【完成】HW2-P1

2024年8月12日 星期一　0:16

S24讲义找不到，用的S23版的，bonus部分用的S24版的，我把两个部分合并了（需要重写Conv2D类，主要是加个padding属性）。

1 目录

2 项目文件结构

```
HW2P1
├─ autograder
│   ├─ hw2_bonus_autograder
│   │   ├─ runner.py                    # 验证HW2P1-bonus各模块的脚本
│   │   └─ test.py
│   ├─ helpers.py
│   ├─ runner.py                        # 验证HW2P1各模块的脚本
│   └─ test.py
├─ models
│   ├─ cnn.py
│   ├─ mlp.py
│   ├─ mlp_scan.py
│   └─ resnet.py
├─ mytorch
│   ├─ activation.py
│   ├─ batchnorm2d.py
│   ├─ Conv1d.py
│   ├─ Conv2d.py
│   ├─ ConvTranspose.py
│   ├─ dropout2d.py
│   ├─ flatten.py
│   ├─ linear.py
│   ├─ loss.py
│   ├─ pool.py
│   ├─ resampling.py
└─ requirements.txt
```

代码测试方法:

- HW2P1部分：**python autograder/runner.py**
-
```
Total score: 100.0/100.0

Summary:
{"scores": {"MCQ 1": 1, "MCQ 2": 1, "MCQ 3": 1, "MCQ 4": 1, "MCQ 5": 1, "Downsampling1d": 2.5, "Upsampling1d": 2.5
, "Downsampling2d": 2.5, "Upsampling2d": 2.5, "Conv1d_stride1": 10, "Conv1d": 5, "Conv2d-stride1": 10, "Conv2d": 5
, "convTranspose1d": 5, "convTranspose2d": 5, "MaxPool2d_stride1": 10, "MaxPool2d": 5, "MeanPool2d_stride1": 10, "
MeanPool2d": 5, "CNN as Simple Scanning MLP": 5, "CNN as Distributed Scanning MLP": 5, "Build a CNN Model": 5}}
```
- HW2P1-bonus部分：**python autograder/hw2_bonus_autograder/runner.py**

```
(torch) E:\project_2024\CMU_11785_2024S\HW2P1>python autograder/hw2_bonus_autograder/runner.py
--------------------
Section 3 - Dropout2d
Passed Dropout2d Test
Dropout2d:  PASS
--------------------

--------------------
Section 4 - BatchNorm2d
Passed BatchNorm2d Test
BatchNorm2d:  PASS
--------------------

--------------------
Section 4 - Resnet
Passed ResBlock Test
Resnet:  PASS
--------------------

{"scores": {"Dropout2d": 5, "BatchNorm2d": 5, "Resnet": 5}}
```

3 上下采样

3.1 上采样（1D）

在深度学习中，上采样通常指将输入图像或特征图的大小增加，以便与另一组特征图或期望输出的大小相匹配。上采样通常与下采样操作（如池化或卷积）结合使用，以构建具有不同分辨率的网络。这里用最简单的零填充。

3.1.1 前向传播

在MyTorch框架的实现中，我们通过一个简单方法来实现上采样，即在元素之间插入k-1个0值（其中k为上采样因子）。如下所示：

Figure 5: Upsampling1d Forward Example (k=2)

输入元素个数和输出元素个数关系如下：

$$W_{out} = k \times (W_{in} - 1) + 1$$

3.1.2 反向传播

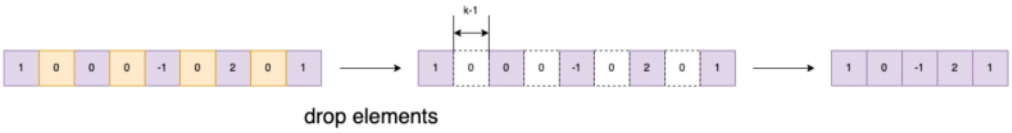


Figure 6: Upsampling1d Backward Example (k=2)

在反向传播过程中，我们执行前向传播的逆操作。这意味着从上采样的输出数据中移除之前添加的零。在这个过程中，只有与原始输入数据对应的导数（梯度）dL/dZ被保留，而对应于插入的零的导数（梯度）则被丢弃，因为这些零不会影响损失函数的导数计算。

3.1.3 代码

Table 1: Upsample1d Class Components

Code Name	Math	Type	Shape	Meaning
batch_size	N	scalar	-	batch size
in_channels	C	scalar	-	Number of channels
input_width	W_{in}	scalar	-	Width of input channels
output_width	W_{out}	scalar	-	Width of output channels
upsampling_factor	k	scalar	-	upsampling_factor $\in \mathbb{Z}^+$
A	A	matrix	$N \times C \times W_{in}$	pre-upsampling values
Z	Z	matrix	$N \times C \times W_{out}$	post-upsampling values
dLdZ	$\partial L/\partial Z$	matrix	$N \times C \times W_{out}$	gradient of Loss wrt Z
dLdA	$\partial L/\partial A$	matrix	$N \times C \times W_{in}$	gradient of Loss wrt A

```python
class Upsample1d():
    def __init__(self, upsampling_factor):
        """
        类初始化函数。

        参数:
            upsampling_factor (int): 上采样因子。
        """
        self.upsampling_factor = upsampling_factor

    def forward(self, A):
        """
        上采样的前向传播。

        参数:
            A (np.array): 输入数组，形状为(batch_size, in_channels, input_width)。

        返回:
            Z (np.array): 上采样后的数组，形状为(batch_size, in_channels, output_width)。
        """
        batch_size, in_channels, input_width = A.shape
        output_width = self.upsampling_factor * (input_width - 1) + 1

        # 初始化输出数组
        Z = np.zeros((batch_size, in_channels, output_width))

        # 使用切片操作填充非零值
        Z[:, :, ::self.upsampling_factor] = A # 从0开始，每隔upsampling_factor个元素赋值为A中的元素

        return Z

    def backward(self, dLdZ):
        """
        上采样的反向传播。

        参数:
            dLdZ (np.array): 损失函数关于上采样输出的梯度，形状为(batch_size, in_channels, output_width)。

        返回:
            dLdA (np.array): 损失函数关于上采样输入的梯度，形状为(batch_size, in_channels, input_width)。
        """

        # 从上采样的梯度中提取原始像素位置的梯度
```

```
        dLdA = dLdZ[:, :, ::self.upsampling_factor] # 从0开始，每隔upsampling_factor个元素提取dLdZ中的元素

        return dLdA
```

