \frac{\partial}{\partial z_n} \left(\frac{\exp(z_m)}{\sum_{k=1}^{C} \exp(z_k)} \right)$$

由于 z_n 仅出现在分母中，我们得到：

$$\frac{\partial a_m}{\partial z_n} = \frac{-\exp(z_m) \exp(z_n)}{(\sum_{k=1}^{C} \exp(z_k))^2} = -a_m a_n$$

总结

所以，雅可比矩阵 J 的元素 J_{mn} 定义为：

$$\begin{cases} a_m(1 - a_m) & \text{if } m = n \\ -a_m a_n & \text{if } m \neq n \end{cases}$$

5.5.3 numpy实现

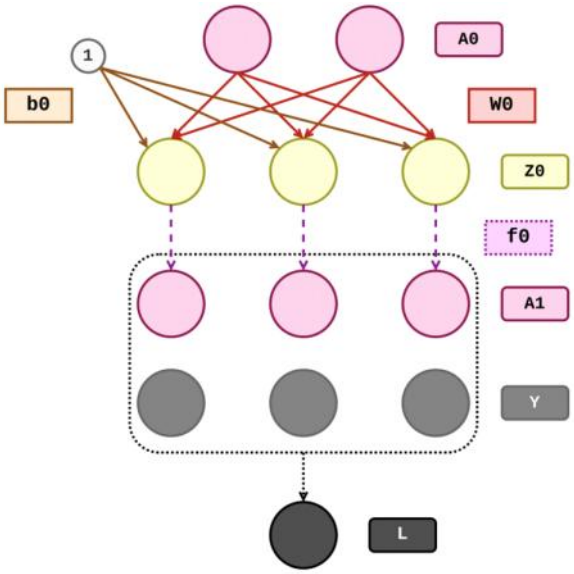
```python
import numpy as np

class Softmax:
    def forward(self, Z):
        """
        参数:
        Z (np.array): 输入数据，形状为 (N, C)，其中 N 是批量大小，C是类别数。

        返回:
        self.A (np.array): 应用Softmax后的输出概率。
        """
```

```
        e_Z = np.exp(Z - np.max(Z, axis=1, keepdims=True))
        self.A = e_Z / np.sum(e_Z, axis=1, keepdims=True)
        return self.A

    def backward(self, dLdA):
        """
        参数:
        dLdA (np.array): 损失函数关于Softmax输出的梯度，形状为 (N, C)。

        返回:
        dLdZ (np.array): 损失函数关于Softmax层输入的梯度，形状为 (N, C)。
        """
        N, C = dLdA.shape
        dLdZ = np.zeros_like(dLdA) # 初始化最终输出的梯度矩阵dLdZ

        # 对每一个样本逐一处理
        for i in range(N):
            a = self.A[i, :]      # 提取第i个样本的Softmax输出
            J = np.zeros((C, C))  # 初始化Jacobian矩阵为零矩阵

            # 根据上述条件填充Jacobian矩阵
            for m in range(C):
                for n in range(C):
                    if m == n:
                        J[m, n] = a[m] * (1 - a[m])
                    else:
                        J[m, n] = -a[m] * a[n]

            # 计算损失函数关于第i个输入的导数
            dLdZ[i, :] = np.dot(dLdA[i, :], J)

        return dLdZ

# 示例使用
softmax = Softmax()
Z = np.array([[1.0, 2.0, 3.0], [1.0, 2.0, 3.0]])
A = softmax.forward(Z)                                  # 前向传播
dLdA = np.array([[0.1, 0.2, 0.7], [0.1, 0.2, 0.7]])  # 损失函数关于输出的梯度
dLdZ = softmax.backward(dLdA)                           # 反向传播

print("Softmax输出 A:", A)
print("损失函数关于Softmax层输入的梯度 dLdZ:", dLdZ)


Softmax输出 A: [[0.09003057 0.24472847 0.66524096]
                [0.09003057 0.24472847 0.66524096]]
损失函数关于Softmax层输入的梯度 dLdZ: [[-0.03813852 -0.0791984   0.11733692]
                                        [-0.03813852 -0.0791984   0.11733692]]
```

3 MLP（全连接层+激活函数）

3.1 MLP（隐藏层 = 0）

下图输入2个元素，单层神经元3个，Y是标签，L是标签Y和输出层A1之间的损失函数值。



前向传播:

$$Z_0 = \texttt{layer0.forward}(A_0) \qquad \in \mathbb{R}^{N \times C_1}$$
$$A_1 = \texttt{f0.forward}(Z_0) \qquad \in \mathbb{R}^{N \times C_1}$$

反向传播:

$$\frac{\partial L}{\partial Z_0} = \texttt{f0.backward}\left(\frac{\partial L}{\partial A_1}\right) \qquad \in \mathbb{R}^{N \times C_1}$$
$$\frac{\partial L}{\partial A_0} = \texttt{layer0.backward}\left(\frac{\partial L}{\partial Z_0}\right) \qquad \in \mathbb{R}^{N \times C_0}$$

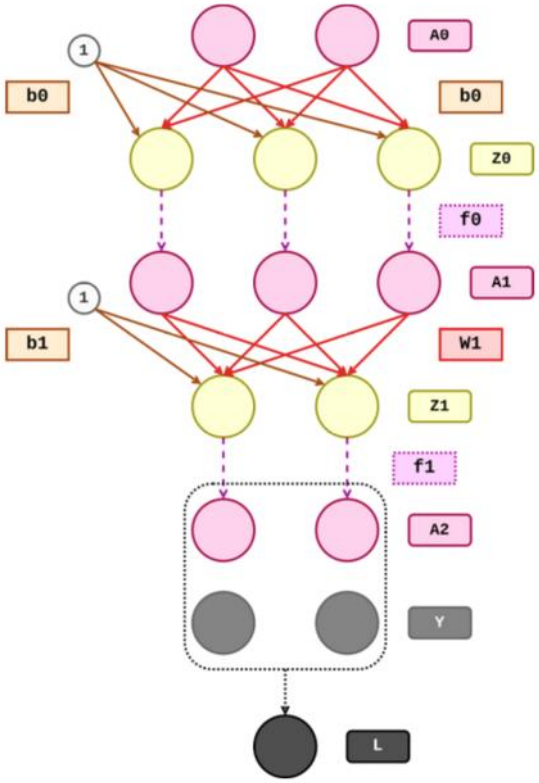
符号表示如下:

- A_0 是网络的原始输入。
- W_0 是第一个线性层的权重。
- Z_0 是应用了 W_0 之后的结果，也就是线性变换后的值。
- f_0 是激活函数。
- A_1 是输出层的激活值，也是模型的最终输出。
- Y 是标签值，用于训练过程中的损失函数计算。
- L 是损失函数的值。
- dL/dA_1、dL/dZ_0 和 dL/dA_0 分别是关于 A_1、Z_0 和 A_0 的损失函数的梯度。

MLP的实现可以直接借用之前全连接层和激励函数的类。

```python
import numpy as np
from mytorch.nn.linear import Linear
from mytorch.nn.activation import ReLU

class MLP0:
    def __init__(self, debug=False):
        """
        初始化一个形状为 (2,3) 的单个线性层。
        使用ReLU激活函数。

        参数:
        - debug: 是否在前向和反向传播中保留中间结果以便于调试。
        """
        # 定义一个包含一个线性层和一个ReLU激活层的列表
        self.layers = [Linear(2, 3), ReLU()]

    def forward(self, A0):
        """
        参数:
        - A0: 输入数据

        返回:
        - A1: 模型输出
        """
        Z0 = self.layers[0].forward(A0)  # 线性层的前向传播
        A1 = self.layers[1].forward(Z0)  # 激活层的前向传播

        return A1  # 返回模型的输出

    def backward(self, dLdA1):
        """
        参数:
        - dLdA1: 损失函数对模型输出的梯度

        返回:
        - dLdA0: 损失函数对模型输入的梯度
        """
        dLdZ0 = self.layers[1].backward(dLdA1)  # 激活层的反向传播
        dLdA0 = self.layers[0].backward(dLdZ0)  # 线性层的反向传播

        return dLdA0  # 返回损失函数对模型输入的梯度
```

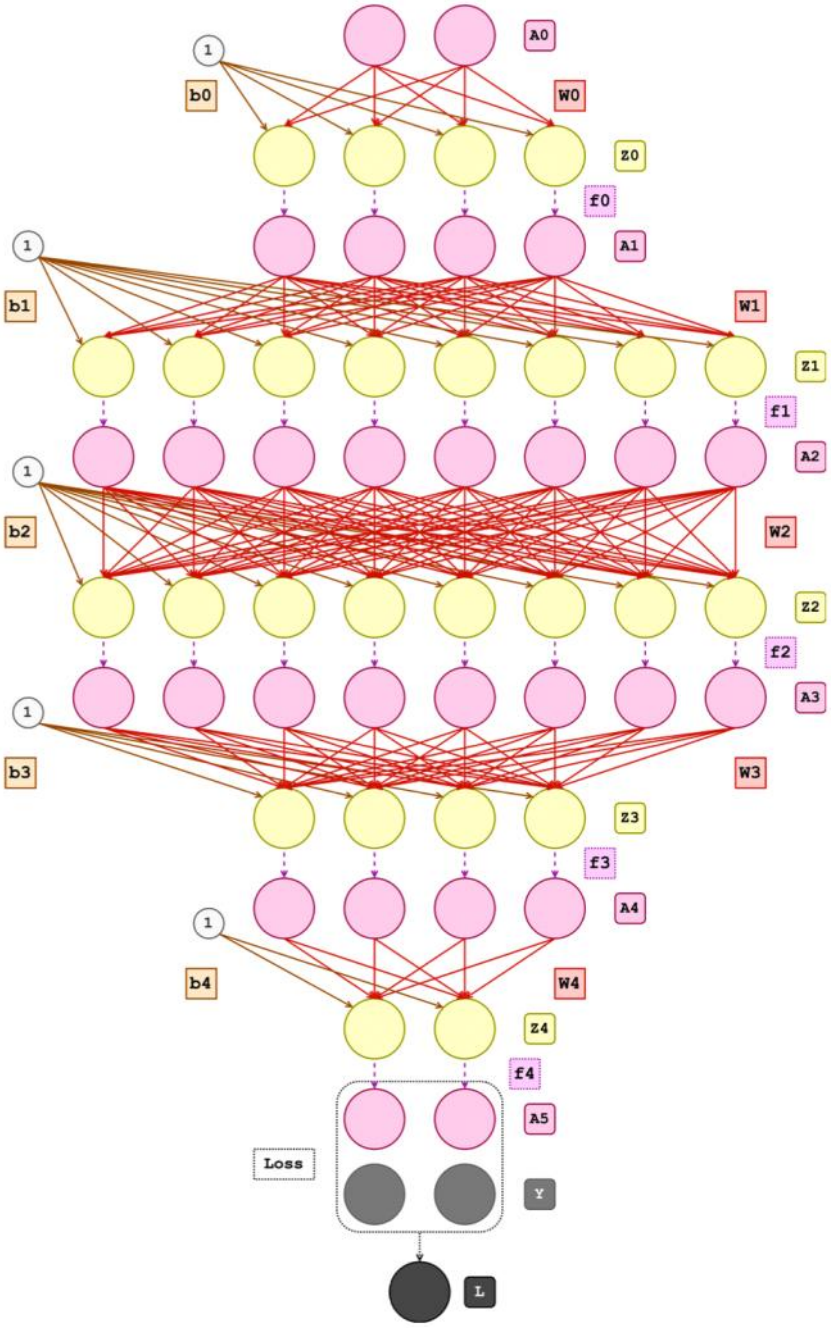
3.2 MLP（隐藏层 = 1）




```python
import numpy as np
from mytorch.nn.linear import Linear
from mytorch.nn.activation import ReLU

class MLP1:
    def __init__(self, debug=False):
        """
        初始化包含两个线性层的网络。第一层的形状为 (2,3)，第二层的形状为 (3,2)。
        两个线性层后都使用ReLU激活函数。
        类似于MLP0，将所有层按顺序存储在列表中。

        参数:
        - debug: 是否在前向和反向传播中保留中间结果以便于调试。
        """
        # 定义线性层和激活层的列表
        self.layers = [Linear(2, 3), ReLU(), Linear(3, 2), ReLU()]

    def forward(self, A0):
        """
        参数:
        - A0: 输入数据

        返回:
        - A2: 模型的输出
        """
        Z0 = self.layers[0].forward(A0)  # 第一个线性层的前向传播
        A1 = self.layers[1].forward(Z0)  # 第一个激活层的前向传播

        Z1 = self.layers[2].forward(A1)  # 第二个线性层的前向传播
```

```
        A2 = self.layers[3].forward(Z1)  # 第二个激活层的前向传播

        return A2  # 返回模型的输出

    def backward(self, dLdA2):
        """
        参数:
        - dLdA2: 损失函数对模型最终输出的梯度

        返回:
        - dLdA0: 损失函数对模型输入的梯度
        """
        # 反向传播激活层和线性层
        dLdZ1 = self.layers[3].backward(dLdA2)  # 第二个激活层的反向传播
        dLdA1 = self.layers[2].backward(dLdZ1)  # 第二个线性层的反向传播

        dLdZ0 = self.layers[1].backward(dLdA1)  # 第一个激活层的反向传播
        dLdA0 = self.layers[0].backward(dLdZ0)  # 第一个线性层的反向传播

        return dLdA0  # 返回损失函数对模型输入的梯度
```

3.3 MLP（隐藏层 = 4）




```python
import numpy as np
from mytorch.nn.linear import Linear
from mytorch.nn.activation import ReLU

class MLP4:
    def __init__(self, debug=False):
        self.layers = [
            Linear(2, 4), ReLU(),
            Linear(4, 8), ReLU(),
            Linear(8, 8), ReLU(),
            Linear(8, 4), ReLU(),
            Linear(4, 2), ReLU()
        ]

    def forward(self, A):
        # 逐层通过网络前向传播
        for layer in self.layers:
```

```
            A = layer.forward(A)  # 应用当前层的前向传播
        return A  # 返回模型的输出

    def backward(self, dLdA):
        # 逐层通过网络反向传播
        for layer in reversed(self.layers):
            dLdA = layer.backward(dLdA)  # 应用当前层的反向传播
        return dLdA  # 返回损失函数对模型输入的梯度
```

4 损失函数

4.1 MSE loss（均方误差损失函数）

均方误差（Mean Squared Error，简称MSE）是衡量模型预测值和实际观测值之间差异的一种常用方法。它在回归问题中特别常见。

回归问题是指模型预测连续值的任务，例如预测房价、温度、销售额等。

均方误差的公式：

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$$

- N是样本数量。
- Y_i是第i个观测值。
- $\hat{Y}_i$ 是第 i 个预测值。

在这个公式中，Y_i 和 $\hat{Y}_i$之间的差异被称为误差（error）。我们对每个样本的误差进行平方，这样做有几个理由：

- 平方确保了所有的误差都是正值。
- 平方放大了较大误差的影响，这意味着较大的误差对总MSE的贡献更大。

总MSE是所有单个平方误差的平均值，给出了关于模型预测精度的整体度量。值越小，表示模型的预测值与实际值之间的差异越小，模型的性能越好。

在神经网络的训练过程中，MSE也常用作损失函数，用于优化模型参数。通过最小化MSE，可以使模型预测的结果更加接近真实值。在训练神经网络时，MSE损失会计算每个输出节点的误差，将它们平方，然后计算平均值。然后，通过反向传播算法来调整模型权重，以减少损失值。