3.2 下采样（1D）

在深度学习中，下采样通常用于减少数据的尺寸或者降低特征图的分辨率。

从实现的角度来看，上采样和下采样是彼此的逆操作。

3.2.1 前向传播

在前向传播中，下采样**保留每k个元素中的第一个元素**并丢弃其他元素来，从而减少输入数据的尺寸。这里的k是下采样因子。

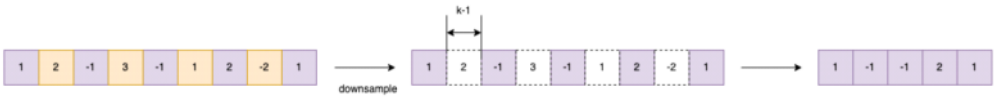


Figure 7: Downsampling1d Forward Example (k=2)

输入元素个数和输出元素个数关系是：

$$W_{out} = W_{in}//k + 1$$

注意：根据上面公式，如果输入元素是9或8个，输出都将是5个元素。所以反向传播时，需要获取输入元素的长度，否则无法正确复原维度。

3.2.2 反向传播

在反向传播中，我们执行与前向传播相反的操作。这意味着我们需要产生与输入相同尺寸的梯度数组。对于下采样，在前向过程中被丢弃的元素，在反向过程中对应的梯度应该为0，因为它们对损失没有贡献。因此，在反向传播过程中，我们在每个保留的梯度元素之间插入k-1个零。

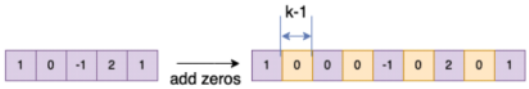


Figure 8: Downsampling1d Backward Example (k=2)

3.2.3 代码

```
class Downsample1d():
    def __init__(self, downsampling_factor):
        """
        类的构造函数，初始化下采样因子。

        参数:
            downsampling_factor (int): 下采样因子。
        """
        self.downsampling_factor = downsampling_factor

    def forward(self, A):
        """
        前向传播函数，实现下采样操作。

        参数:
            A (np.array): 输入数组，形状为(batch_size, in_channels, input_width)。

        返回:
```

```
            Z (np.array): 下采样后的数组，形状为(batch_size, in_channels, output_width)。
        """
        # 存储原始输入宽度，以便在反向传播时使用
        self.input_width = A.shape[2]

        # 使用切片操作进行下采样
        Z = A[:, :, ::self.downsampling_factor]
        return Z

    def backward(self, dLdZ):
        """
        反向传播函数，将梯度映射回下采样前的维度。

        参数:
            dLdZ (np.array): 关于下采样输出的损失梯度，形状为(batch_size, in_channels,
output_width)。

        返回:
            dLdA (np.array): 映射回原始输入尺寸的损失梯度，形状为(batch_size, in_channels,
input_width)。
        """
        batch_size, in_channels, _ = dLdZ.shape

        # 初始化一个全零的梯度数组，根据存储的原始输入宽度设置大小
        dLdA = np.zeros((batch_size, in_channels, self.input_width))

        # 将损失梯度映射回被下采样的位置
        dLdA[:, :, ::self.downsampling_factor] = dLdZ #从0开始，每隔downsampling_factor
个元素赋值为dLdZ中的元素

        return dLdA
```

3.3 上采样（2D）

跟1D同理，只是多了一个维度。

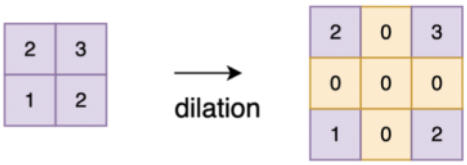


Figure 9: Upsampling 2d (k=2)

```
class Upsample2d():
    def __init__(self, upsampling_factor):
        """
        初始化Upsample2d类的实例。

        参数:
        upsampling_factor (int): 上采样因子，即在每个像素之间插入的零的个数加一（在x和y方向
上）。
        """
        self.upsampling_factor = upsampling_factor
```

```
    def forward(self, A):
        """
        对输入的图像进行上采样（膨胀）。

        参数:
        A (numpy array): 输入的图像，形状为(N, C, H_in, W_in)。

        返回:
        Z (numpy array): 上采样后的图像，形状为(N, C, H_out, W_out)。
        """

        N, C, H_in, W_in = A.shape
        H_out = (H_in-1) * self.upsampling_factor + 1
        W_out = (W_in-1) * self.upsampling_factor + 1

        # 创建一个形状为(N, C, H_out, W_out)的新数组，初始值为零
        Z = np.zeros((N, C, H_out, W_out), dtype=A.dtype)

        # 使用切片操作填充非零值
        Z[:, :, ::self.upsampling_factor, ::self.upsampling_factor] = A

        return Z

    def backward(self, dLdZ):
        """
        对上采样后的梯度图进行下采样（池化），这是上采样的逆操作。

        参数:
        dLdZ (numpy array): 关于上采样后图像的损失梯度，形状为(N, C, H_out, W_out)。

        返回:
        dLdA (numpy array): 关于输入图像的损失梯度，形状为(N, C, H_in, W_in)。
        """

        # 从上采样的梯度中提取原始像素位置的梯度
        dLdA = dLdZ[:, :, ::self.upsampling_factor, ::self.upsampling_factor]

        return dLdA
```

3.4 下采样（2D）

同下采样1D同理。

前向传播：

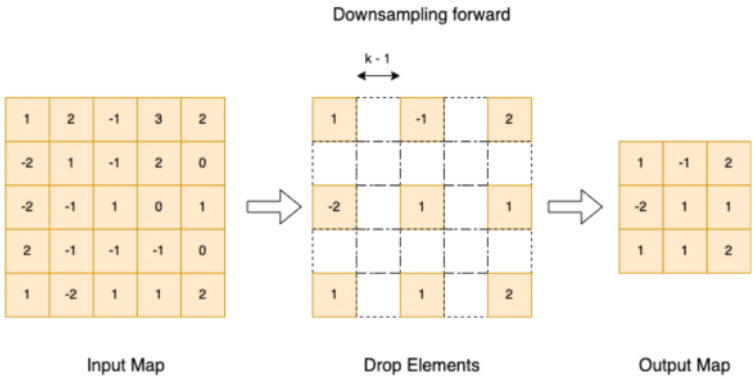


Figure 10: Downsample 2d Forward Example (k=2)

反向传播:

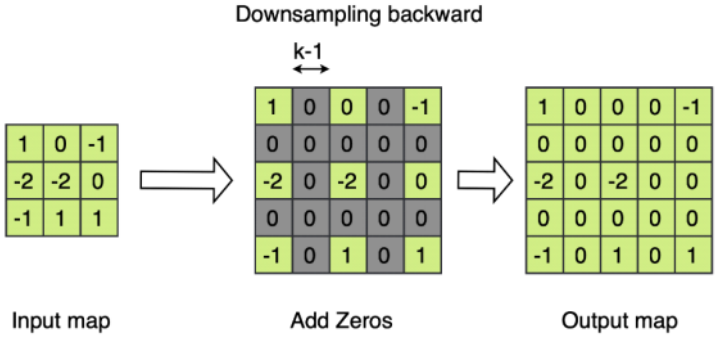


Figure 11: Downsample 2d Backward Example (k=2)

```python
class Downsample2d():

    def __init__(self, downsampling_factor):
        self.downsampling_factor = downsampling_factor

    def forward(self, A):
        """
        Argument:
            A (np.array): (batch_size, in_channels, input_height, input_width)
        Return:
            Z (np.array): (batch_size, in_channels, output_height, output_width)
        """
        # 存储原始输入宽度，以便在反向传播时使用
        self.input_width, self.input_height = A.shape[2], A.shape[3]

        # 使用切片操作进行下采样
        Z = A[:, :, ::self.downsampling_factor, ::self.downsampling_factor]
        return Z

    def backward(self, dLdZ):
        """
        Argument:
            dLdZ (np.array): (batch_size, in_channels, output_height, output_width)
        Return:
            dLdA (np.array): (batch_size, in_channels, input_height, input_width)
        """

        batch_size, in_channels, _, _ = dLdZ.shape

        # 初始化一个全零的梯度数组，根据存储的原始输入宽度设置大小
```

```
        dLdA = np.zeros((batch_size, in_channels, self.input_width, self.input_height))

        # 将损失梯度映射回被下采样的位置
        dLdA[:, :, ::self.downsampling_factor, ::self.downsampling_factor] = dLdZ # 从0
开始，每隔downsampling_factor个元素赋值为dLdZ中的元素

        return dLdA
```

4 卷积

4.1 Conv1d_stride1（1D卷积，且步长为1）

4.1.1 前向传播

当输入的通道数为1时，1D卷积效果如下：

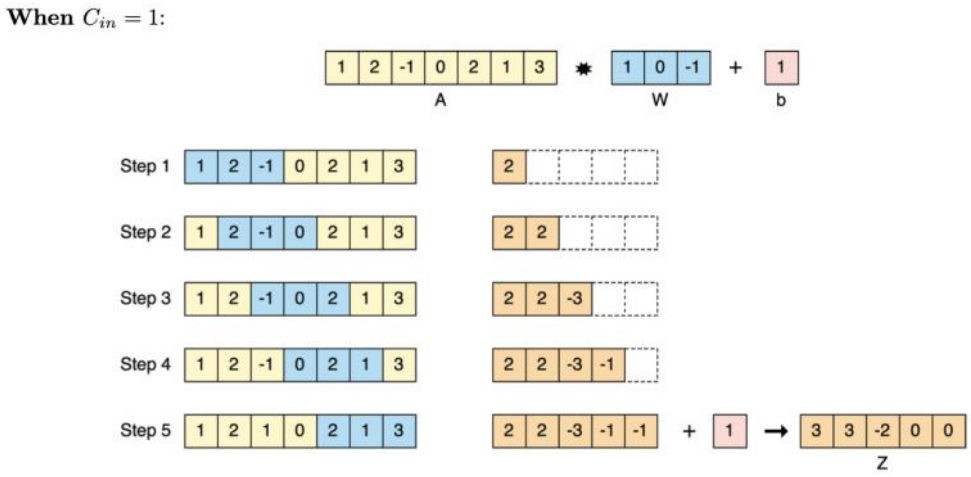


Figure 12: Conv1d with single channel input

备注：上述1D卷积的核大小=3，输出通道数=1，输入数据通道数=1。

上述的代码实现：Conv1d(in_channels=1, out_channels=1, kernel_size=3, stride=1)

当输入的通道数大于1时，1D卷积效果如下：

When $C_{in} > 1$:

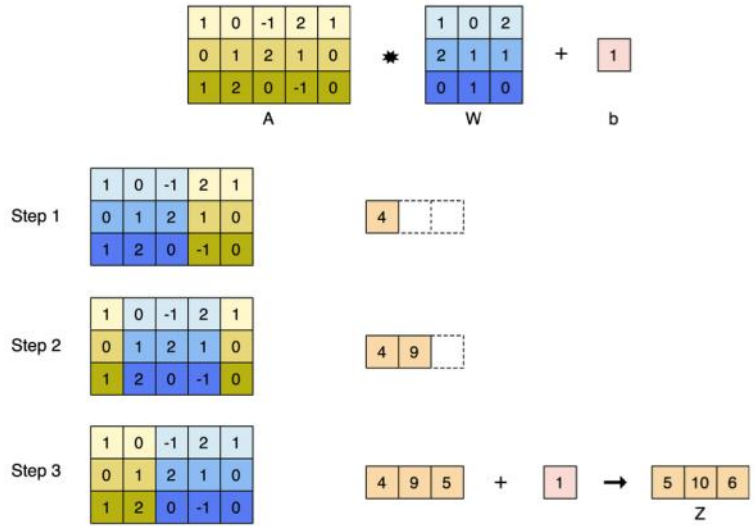


Figure 13: Conv1d with multichannel input

备注：上述1D卷积的输出都是单通道，如果要输出多通道结果，那就是用多套1D卷积核。每有1套卷积核，输出就多一个。

上述的代码实现：Conv1d(in_channels=3, out_channels=1, kernel_size=3, stride=1)

4.1.2 反向传播

在实现1维卷积的反向传播之前，先简单回顾一下反向传播在卷积层中需要计算的几个主要部分：

- **梯度相对于输出（dLdZ）**：这是损失函数相对于卷积层输出的梯度，通常由后一层（如果存在）提供。
- **梯度相对于权重（dLdW）**：这是损失函数相对于卷积层中权重的梯度，表示我们需要调整权重的方向和幅度。
- **梯度相对于偏置（dLdb）**：这是损失函数相对于卷积层偏置的梯度。
- **梯度相对于输入（dLdA）**：这是损失函数相对于卷积层输入的梯度，用于反向传播到前一层。

（1）前向传播中的卷积运算

在一维卷积的前向传播中，输出 z_i 是输入 A 与卷积核 W 的卷积结果加上偏置 b：

$$z_i = (W * A)_i + b = \sum_{j=1}^{m} w_j \cdot a_{i+j-1} + b$$

其中：

- $A = [a_1, a_2, \ldots, a_n]$ 是输入序列
- $W = [w_1, w_2, \ldots, w_m]$ 是卷积核
- $Z = [z_1, z_2, \ldots, z_{n-m+1}]$ 是输出序列
- b 是偏置项