MSE优点：

- **易于理解和实现**：MSE提供了一个清晰的度量，直观地表示了预测值与实际值之间的平均平方差距。
- **处罚较大的误差**：由于误差项是平方的，这意味着较大的误差对总损失的贡献比小误差大得多，这有助于模型识别

并纠正那些有着较大误差的预测。

- **可微性**：MSE损失函数是可微的，这意味着可以用标准的梯度下降法等优化算法来最小化它，这是训练大多数类型的神经网络的基础。

MSE缺点：

- **对异常值敏感**：由于平方项的影响，MSE对异常值非常敏感。这意味着，如果数据中存在离群点，它们将对总损失有很大影响，有时会导致模型偏向这些异常值。
- **可能导致过拟合**：在某些情况下，尤其是当模型非常复杂时，最小化MSE可能会导致过拟合，即模型在训练数据上表现很好，但在未见过的数据上表现不佳。

在实践中，可能会使用MSE的变体或其他损失函数来解决这些问题，例如均方根误差（RMSE）或平均绝对误差（MAE），或者在损失函数中添加正则化项来减轻过拟合的风险。

4.1.1 前向传播

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2$$

- N 是样本数量。
- Y_i是第i个观测值。
- $\hat{Y}_i$ 是第 i 个预测值。

4.1.2 反向传播

反向传播需要计算损失函数相对于模型输出 A 的梯度，即 $\frac{\partial MSE}{\partial A}$。

根据均方误差损失函数公式，以及链式法则，对 A 的每个元素 A_{ij} 计算偏导数得（i 可理解为样本在数据集中的索引，j 可以理解为类别的索引）：

$$\frac{\partial MSE}{\partial A_{ij}} = \frac{2}{N \times C}(A_{ij} - Y_{ij})$$

C 是每个样本的输出类别数量或特征数量。

这里我们利用了以下导数规则：

- $(x^2)' = 2x$ 的导数是 $2x$。
- 平均值的导数是每个项的导数除以项的数量。

因此，对于整个输出矩阵 A，其梯度 $dLdA$ 为：

$$dLdA = \frac{2}{N \times C}(A - Y)$$

4.1.3 代码

```
import numpy as np

class MSELoss:
    def forward(self, A, Y):
```

```
        """
        Calculate the Mean Squared Error (MSE) loss
        :param A: Output of the model of shape (N, C)
        :param Y: Ground-truth values of shape (N, C)
        :Return: MSE Loss (scalar)
        """
        self.A = A
        self.Y = Y
        self.N, self.C = A.shape

        se = (A - Y) ** 2               # 计算平方误差
        sse = np.sum(se)                # 计算所有平方误差的和
        mse = sse / (self.N * self.C)   # 计算均方误差

        return mse

    def backward(self):
        """
        Compute the gradient of the loss with respect to the model output A
        :Return: Gradient of the loss with respect to A
        """

        # 计算MSE损失对A的梯度
        dLdA = 2 * (self.A - self.Y) / (self.N * self.C)

        return dLdA
```

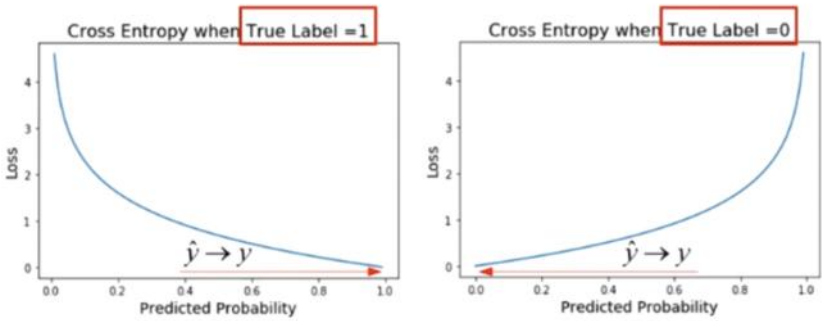
4.2 交叉熵损失

交叉熵损失函数（Cross-Entropy Loss Function），也称为对数损失（Log Loss），是用于分类问题的一种常用损失函数，特别是在二分类和多分类的神经网络模型中。

对于**二分类问题**，交叉熵损失函数的数学表达式为：

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i\, log(p_i) + (1-y_i)\, log(1-p_i)]$$

- N 是样本数量。
- y_i 是第 i 个样本的真实标签，通常是 0 或 1。
- p_i 是模型预测第 i 个样本为正类的概率。
- **该函数物理意义**：如果标签 y=1, 则 L = -log(p)；如果 y=0, 则 L = -log(1-p)。从下图可看出，当预测的值与真值标签差距越大，损失值增长的就越大。

- 

  - **一致性**：指损失函数的单调一致性。

对于**多分类问题**，交叉熵损失扩展为：

$$L = -\frac{1}{N}\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic}\, log(p_{ic})$$

- C 是类别总数。
- y_{ic} 是一个独热编码的向量，表示第 i 个样本是否属于类别 c。
- p_{ic} 是模型预测第 i 个样本属于类别 c 的概率。

优点：

1. **概率解释**： 它直接与模型输出的概率分布相关联，提供了模型预测与实际分布差异的概率解释。
2. **梯度特性**： 它为每个类别提供了不同的梯度值，这有助于模型在学习过程中区分不同类别。
3. **与信息论的联系**： 交叉熵来源于信息论，量化了两个概率分布之间的差异，即模型预测的分布与实际分布。