如果我们有多个通道的输入或输出，计算过程类似，但为了简单起见，我们这里考虑单通道的情况。

（2）反向传播：计算 dL/dA

首先，我们考虑损失函数对输入 A 的梯度 dL/dA。根据链式法则，梯度可以分解为：

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

表示损失函数 L 对 A 的梯度可以通过 dL/dZ 和 dZ/dA 相乘得到。

计算 dZ/dA：

接下来，我们需要计算 dZ/dA。由于输出 Z 是卷积操作的结果，输出序列中的每个元素 z_i 是输入序列 A 中一部分元素与卷积核 W 的点积。因此，z_i 对输入序列中的每个元素 a_k 的导数 $\frac{dz_i}{dA_k}$ 只有在 a_k 参与了这个卷积计算时才非零。具体来说：

$$\frac{dL}{dA_k} = \sum_{i=1}^{n-m+1} \frac{dL}{dz_i} \cdot \frac{dz_i}{dA_k}$$

它表示输入 A 中每个元素 a_k 的梯度是所有 z_i 的梯度 dL/dz_i 和 z_i 对 a_k 的导数 dz_i/dA_k 的累加。

由于 $z_i = \sum_{j=1}^{m} w_j \cdot a_{i+j-1} + b$，所以：

$$\frac{dz_i}{dA_k} = w_{k-i+1} \text{ 当且仅当 } 1 \le k-i+1 \le m$$

使用翻转的卷积核进行卷积：

进一步我们可以得到：

$$\frac{dL}{dA_k} = \sum_{i=1}^{n-m+1} \frac{dL}{dz_i} \cdot w_{k-i+1}$$

表示损失函数对输入序列中每个元素 a_k 的梯度是所有输出梯度 dL/dz_i 与相应的卷积核元素 w_{k-i+1} 的乘积之和。

上述的累加和公式，实际上可以转化成如下公式：

$$\frac{dL}{dA} = \frac{dL}{dZ} * W_{\text{flip}}$$

其中 W_{flip} 是 W 反转后的卷积核。即 dL/dA 的计算实际上可以通过翻转的卷积核 W_{flip} 与输出梯度 dL/dZ 进行卷积运算得到的。

举例解释

假设我们有输入 $A = [a_1, a_2, a_3, a_4]$ 和卷积核 $W = [w_1, w_2]$，那么卷积层输出 Z 为：

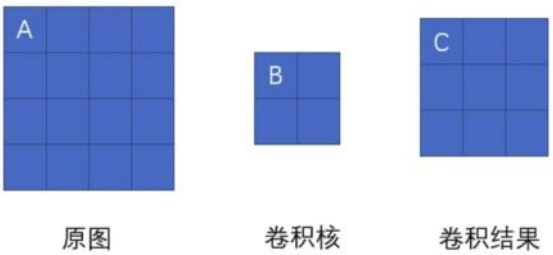
- $z_1 = w_1 \cdot a_1 + w_2 \cdot a_2 + b$
- $z_2 = w_1 \cdot a_2 + w_2 \cdot a_3 + b$
- $z_3 = w_1 \cdot a_3 + w_2 \cdot a_4 + b$

在反向传播中，我们需要计算 dL/dA：

- 对于 a_1：只有 z_1 包含 a_1，所以 $\frac{dL}{da_1} = \frac{dL}{dz_1} \cdot w_1$
- 对于 a_2：z_1 和 z_2 都包含 a_2，所以 $\frac{dL}{da_2} = \frac{dL}{dz_1} \cdot w_2 + \frac{dL}{dz_2} \cdot w_1$
- 对于 a_3：z_2 和 z_3 都包含 a_3，所以 $\frac{dL}{da_3} = \frac{dL}{dz_2} \cdot w_2 + \frac{dL}{dz_3} \cdot w_1$
- 对于 a_4：只有 z_3 包含 a_4，所以 $\frac{dL}{da_4} = \frac{dL}{dz_3} \cdot w_2$

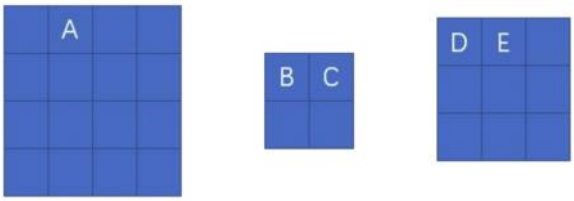
这说明对于每个位置的 a_k，其梯度实际上是 dL/dZ 和 W 的卷积。由于卷积核在每个卷积步中是从右到左应用的（即从 w_2 到 w_1），在反向传播时我们需要将卷积核 W 翻转。

备注：图形化理解



如上图所示，我们求原图A处的delta误差，就先分析，它在前向传播时影响了下一层的哪些结点。显然，它只对结点C有一个权重为B的影响，对卷积结果中的其它结点没有任何影响。因此A的delta误差应该等于C点的delta误差乘上权重B。

我们现在将原图A点位置移动一下，再看看变换位置后A点的delta误差是多少，同样先分析它前向传播影响了卷积结果的哪些结点。



经过分析，A点以权重C影响了卷积结果的D点，以权重B影响了卷积结果的E点。那它的delta误差就等于D点delta误差乘上C加上E点的delta误差乘上B。

可以尝试用相同的方法去分析原图中其它结点的delta误差，结果会发现，原图的delta误差，等于卷积结果的delta误差经过零填充后与卷积核旋转180度后的卷积。

(3) 计算 dL/dW

类似地，我们还需要计算损失函数对卷积核 W 的梯度 dL/dW。这里的梯度计算遵循同样的链式法则：

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

对于每一个 w_j，它对 z_i 的梯度是：

$$\frac{dz_i}{dw_j} = a_{i+j-1}$$

因此：

$$\frac{dL}{dw_j} = \sum_i \frac{dL}{dz_i} \cdot a_{i+j-1}$$

这实际上可以表示为 A 和 dL/dZ 之间的卷积：

$$dL/dW = A * dL/dZ$$

(4) 计算 dL/db

在卷积层中，每一个输出元素 z_i 通常都包含一个偏置项 b。如果我们有以下卷积操作：

$$z_i = (W * A)_i + b$$

其中 $(W * A)_i$ 是输入 A 与卷积核 W 的卷积结果，偏置 b 是加在每个卷积结果上的一个常数。

根据链式法则，损失函数 L 对偏置 b 的梯度 dL/db 是：

$$\frac{dL}{db} = \sum_{i=1}^{n} \frac{dL}{dz_i} \cdot \frac{dz_i}{db}$$

因为偏置 b 是一个常数，它对每个 z_i 的贡献都是相同的，即 $\frac{dz_i}{db} = 1$（这是因为 z_i 中的偏置项不依赖于 A 或 W，而是直接加上去的一个常数值）。