4.2.1 前向传播

给定模型的原始输出 A 和真实的标签 Y，在交叉熵损失函数的计算中，我们首先需要将网络的原始输出转换为概率分布，这是通过 `softmax` 函数完成的。

对于每个样本 i 和每个类别 j，softmax函数的计算如下：

$$\sigma(A_{ij}) = \frac{e^{A_{ij}}}{\sum_{k=1}^{C} e^{A_{ik}}}$$

- A_{ij} 是第i个样本的第j个特征值。
- 分母是对应样本 i 所有类别输出指数的总和，确保了 $\sigma(A_{ij})$ 的和为1，即概率分布。

交叉熵损失函数定义为：

$$L = -\frac{1}{N}\sum_{i=1}^{N} \sum_{j=1}^{C} Y_{ij}\, log(\sigma(A_{ij}))$$

- N 是样本数量。

- C 是类别数量。
- Y_{ij} 是一个独热编码的标签，如果样本 i 属于类别 j，则 $Y_{ij} = 1$，否则 $Y_{ij} = 0$。

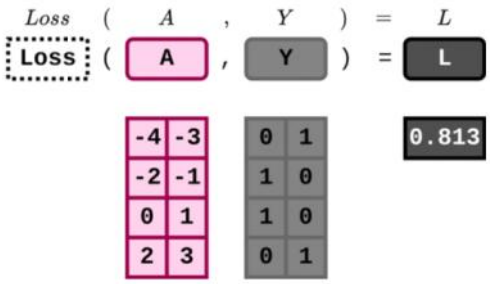


Figure K: Cross Entropy Loss Example

4.2.2 反向传播

在反向传播中，我们需要计算交叉熵损失相对于模型原始输出 A 的梯度，即 $\frac{\partial L}{\partial A}$。

由于真实标签 Y 是独热编码的，对于每个样本 i 和每个类别 j，损失函数 L 关于 A_{ij} 的偏导数可以简化为：

$$\frac{\partial L}{\partial A_{ij}} = \sigma(A_{ij}) - Y_{ij}$$

因此，对于整个输出矩阵 A，其梯度 $dLdA$ 为：

$$dLdA = \frac{1}{N}(\sigma(A) - Y)$$

4.2.3 代码

```python
import numpy as np

class CrossEntropyLoss:

    def softmax(self, x):
        """
        计算softmax概率分布。
        :param x: 输入数组。
        :return: softmax概率分布。
        """
        # 防止指数运算溢出，通过减去每一行的最大值来进行数值稳定化
        x = x - np.max(x, axis=1, keepdims=True)
        exps = np.exp(x)
        return exps / np.sum(exps, axis=1, keepdims=True)

    def forward(self, A, Y):
        """
        计算交叉熵损失。
        :param A: 模型的输出，形状为(N, C)，其中N是样本数量，C是类别数。
        :param Y: 真实标签的独热编码，形状与A相同。
        :return: 交叉熵损失的标量值。
        """
        self.A = A
```

```
        self.Y = Y
        N, C = A.shape  # N是样本数量，C是类别数

        # 计算softmax概率分布
        self.softmax = self.softmax(A)
        # 计算交叉熵
        crossentropy = -np.sum(Y * np.log(self.softmax + 1e-15))  # 加上小数1e-15防止对数为负无穷
        # 计算平均交叉熵损失
        L = crossentropy / N

        return L

    def backward(self):
        """
        计算损失关于模型输出的梯度。
        :return: 损失关于A的梯度。
        """
        N = self.Y.shape[0]  # N是样本数量
        # 计算交叉熵损失对A的梯度
        dLdA = (self.softmax - self.Y) / N

        return dLdA
```

5 优化器

- **优化器**：即网络参数更新规则的方法。
- 在深度学习中，常使用基于梯度下降及其变种的优化器。比如：SGD、Adam等。
- **优化器不全是基于梯度下降类型的**，还有如：遗传算法、模拟退火、粒子群优化、贝叶斯优化等。只是非梯度下降类算法，在深度学习领域，其效率和性能不太好。

5.1 随机梯度下降（SGD）

随机梯度下降（SGD）是深度学习中最基本也是最常用的优化算法之一。其核心思想是利用训练数据的子集（通常称为mini-batch）来近似计算梯度，并使用这个近似梯度来更新模型的权重。

工作原理：

1. **初始化参数**：在训练开始之前，首先随机初始化模型的参数（如权重和偏置）。
2. **计算梯度**：在每次迭代中，随机选择一个小批量的训练样本，然后基于这些样本计算损失函数关于当前参数的梯度。
3. **参数更新**：使用以下公式更新参数：

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

其中，θ 表示模型参数，η 是学习率，$\nabla_\theta J$ 是损失函数 J 关于 θ 的梯度，$x^{(i)}$ 和 $y^{(i)}$ 是随机选中的训练样本。

或者这种表述：

$$W := W - \lambda \frac{\partial L}{\partial W}$$

$$b := b - \lambda \frac{\partial L}{\partial b}$$

其中 λ 是学习率，决定了每一步更新的大小；$\frac{\partial L}{\partial W}$ 和 $\frac{\partial L}{\partial b}$ 是当前参数下损失函数 L 关于权重和偏置的梯度。

优点：

1. **简单有效**：SGD 的实现简单，计算效率高，特别是在处理大规模数据集时。
2. **良好的泛化**：由于每次只使用一部分样本来更新权重，SGD 能够帮助模型避免陷入某些局部最小值，通常能获得更好的泛化性能。

局限性：

1. **收敛速度**：由于每次更新只使用部分数据，SGD 的收敛速度相比于全批量梯度下降（使用全部数据计算梯度）来说可能较慢，尤其是在接近最优解时。
2. **超参数调整**：学习率 η 的设置对SGD的表现至关重要，不当的学习率设置可能导致训练不稳定甚至发散。此外，SGD 需要手动设置一个合适的学习率调整策略，如学习率衰减。
3. **振荡**：由于每次更新只依赖于一部分样本，SGD 的更新路径可能出现较大的振荡，不像基于全数据集的方法那样平滑。

改进方法：

- **动量（Momentum）**：引入动量因子来帮助SGD在相关方向上加速收敛，抑制振荡，通过累积过去梯度的加权平均来调整更新方向。
- **自适应学习率算法**：例如Adagrad、RMSprop和Adam等，这些方法根据参数的历史梯度自动调整各参数的学习率，通常能更快地收敛。

SGD带动量（Momentum）：

SGD带动量（Momentum）是对传统随机梯度下降的改进，它旨在加速学习过程，尤其是对于高曲率、小但一致的梯度或带有噪声的梯度的情况非常有效。

在没有动量的情况下，参数 W（权重）和 b（偏置）的更新规则是基于当前的梯度：

$$W := W - \lambda \frac{\partial L}{\partial W}$$

$$b := b - \lambda \frac{\partial L}{\partial b}$$

其中 λ 是学习率，$\frac{\partial L}{\partial W}$ 和 $\frac{\partial L}{\partial b}$ 是损失函数 L 关于权重 W 和偏置 b 的梯度。

在带动量的SGD中，更新规则添加了前一时间步的更新向量，即“动量”，这个动量不仅考虑了当前的梯度，还考虑了之前梯度的方向和大小。公式如下：

- **首先**，计算动量项 v，它是**当前梯度**和**之前动量项**的线性组合：

$$v_W := \mu v_W + \frac{\partial L}{\partial W}$$

$$v_b := \mu v_b + \frac{\partial L}{\partial b}$$

  其中 μ（通常取值在0和1之间）是动量系数，它决定了前一时间步更新向量 v 对当前更新的贡献程度。如果 μ 为0，那么SGD带动量就退化成了普通的SGD。

- **然后**，使用动量项 v 来更新参数：

$$W := W - \lambda v_W$$

$$b := b - \lambda v_b$$

  在这里，动量项 v_W 和 v_b 相当于加权移动平均的梯度，它不仅包含了当前的梯度信息，还累积了之前的梯度信息，从而在参数空间中创建了一种惯性效果。这有助于参数更新在稳定方向上更加平滑，并且有助于跨越局部最小值和鞍点。

简而言之，动量项使得参数更新不仅受到当前梯度的影响，还受到历史梯度的影响，这样可以平滑震荡和加速收敛。在实践中，使用动量通常可以达到更快的收敛速度，并有助于避免陷入局部最小值。

代码：

- **类属性**：
  - `l`：模型层的列表
  - `L`：模型层的数量
  - `lr`：学习率，可调的超参数，用于调整更新的大小
  - `mu`：动量μ，可调的超参数，控制以前更新影响当前更新的程度。μ = 0表示无动量。
  - `v_w`：每一层的权重速度列表
  - `v_b`：每一层的偏置速度列表
- **类方法**：
  - `step`：更新模型层的W和b：
    - 因为参数梯度告诉我们哪个方向使模型变得更糟，我们向相反的方向移动梯度来更新参数。
    - 当动量不为零时，更新速度v_w和v_b，它们是梯度变化的方向，有助于到达全局最小值。先前更新的速度通过超参数 μ 来缩放。

```
import numpy as np

class SGD:
    def __init__(self, model, lr=0.1, momentum=0):
        """
        初始化SGD优化器。
        :param model: 包含需要优化的参数的模型。
```

```
        :param lr: 学习率，默认为0.1。
        :param momentum: 动量系数，默认为0，如果不为0，则应用动量优化。
        """
        # 筛选出模型中所有含有权重W的层，存入列表self.l
        # self.l = list(filter(lambda x: hasattr(x, 'W'), model.layers))
        self.l = model.layers # HW1P1测试用例可直接使用模型所有层
        # 获取模型层数量
        self.L = len(self.l)
        # 设置学习率本
        self.lr = lr
        # 设置动量系数
        self.mu = momentum
        # 初始化权重和偏置的动量变量为0，形状与对应权重相同
        self.v_W = [np.zeros_like(layer.W) for layer in self.l]
        self.v_b = [np.zeros_like(layer.b) for layer in self.l]

    def step(self):
        """
        执行一步参数更新。
        """
        # 遍历每一层
        for i in range(self.L):
            if self.mu == 0:  # 如果动量系数为0，执行标准的SGD更新
                # 直接用当前层的梯度和学习率更新权重和偏置
                self.l[i].W -= self.lr * self.l[i].dLdW
                self.l[i].b -= self.lr * self.l[i].dLdb
            else:
                # 更新动量
                self.v_W[i] = self.mu * self.v_W[i] + self.l[i].dLdW
                self.v_b[i] = self.mu * self.v_b[i] + self.l[i].dLdb
                # 使用更新后的动量变量更新权重和偏置
                self.l[i].W -= self.lr * self.v_W[i]
                self.l[i].b -= self.lr * self.v_b[i]
```