因此，我们可以简化上面的式子为：

$$\frac{dL}{db} = \sum_{i=1}^{n} \frac{dL}{dz_i} \cdot 1 = \sum_{i=1}^{n} \frac{dL}{dz_i}$$

这表明了 dL/db 是所有 dL/dZ 的和。

(5) 总结

通过以上推导我们得出：

- dL/dA 是通过将 dL/dZ 与翻转后的 W 卷积计算得到的。
- dL/dW 是通过将 A 与 dL/dZ 卷积计算得到的。
- dL/db 是 dL/dZ 的所有元素的和。

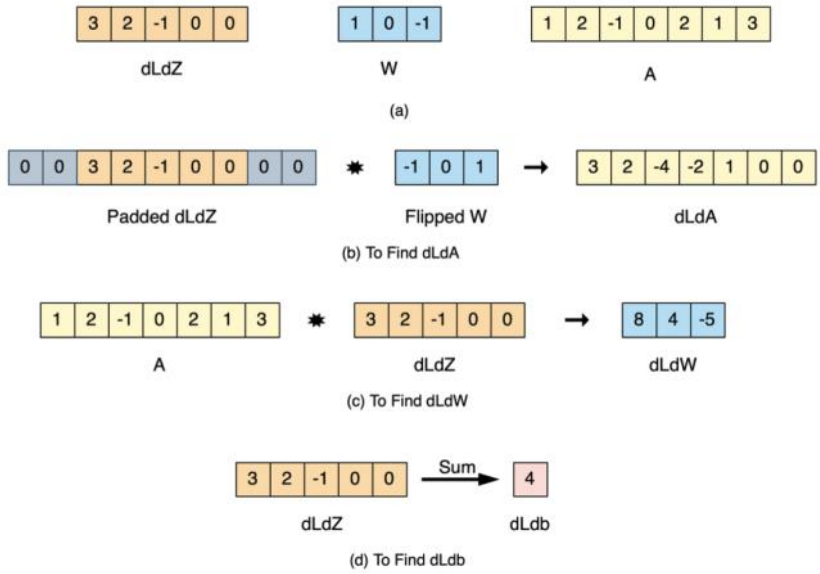


Figure 14: Conv1d backward with single channel input

当输入是多通道情况：

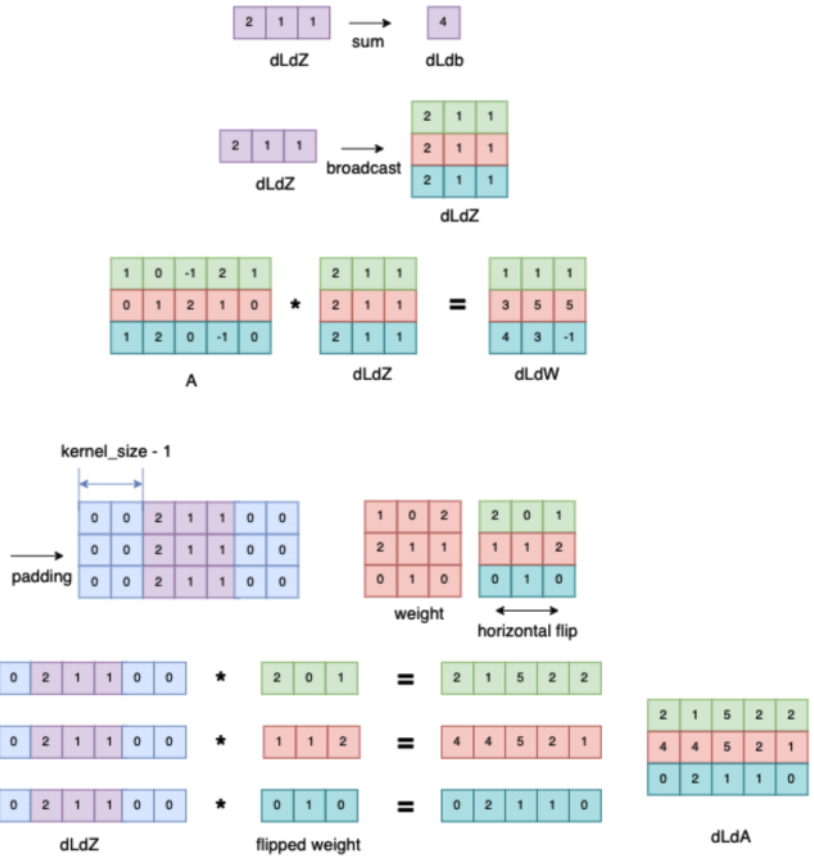


Figure 15: Conv1d_stride1 Multichannel Backward Example

4.1.3 代码实现

```
class Conv1d_stride1():
    def __init__(self, in_channels, out_channels, kernel_size, weight_init_fn=None,
bias_init_fn=None):
        self.in_channels = in_channels # 输入通道数
        self.out_channels = out_channels # 输出通道数
```

```
        self.kernel_size = kernel_size  # 卷积核大小

        # 权重和偏置的初始化
        # self.W的维度：(out_channels, in_channels, kernel_size)
        # out_channels 是输出通道数（与卷积核的个数有关）
        # in_channels 是输入通道数（与输入数据的通道数相对应）
        if weight_init_fn is None:
            # 使用正态分布初始化权重
            self.W = np.random.normal(0, 1.0, (out_channels, in_channels, kernel_size))
        else:
            # 使用自定义的初始化函数初始化权重
            self.W = weight_init_fn(out_channels, in_channels, kernel_size)

        if bias_init_fn is None:
            self.b = np.zeros(out_channels)
        else:
            self.b = bias_init_fn(out_channels)

        # 初始化权重和偏置的梯度
        self.dLdW = np.zeros(self.W.shape)
        self.dLdb = np.zeros(self.b.shape)

    def forward(self, A):
        """
        输入参数:
            A : (batch_size, in_channels, input_length)
                        input_length: 输入张量的长度
                        in_channels: 输入张量的通道数

        返回参数:
            Z : (batch_size, out_channels, output_length)
        """

        self.A = A # 保存输入张量，以便在反向传播时使用
        batch_size, _, input_length = A.shape
        output_length = input_length - self.kernel_size + 1

        # 初始化输出张量
        Z = np.zeros((batch_size, self.out_channels, output_length))

        # 计算卷积，
        # 方法1
        for i in range(output_length):
            # np.tensordot(): 这个函数计算沿着指定轴的张量点积。
            # 在这里，它执行了被切片的输入张量 A[:, :, i:i+self.kernel_size] 和卷积核
self.W 的点积。
            # axes 参数 ([1, 2], [1, 2]) 表示点积沿着 A 的第2、3维度（in_channels,
input_length），以及 W 的第2、3维度（in_channels, kernel_size）进行。
            Z[:, :, i] = np.tensordot(A[:, :, i:i+self.kernel_size], self.W, axes=([1,
2], [1, 2])) + self.b

        # 方法2
        for n in range(Z.shape[0]): # 遍历batchsize
            for c in range(Z.shape[1]): # 遍历输出通道数
                for w in range(Z.shape[2]): # 遍历output_length
                    Z[n, c, w] = np.sum(A[n, :, w:w+self.kernel_size] * self.W[c,:,:])
+ self.b[c]
```

```
        return Z


    def backward(self, dLdZ):
        """ 参数
                dLdZ :  (batch_size, out_channels, output_length)
                   A :  (batch_size, in_channels, input_length)
                dLdW :  (out_channels, in_channels, kernel_size)
            返回:
                dLdA :  (batch_size, in_channels, input_length)"""

        _, _, output_length = dLdZ.shape

        #  (1) 计算dLdW
        # 方法1
        for i in range(self.kernel_size):
            # axes参数([0, 2], [0, 2])表示点积沿着dLdZ和A的第一个和第三个维度进行。
            self.dLdW[:, :, i] = np.tensordot(dLdZ, self.A[:, :, i:i+output_length],
axes=([0, 2], [0, 2]))

        # 方法2
        for o in range(self.out_channels): #遍历卷积核个数
            for i in range(self.in_channels): #遍历卷积核的通道数（与输入数据的通道数相同）
                for k in range(self.kernel_size): #遍历卷积核长度
                    self.dLdW[o, i, k] = np.sum(self.A[:, i, k:k+output_length] *
dLdZ[:, o, :])


        #  (2) 计算dLdb
        self.dLdb = np.sum(dLdZ, axis=(0, 2))


        # #  (3) 计算dLdA
        # # 方法1
        # dLdA = np.zeros_like(self.A)
        # for i in range(output_length): #遍历输出特征图的每一个元素位置。
        #     dLdA[:, :, i:i+self.kernel_size] += np.tensordot(dLdZ[:, :, i], self.W,
axes=([1], [0]))

        # # # 方法2
        # dLdA = np.zeros(self.A.shape)
        # flipped_W = np.flip(self.W, axis=2) #将卷积核 W 在最后一个维度上（即宽度方向）进
行翻转
        # # 在宽度方向上前、后各填充 self.kernel_size - 1 个0，保持其他维度不变。
        # # 填充操作是为了确保反卷积操作能够覆盖输入 A 的所有位置。卷积核大小为
self.kernel_size,
        # # 所以需要在 dL/dZ 的两边进行适当的填充。
        # padded_dLdZ = np.pad(dLdZ, ((0,0), (0,0), (self.kernel_size-1,
self.kernel_size-1)), 'constant')
        # for n in range(dLdA.shape[0]): # 遍历输入数据的【batchsize】
        #     for c in range(dLdA.shape[1]): # 遍历输入数据的【通道数】（与卷积核的通道数相
同）
        #         for w in range(dLdA.shape[2]): # 遍历输入数据的【宽度】（即input
length）
        #             dLdA[n,c,w] = dLdA[n,c,w] + np.sum(padded_dLdZ[n, :,
w:w+self.kernel_size] * flipped_W[:,c,:])
```

```
        # 方法3（本方法对应讲义中【图14】的计算过程）
        dLdA = np.zeros(self.A.shape)
        dLdZ_pad = np.pad(dLdZ, pad_width=((0, 0), (0, 0), (self.kernel_size-1,
self.kernel_size-1)), mode='constant', constant_values=(0, 0))
        W_flipped = np.flip(self.W, axis=2)

        for i in range(dLdA.shape[2]): # 遍历输入数据的【宽度】（即input length）
            # 取出 dLdZ_pad 中从第 i 个位置开始，宽度为 self.kernel_size 的片段，作为一个局
部区域进行计算
            section = dLdZ_pad[:, :, i:i+self.kernel_size]
            # 对局部区域 section 与翻转后的卷积核 W_flipped 进行张量点积，计算这个局部区域的
梯度
            # axes=([1,2], [0,2]) 表示将 section 的第 1, 2 维与 W_flipped 的第 0, 2 维进
行点积
            result = np.tensordot(section, W_flipped, axes=([1,2], [0,2]))
            # 将计算得到的梯度 result 存储到 dLdA 的对应位置
            dLdA[:,:,i] = result

        return dLdA
```

4.2 Conv1d（1D卷积，任意步长）

步长大于1，具有下采样功能。

我们可以通过将步长为1的卷积（Conv1d_stride1）与下采样（Downsample1d）相结合来实现任意步长的1D卷积操作。这是一种分解卷积操作的方式，允许我们使用标准的步长为1的卷积层来处理输入，然后通过下采样来减少输出的尺寸，从而模拟更大步长的效果。以下是详细的解释：

- **卷积操作**：首先，输入数据A通过步长为1的1D卷积层（Conv1d_stride1）。这个卷积层将每个输入通道的每个段与卷积核进行卷积，输出的每个通道包含了输入数据的一个压缩表示。由于步长为1，卷积输出的宽度会比输入数据的宽度小，小的数量取决于卷积核的大小。这个操作不会改变数据的深度（即输出通道数）。
- **下采样操作**：卷积层的输出随后被送入下采样层（Downsample1d），这个层按照指定的步长stride减少序列的长度。例如，如果步长为2，那么下采样层会每两个元素取一个，从而减少输出序列的长度。