5.2 Adam

Adam（Adaptive Moment Estimation）优化器是一种在深度学习中广泛使用的算法，它结合了动量（Momentum）和RMSprop两种优化方法的优点。Adam 通过计算梯度的一阶矩估计（即均值）和二阶矩估计（即未中心化的方差），自适应地调整每个参数的学习率。

工作原理

Adam优化器的主要步骤包括：

1. **初始化**：

- 初始化参数 θ，通常是随机的。
- 初始化一阶矩向量 m 和二阶矩向量 v，均为0。
- 初始化时间步 t，起始为0。
- 选择合适的学习率 α、一阶矩估计的指数衰减率 β_1、二阶矩估计的指数衰减率 β_2 和非常小的数值 ϵ（用于防止除以零）。

- **每步更新**：
  - 在每个时间步 t，首先计算当前批次的梯度 g_t。
  - 更新一阶矩估计：$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$。
  - 更新二阶矩估计：$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$。
  - 修正一阶矩偏差：$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$。
  - 修正二阶矩偏差：$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$。
  - 更新参数：$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$。

优点

1. **自适应学习率**：Adam 根据参数的不同自动调整学习率，减少了学习率选择的复杂性。
2. **适用性广**：Adam 由于其鲁棒性，被广泛应用于各种不同的深度学习应用中，从计算机视觉到自然语言处理。
3. **高效**：在许多情况下，Adam 显示出比其他优化算法更快的收敛速度。

局限性

1. **内存消耗**：由于需要存储每个参数的一阶和二阶矩估计，Adam 的内存需求比简单的梯度下降或SGD要高。
2. **可能的非收敛行为**：在某些情况下，Adam 可能不会收敛到最小值，尤其是在梯度稀疏或噪声较大的情况下。

尽管有局限性，Adam 仍然是深度学习优化中最受欢迎的算法之一，其高效的性能和易用性使其成为许多研究和工业应用的首选优化器。

代码

```
import numpy as np
import pdb  # Python 的调试工具

class Adam():
    def __init__(self, model, lr, beta1=0.9, beta2=0.999, eps=1e-8):
        # 初始化优化器，设置参数
        self.l = model.layers[::2]  # 每2层取第1层，因为第2层为激励函数层
        self.beta1 = beta1  # 一阶矩估计的衰减率
        self.beta2 = beta2  # 二阶矩估计的衰减率
        self.eps = eps  # 防止除零的小常数
        self.lr = lr  # 学习率
```

```
        self.t = 0  # 初始化时间步

        # 初始化一阶矩和二阶矩阵列
        self.m_W = [np.zeros(l.W.shape, dtype="f") for l in self.l]  # 权重的一阶矩
        self.v_W = [np.zeros(l.W.shape, dtype="f") for l in self.l]  # 权重的二阶矩

        self.m_b = [np.zeros(l.b.shape, dtype="f") for l in self.l]  # 偏置的一阶矩
        self.v_b = [np.zeros(l.b.shape, dtype="f") for l in self.l]  # 偏置的二阶矩

    def step(self):
        # 执行一步参数更新
        self.t += 1
        for layer_id, layer in enumerate(self.l):
            # pdb.set_trace()  # 可在此处设置断点，用于调试

            # 计算权重的一阶和二阶矩更新
            self.m_W[layer_id] = self.beta1 * self.m_W[layer_id] + (1 - self.beta1) *
layer.dLdW
            self.v_W[layer_id] = self.beta2 * self.v_W[layer_id] + (1 - self.beta2) *
(layer.dLdW ** 2)

            # 计算偏置的一阶和二阶矩更新
            self.m_b[layer_id] = self.beta1 * self.m_b[layer_id] + (1 - self.beta1) *
layer.dLdb
            self.v_b[layer_id] = self.beta2 * self.v_b[layer_id] + (1 - self.beta2) *
(layer.dLdb ** 2)

            # 计算偏差修正后的一阶和二阶矩
            m_W_hat = self.m_W[layer_id] / (1 - self.beta1 ** self.t)
            v_W_hat = self.v_W[layer_id] / (1 - self.beta2 ** self.t)
            m_b_hat = self.m_b[layer_id] / (1 - self.beta1 ** self.t)
            v_b_hat = self.v_b[layer_id] / (1 - self.beta2 ** self.t)

            # 更新权重和偏置
            layer.W -= self.lr * m_W_hat / (np.sqrt(v_W_hat + self.eps))
            layer.b -= self.lr * m_b_hat / (np.sqrt(v_b_hat + self.eps))
```

5.3 AdamW

AdamW 是一种深度学习优化算法，它基于 Adam 优化器，并在权重衰减（weight decay）的处理上进行了改进。这种优化器由 Ilya Loshchilov 和 Frank Hutter 在 2017 年提出，目的是解决原始 Adam 优化器在应用 L2 正则化时可能导致性能下降的问题。

工作原理

AdamW 优化器的核心在于如何处理权重衰减。在传统的 Adam 优化器中，权重衰减通常被实现为损失函数的一部分，即通过在损失函数中加入对参数的 L2 惩罚来实现。然而，这种方法在使用自适应学习率算法（如 Adam）时可能会导致不

理想的权重更新，因为权重衰减直接与梯度的大小挂钩，这可能会破坏梯度的信息。

与之不同，AdamW 将权重衰减与梯度更新分离。具体来说，它调整参数更新的方式，使权重衰减独立于梯度更新，从而更加有效地实现正则化目标。这一改进使得模型训练更稳定，尤其是在大规模训练和复杂模型中。

更新规则

AdamW 的更新规则如下：

1. **计算梯度**：首先计算损失函数关于参数 θ_t 的梯度 g_t。
2. **更新一阶和二阶矩估计**：
   - 一阶矩 $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
   - 二阶矩 $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
- **修正偏差**：
  - 修正一阶矩 $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
  - 修正二阶矩 $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
- **权重衰减**：在应用梯度更新之前，首先对参数应用权重衰减：
  - $\theta_t = \theta_t - \eta \cdot \lambda \cdot \theta_t$
- **参数更新**：
  - $\theta_{t+1} = \theta_t - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$

其中，η 是学习率，λ 是权重衰减系数，β_1 和 β_2 是动量项的衰减率，ϵ 是防止除以零的小常数。

优点

- **有效的正则化**：通过分离权重衰减和梯度更新，AdamW 在包含正则化的训练中提供了更好的性能和泛化。
- **适应性强**：继承了 Adam 的自适应学习率优点，对各种类型的数据和任务都有良好表现。

局限性

- **超参数调整**：需要仔细选择学习率、权重衰减系数等超参数。
- **计算开销**：与原始的 Adam 相比，实现更为复杂，可能稍微增加计算开销。

AdamW 的提出是对传统优化算法的一次重要改进，特别是在处理深度学习中权重衰减时。这使得它在需要细致调整正则化影响的应用中尤为有用。

代码：

```
import numpy as np

class AdamW():
    def __init__(self, model, lr, beta1=0.9, beta2=0.999, eps=1e-8, weight_decay=0.01):
        # 初始化优化器参数
        self.l = model.layers[::2]
```

```
        self.beta1 = beta1  # 一阶矩估计的衰减率
        self.beta2 = beta2  # 二阶矩估计的衰减率
        self.eps = eps  # 防止除零的小常数
        self.lr = lr  # 学习率
        self.t = 0  # 初始化时间步
        self.weight_decay = weight_decay  # 权重衰减率

        # 初始化一阶矩和二阶矩数组
        self.m_W = [np.zeros(l.W.shape, dtype="f") for l in self.l]  # 权重的一阶矩
        self.v_W = [np.zeros(l.W.shape, dtype="f") for l in self.l]  # 权重的二阶矩

        self.m_b = [np.zeros(l.b.shape, dtype="f") for l in self.l]  # 偏置的一阶矩
        self.v_b = [np.zeros(l.b.shape, dtype="f") for l in self.l]  # 偏置的二阶矩

    def step(self):
        # 执行一步参数更新
        self.t += 1
        for layer_id, layer in enumerate(self.l):
            # 计算权重的一阶和二阶矩更新
            self.m_W[layer_id] = self.beta1 * self.m_W[layer_id] + (1 - self.beta1) *
layer.dLdW
            self.v_W[layer_id] = self.beta2 * self.v_W[layer_id] + (1 - self.beta2) *
(layer.dLdW ** 2)

            # 计算偏置的一阶和二阶矩更新
            self.m_b[layer_id] = self.beta1 * self.m_b[layer_id] + (1 - self.beta1) *
layer.dLdb
            self.v_b[layer_id] = self.beta2 * self.v_b[layer_id] + (1 - self.beta2) *
(layer.dLdb ** 2)

            # 计算偏差修正后的一阶和二阶矩
            m_W_hat = self.m_W[layer_id] / (1 - self.beta1 ** self.t)
            v_W_hat = self.v_W[layer_id] / (1 - self.beta2 ** self.t)
            m_b_hat = self.m_b[layer_id] / (1 - self.beta1 ** self.t)
            v_b_hat = self.v_b[layer_id] / (1 - self.beta2 ** self.t)

            # 先应用权重衰减，然后更新权重和偏置
            layer.W = layer.W - layer.W * self.lr * self.weight_decay  # 应用权重衰减
            layer.b = layer.b - layer.b * self.lr * self.weight_decay  # 应用权重衰减

            layer.W -= self.lr * m_W_hat / (np.sqrt(v_W_hat + self.eps))  # 更新权重
            layer.b -= self.lr * m_b_hat / (np.sqrt(v_b_hat + self.eps))  # 更新偏置
```

6 正则化

正则化:

- 通过对**模型结构**、**损失函数**、**学习过程**等增加某种形式的 约束 或 惩罚，以减少模型的复杂性，从而**防止过拟合**。

为什么叫 “正则化” ？

词根“regula”在拉丁语中意味着 **规则** 或 **直线**。这个词准确地描述了在模型训练过程中引入额外规则（即惩罚项）来“规范”模型行为。

常见正则化方法：

1. **L1 正则化和 L2 正则化**：
   - **L1 正则化（Lasso）**：通过向损失函数添加权重的绝对值之和作为惩罚项来实施。这种方法倾向于产生稀疏权重矩阵（许多权重为零），有助于特征选择。
   - **L2 正则化（Ridge）**：通过添加权重的平方和作为惩罚项到损失函数中。这不会导致稀疏模型，但会减少所有权重的大小，使得模型对输入中的噪声不那么敏感。
2. **Dropout**：
   - 在深度学习中特别流行的一种正则化技术。它并不通过修改损失函数来实现，而是在训练过程中随机丢弃（即暂时移除）网络中的部分神经元（包括它们的连接）。这迫使网络在训练过程中不依赖于任何一个特征，从而能够增强模型的泛化能力。
3. **早停（Early Stopping）**：
   - 另一种常见的正则化技术，通过在验证数据集上的性能不再提升时停止训练来实施。这样可以防止模型在训练数据上过度优化和学习噪声。
4. **数据增强（Data Augmentation）**：
   - 在训练深度学习模型时，通过对训练数据应用各种随机但现实的变换来增加样本的多样性，这本身也是一种正则化形式，因为它有助于模型学习到更加健壮的特征。

6.1 BN（批归一化）

关于BN层介绍，见OneNote笔记《模块：BN 、LN、IN、GN标准化》。

6.1.1 前向传播训练公式（`eval=False`）

在训练模式下，对于输入的数据 $Z \in \mathbb{R}^{N \times D}$（其中 N 是批次大小，D 是特征的维度），BN 层会计算每个特征的均值 μ 和方差 σ^2，然后使用这些参数进行归一化，最后应用可学习的参数进行缩放和位移。

- **计算均值**：

$$\mu_B = \frac{1}{N} \sum_{i=1}^{N} Z_i$$

这是每个特征在当前批次中所有样本的平均值。

- **计算方差**：

$$\sigma_B^2 = \frac{1}{N} \sum_{i=1}^{N} (Z_i - \mu_B)^2$$

这是每个特征在当前批次中所有样本的方差。

- **归一化**:

$$\hat{Z}_i = \frac{Z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- **缩放和位移**:

$$Y_i = \gamma \hat{Z}_i + \beta$$

其中，Y_i 是 BN 层的输出，γ 是缩放参数（对应代码中的 `self.BW`），β 是位移参数（对应代码中的 `self.Bb`），ϵ 是一个很小的常数，防止除以零。

6.1.2 前向传播推理公式（`eval=True`）

在推理模式下，我们使用在训练过程中计算得到的运行时均值和方差来进行归一化。

- **归一化**:

$$\hat{Z}_i = \frac{Z_i - \mu_{\text{running}}}{\sqrt{\sigma_{\text{running}}^2 + \epsilon}}$$

- **缩放和位移**（同训练模式）:

$$Y_i = \gamma \hat{Z}_i + \beta$$

其中 μ_{running} 和 $\sigma_{\text{running}}^2$ 分别是在训练期间计算的运行时均值和方差。

在批量归一化（Batch Normalization）过程中，运行时均值（μ_{running}）和运行时方差（$\sigma_{\text{running}}^2$）通常是通过指数加权移动平均（Exponential Moving Average，EMA）来更新的。更新公式如下：

- **运行时均值的更新公式**:

$$\mu_{\text{running}} = \alpha \cdot \mu_{\text{running}} + (1 - \alpha) \cdot \mu_{\text{batch}}$$

- **运行时方差的更新公式**:

$$\sigma_{\text{running}}^2 = \alpha \cdot \sigma_{\text{running}}^2 + (1 - \alpha) \cdot \sigma_{\text{batch}}^2$$

这里的 μ_{batch} 和 σ_{batch}^2 分别是当前批次数据的均值和方差。α 是一个介于 0 和 1 之间的超参数，称为衰减系数（decay factor），它决定了历史统计信息的权重。当 α 接近1时，历史统计信息的权重更大，更新速度变慢；当 α 较小，新批次数据的影响更大。

在实际应用中，μ_{running} 和 $\sigma_{\text{running}}^2$ 通常在每个训练批次后更新，并用于模型的推理阶段。这样可以确保模型在推理时使用的是整个训练数据的全局统计信息，而不是仅仅依赖于单个批次的局部统计信息。

6.1.3 反向传播公式

在BN层的反向传播过程中，我们需要计算3部分梯度：

- 损失函数 L 关于原始输入Z的梯度：用于继续往前传递梯度。
- 损失函数 L 关于缩放参数 γ 和位移参数 β 的梯度：用于更新γ 和β 的值。
- **关于** β 的梯度：