这种组合使得Conv1d能够在不直接实现任意步长的卷积的情况下模拟步长大于1的1D卷积。

```
class Conv1d():
    def __init__(self, in_channels, out_channels, kernel_size, stride,
                 weight_init_fn=None, bias_init_fn=None):
        # 初始化步长
        self.stride = stride

        # 初始化Conv1d_stride1实例和Downsample1d实例
        self.conv1d_stride1 = Conv1d_stride1(in_channels, out_channels, kernel_size,
weight_init_fn, bias_init_fn)
        self.downsample1d = Downsample1d(stride)

    def forward(self, A):
        """
        参数:
            A (np.array): (批量大小, 输入通道数, 输入尺寸)
        返回:
            Z (np.array): (批量大小, 输出通道数, 输出尺寸)
        """
        # 调用步长为1的1D卷积层
```

```
        con_A = self.conv1d_stride1.forward(A)

        # 执行下采样操作
        Z = self.downsample1d.forward(con_A)

        return Z

    def backward(self, dLdZ):
        """
        参数:
            dLdZ (np.array): (批量大小, 输出通道数, 输出尺寸)
        返回:
            dLdA (np.array): (批量大小, 输入通道数, 输入尺寸)
        """
        # 调用下采样层的反向传播方法
        down_dLdZ = self.downsample1d.backward(dLdZ)

        # 调用步长为1的1D卷积层的反向传播方法
        dLdA = self.conv1d_stride1.backward(down_dLdZ)

        return dLdA
```