  BN层的前向传播公式如下：

$$Y_i = \gamma \hat{Z}_i + \beta$$

  这里：

- Y_i 是 BN 层输出的第 i 个样本值。

- $\hat{Z}_i$ 是归一化后的第 i 个样本值。

- γ 是缩放因子。

- β 是位移项。

我们需要计算损失函数 L （**注意这个loss是整个batch的总loss**）相对于 β 的梯度，记为 $\frac{\partial L}{\partial \beta}$。

考虑到 β 是一个标量，并且直接加到每个归一化后的样本 $\hat{Z}_i$ 上，我们可以将损失函数 L 相对于 β 的梯度看作是损失函数 L 相对于 BN 层输出 Y 的梯度 $\frac{\partial L}{\partial Y}$ 的和。

根据链式法则：

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{N} \frac{\partial L}{\partial Y_i} \cdot \frac{\partial Y_i}{\partial \beta}$$

由于 $Y_i = \gamma \hat{Z}_i + \beta$，$\beta$ 对每个样本的影响是相同的，即对于任何 i：

$$\frac{\partial Y_i}{\partial \beta} = 1$$

因此，损失函数 L 相对于 β 的梯度就是对所有样本的 $\frac{\partial L}{\partial Y_i}$ 的和：

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{N} \frac{\partial L}{\partial Y_i}$$

其中 N 是批次中的样本数量，Y_i 是 BN 层输出的第 i 个样本值。

- **关于** γ 的梯度：

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{N} \frac{\partial L}{\partial Y_i} \hat{Z}_i$$

同理，这里 $\hat{Z}_i$ 是 γ 乘以的对象，所以 γ 的梯度是 $\frac{\partial L}{\partial Y_i}$ 和 $\hat{Z}_i$ 的乘积的和。

- **关于** Z_i 的梯度，这是最复杂的部分，需要应用链式法则和一些微积分运算：

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i}\frac{\partial \hat{Z}_i}{\partial Z_i} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial Z_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i}\frac{1}{\sqrt{\sigma_B^2+\epsilon}} + \frac{\partial L}{\partial \sigma_B^2}\frac{2(Z_i-\mu_B)}{N} + \frac{\partial L}{\partial \mu_B}\frac{1}{N}$$

其中：

$\frac{\partial L}{\partial \hat{Z}_i} = \frac{\partial L}{\partial Y_i}\frac{\partial Y_i}{\partial \hat{Z}_i} = \frac{\partial L}{\partial Y_i}\gamma$

$\frac{\partial L}{\partial \sigma_B^2} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i}(\hat{Z}_i - \mu_B)(-\frac{1}{2})(\sigma_B^2+\epsilon)^{-\frac{3}{2}}$

$\frac{\partial L}{\partial \mu_B} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i}(-\frac{1}{\sqrt{\sigma_B^2+\epsilon}}) + \frac{\partial L}{\partial \sigma_B^2}\frac{\sum_{i=1}^{N} -2(Z_i-\mu_B)}{N}$

举例：

假设我们有一个很简单的批次数据$Z = [1,2,3]$ 和$\gamma = 2$、$\beta = 1$，损失函数关于Y 的梯度$G = [0.1,0.2,0.3]$。计算过程如下：

1. 前向传播计算μ、σ^2 和$\hat{Z}$：

- $\mu = \frac{1+2+3}{3} = 2$
- $\sigma^2 = \frac{(1-2)^2+(2-2)^2+(3-2)^2}{3} = \frac{2}{3}$
- $\hat{Z} = \frac{Z-\mu}{\sqrt{\sigma^2+\epsilon}} = [-1.2247, 0, 1.2247]$

2. 反向传播计算$\frac{\partial L}{\partial Z}$:

- $$\frac{\partial L}{\partial \beta} = 0.1 + 0.2 + 0.3 = 0.6$$
- $$\frac{\partial L}{\partial \gamma} = 0.1 \cdot (-1.2247) + 0.2 \cdot 0 + 0.3 \cdot 1.2247 = 0.12247$$
- $\frac{\partial L}{\partial Z}$的计算需要通过上面的公式，涉及到$\frac{\partial L}{\partial \sigma^2}$和$\frac{\partial L}{\partial \mu}$，这部分计算较为复杂。

6.1.4 代码

Table 6: Batch Normalization Components

Code Name	Math	Type	Shape	Meaning
N	N	scalar	-	batch size
num_features	C	scalar	-	number of features (same for input and output)
alpha	α	scalar	-	the coefficient used for running_M and running_V computations
eps	ϵ	scalar	-	a value added to the denominator for numerical stability.
Z	Z	matrix	$N \times C$	data input to the BN layer
NZ	$\hat{Z}$	matrix	$N \times C$	normalized input data
BZ	$\tilde{Z}$	matrix	$N \times C$	data output from the BN layer
M	μ	matrix	$1 \times C$	Mini-batch per feature mean
V	σ^2	matrix	$1 \times C$	Mini-batch per feature variance
running_M	$E[Z]$	matrix	$1 \times C$	Running average of per feature mean
running_V	$Var[Z]$	matrix	$1 \times C$	Running average of per feature variance
BW	γ	matrix	$1 \times C$	Scaling parameters
Bb	β	matrix	$1 \times C$	Shifting parameters
dLdBW	$\partial L/\partial \gamma$	matrix	$1 \times C$	how changes in γ affect loss
dLdBb	$\partial L/\partial \beta$	matrix	$1 \times C$	how changes in β affect loss
dLdZ	$\partial L/\partial Z$	matrix	$N \times C$	how changes in inputs affect loss
dLdNZ	$\partial L/\partial \hat{Z}$	matrix	$N \times C$	how changes in $\hat{Z}$ affect loss
dLdBZ	$\partial L/\partial \tilde{Z}$	matrix	$N \times C$	how changes in $\tilde{Z}$ affect loss
dLdV	$\partial L/\partial (\sigma^2)$	matrix	$1 \times C$	how changes in (σ^2) affect loss
dLdM	$\partial L/\partial \mu$	matrix	$1 \times C$	how changes in μ affect loss

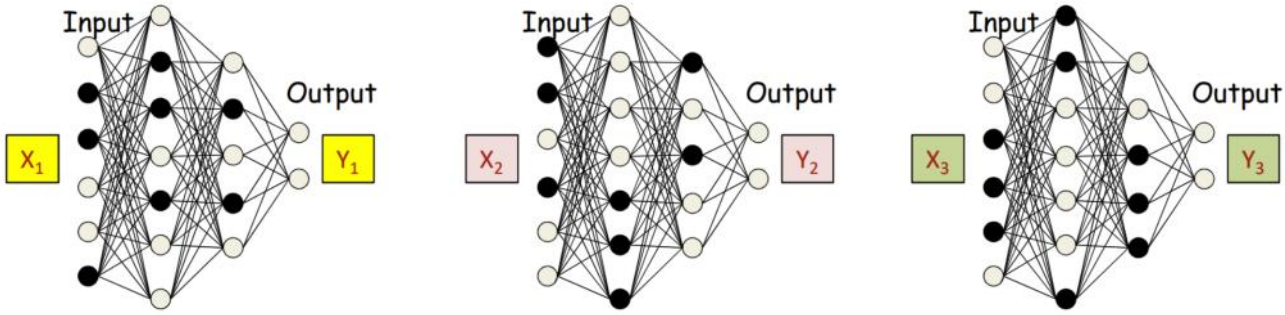
```python
import numpy as np


class BatchNorm1d:
    def __init__(self, num_features, alpha=0.9):
        """
        批量归一化层的初始化函数。

        :param num_features: 特征的数量
        :param alpha: 用于计算运行均值和方差的指数加权移动平均系数
        """
        self.alpha = alpha  # 指数加权移动平均的系数
        self.eps = 1e-8  # 防止除以零的小数值

        # 初始化可学习参数，缩放因子和位移项
        self.BW = np.ones((1, num_features))
        self.Bb = np.zeros((1, num_features))
```

```
        # 梯度初始化
        self.dLdBW = np.zeros((1, num_features))
        self.dLdBb = np.zeros((1, num_features))

        # 运行时均值和方差，训练时更新，推理时使用
        self.running_M = np.zeros((1, num_features))
        self.running_V = np.ones((1, num_features))

    def forward(self, Z, eval=False):
        """
        批量归一化层的前向传播函数。

        :param Z: 输入数据
        :param eval: 是否为推理模式
        :return: 归一化并缩放、位移后的数据
        """
        self.Z = Z  # 输入数据
        self.N = Z.shape[0]  # 数据批次大小

        if not eval:
            # 训练模式
            self.M = np.mean(Z, axis=0)  # 计算均值
            self.V = np.var(Z, axis=0)  # 计算方差

            # 更新运行时均值和方差
            self.running_M = self.alpha * self.running_M + (1 - self.alpha) * self.M
            self.running_V = self.alpha * self.running_V + (1 - self.alpha) * self.V

            # 归一化处理
            self.NZ = (Z - self.M) / np.sqrt(self.V + self.eps)
        else:
            # 推理模式
            self.NZ = (Z - self.running_M) / np.sqrt(self.running_V + self.eps)

        # 缩放和位移
        self.BZ = self.BW * self.NZ + self.Bb
        return self.BZ

    def backward(self, dLdBZ):
        """
        批量归一化层的后向传播函数。

        :param dLdBZ: 损失函数关于批量归一化后数据的梯度
        :return: 损失函数关于批量归一化前数据的梯度
        """
        # 计算可学习参数的梯度
        self.dLdBW = np.sum(dLdBZ * self.NZ, axis=0, keepdims=True)
        self.dLdBb = np.sum(dLdBZ, axis=0, keepdims=True)
```

```
        # 计算传递给前一层的梯度
        dLdNZ = dLdBZ * self.BW
        dLdV = np.sum(dLdNZ * (self.Z - self.M) * -0.5 * (self.V + self.eps) ** (-3/2),
axis=0, keepdims=True)
        dLdM = np.sum(dLdNZ * -1 / np.sqrt(self.V + self.eps), axis=0, keepdims=True) +
\
               dLdV * np.mean(-2 * (self.Z - self.M), axis=0, keepdims=True)

        # 归一化前数据的梯度
        dLdZ = dLdNZ / np.sqrt(self.V + self.eps) + \
               dLdV * 2 * (self.Z - self.M) / self.N + \
               dLdM / self.N

        return dLdZ
```

6.2 dropout

dropout：指在训练过程中随机丢弃（即**暂时移除**）网络中的部分神经元（包括它们的连接）。这迫使网络在训练过程中不依赖于任何一个特征，从而能够增强模型的泛化能力。



6.2.1 前向传播

在前向传播过程中，Dropout 通过以下步骤进行计算：

1. **生成掩码（Mask）**：
   - 对于每个神经元，以概率 $1-p$ 保留该神经元的输出，以概率 p 将该神经元的输出置为0。
   - 掩码 mask 是一个与输入 x 形状相同的二值矩阵，掩码中的每个元素都独立地服从伯努利分布（也即0-1分布，只有两种结果）：

$$\text{mask}_i \sim \text{Bernoulli}(1-p)$$

2. **应用掩码**：
   - 将输入 x 与掩码 mask 逐元素相乘：

$$x' = x \times \text{mask}$$

   - 为了在训练和测试时保持相同的激活期望值，对被保留下来的神经元进行缩放：

$$y = \frac{x'}{1-p}$$

   具体必要性如下：

在实施 Dropout 正则化时，缩放（或重新缩放）非常关键，原因在于保持训练和测试时神经元的输出激活总量大致相同。这是为了避免在模型训练和推理（测试）阶段发生激活值的显著变化，从而提高模型的泛化能力。具体来说，有以下几个原因：

1. 维持激活值的一致性

在训练阶段，由于 Dropout 会随机地将部分神经元的输出置为 0，直接结果是这部分神经元在当前训练步骤中不会对后续层产生任何贡献。如果不进行缩放，那么当 Dropout 被应用时，神经元的平均激活输出会减少。这可能导致在训练和测试时网络行为的不一致，因为在测试时通常不使用 Dropout，所有的神经元都是活跃的。

2. 防止训练和测试时性能差异

如果在训练时应用了 Dropout 而不进行缩放，那么由于神经元输出的平均值下降，训练得到的权重可能会适应于这种较低水平的输入信号。然而，在测试时，所有神经元都是活跃的，输入的总激活量会相对较高，这会导致模型在测试时表现出较大的波动或性能下降。

3. 保持输出的期望不变

理论上，对于概率 p 的 Dropout，神经元被保留的概率是 $1-p$。因此，任何给定的输入在训练过程中平均只有 $1-p$ 的部分被激活。为了使网络在训练和测试阶段对输入的响应保持一致，我们需要对那些在训练中未被丢弃的神经元输出进行缩放，即乘以 $\frac{1}{1-p}$。这样可以确保无论 Dropout 层是否激活，输出的总体期望保持一致。

实例说明

例如，如果设置 Dropout 率 $p=0.5$，那么在训练时大约有一半的神经元输出将被置为零。为了补偿这种丢失，剩余的活跃神经元输出将被放大两倍（即乘以 $\frac{1}{0.5}=2$），以保证整体输出信号的强度与没有使用 Dropout 时大致相同。

通过这种方式，Dropout 不仅帮助提高了模型对数据的泛化能力，还确保了在不同模式（训练与测试）之间切换时模型的稳定性。这种缩放处理是实现 Dropout 功能的一个重要步骤。

6.2.2 反向传播

在反向传播过程中，被丢弃的神经元本次不会对梯度进行更新，因为这些神经元的参数本次前向传播时并没有参与损失计算，所以反向传播时也不需要更新参数。即梯度只要原封不动的传递给参与了计算的神经元。

1. **传递梯度**：
   - 将反向传播得到的梯度 δ 与掩码 mask 逐元素相乘：

$$\delta' = \delta \times \text{mask}$$

   - 由于丢弃的神经元在前向传播时被置为0，所以这些神经元的梯度也应为0。

6.2.3 代码

备注：关于 `np.random.binomial(1, 1-self.p, x.shape)`：

1. **功能**：使用二项分布生成随机数。
2. **第一个参数**： 1 表示每次试验的“成功”次数上限，在这里每个元素只有一次机会被保留或丢弃，因此设置为1。
3. **第二个参数**： 1-self.p 是每次试验中成功（即元素被保留）的概率。

4. **第三个参数**：是要生成的随机数的维度。这保证了生成的掩码矩阵与输入矩阵 `x` 的大小完全相同。

```python
import numpy as np
import pdb

class Dropout(object):
    def __init__(self, p=0.5):
        self.p = p

    def __call__(self, x):
        # 使得实例可以像函数一样被调用，直接传递x给前向传播
        return self.forward(x)

    def forward(self, x, train=True):
        if train:
            # 生成一个随机的二值掩码（mask），形状与输入 x 相同
            self.mask = np.random.binomial(1, 1-self.p, x.shape)
            # 应用掩码，并对非置零的输入进行缩放
            return x * self.mask * 1/(1-self.p)
        else:
            # 如果是测试模式，不使用dropout
            return x

    def backward(self, delta):
        # 将前向传播时生成的掩码应用到梯度上
        # 这样可以确保只有在前向传播中未被置零的神经元才会在反向传播中更新权重
        return delta * self.mask
```