4.3 Conv2d_stride1（2D卷积，步长 = 1）

4.3.1 前向传播

当输入通道数为1时，卷积计算过程如下：

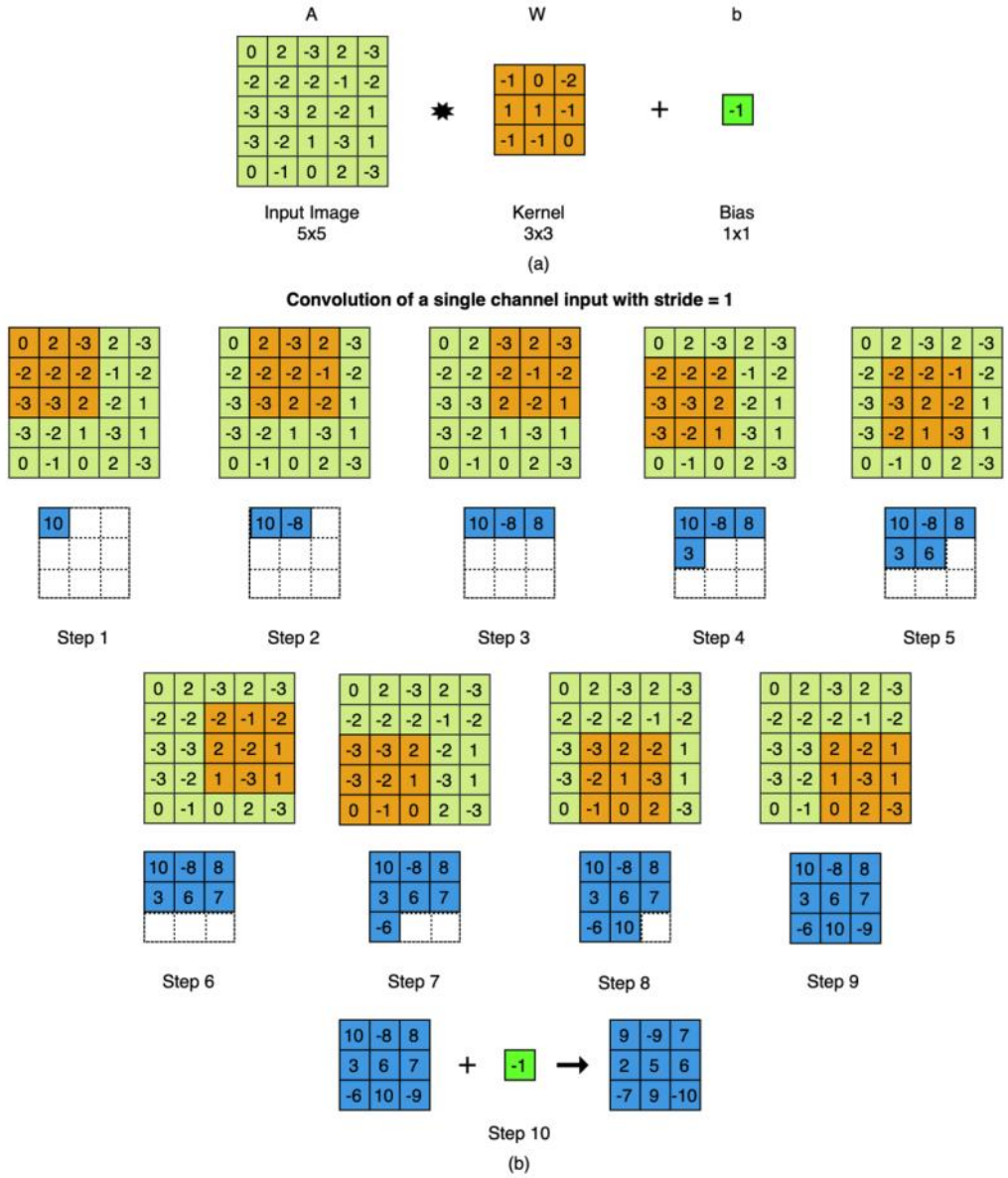


Figure 16: 2d Convolution Example stride=1 $C_{in} = 1$

当输入通道数=2，步长=2时，卷积计算过程如下：（3个卷积核，所以输出通道会变成3）

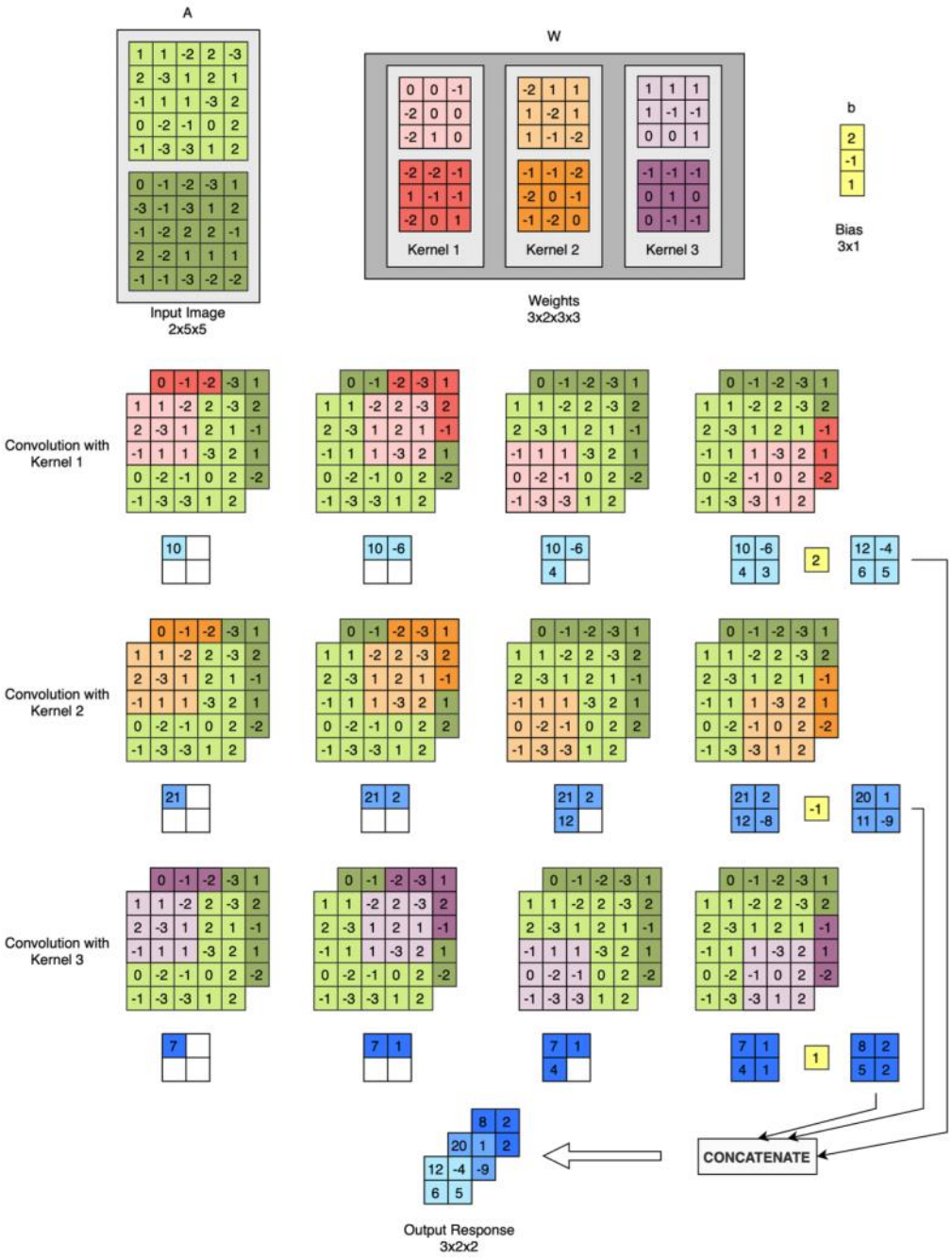


Figure 17: Multi channel convolution example $C_{in} = 2$

4.3.2 反向传播

原理跟1D卷积反向传播基本一致。

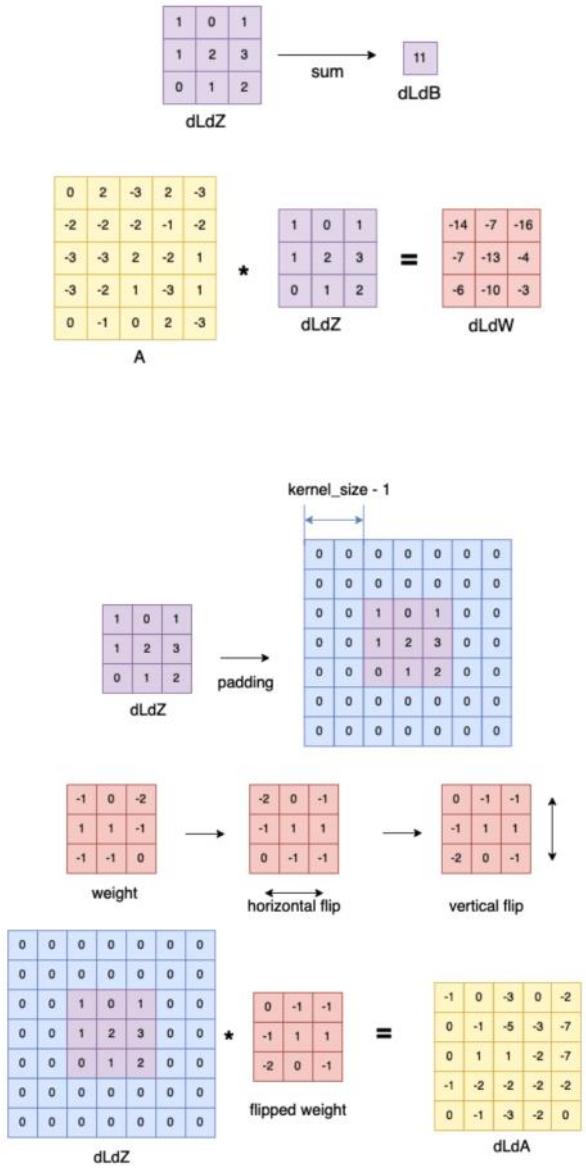


Figure 18: Conv2d_stride1 Backward: Convolution

4.3.3 代码

```
class Conv2d_stride1():
    def __init__(self, in_channels, out_channels, kernel_size, weight_init_fn=None,
bias_init_fn=None):
        # 构造函数初始化2D卷积层
        self.in_channels = in_channels  # 输入通道数
        self.out_channels = out_channels  # 输出通道数
        self.kernel_size = kernel_size  # 卷积核尺寸

        # 权重初始化
        if weight_init_fn is None:
            self.W = np.random.normal(0, 1.0, (out_channels, in_channels, kernel_size,
kernel_size))
        else:
            self.W = weight_init_fn(out_channels, in_channels, kernel_size,
kernel_size)
```

```
        # 偏置初始化
        if bias_init_fn is None:
            self.b = np.zeros(out_channels)
        else:
            self.b = bias_init_fn(out_channels)

        self.dLdW = np.zeros(self.W.shape)  # 用于存储权重的梯度
        self.dLdb = np.zeros(self.b.shape)  # 用于存储偏置的梯度

    def forward(self, A):
        """
        前向传播函数
        参数:
            A (np.array): 输入数据，形状为(batch_size, in_channels, input_height,
input_width)
        返回:
            Z (np.array): 卷积操作后的输出，形状为(batch_size, out_channels,
output_height, output_width)
        """
        batch_size, _, input_height, input_width = A.shape
        output_height = input_height - self.kernel_size + 1
        output_width = input_width - self.kernel_size + 1

        # 初始化输出Z
        Z = np.zeros((batch_size, self.out_channels, output_height, output_width))

        # 进行卷积操作
        for i in range(output_height):
            for j in range(output_width):
                # 提取输入数据的局部区域
                patch = A[:, :, i:i+self.kernel_size, j:j+self.kernel_size]
                # 在局部区域和卷积核之间进行元素乘法操作，并对结果求和
                for k in range(self.out_channels):
                    Z[:, k, i, j] = np.sum(patch * self.W[k, :, :, :], axis=(1, 2, 3))
# axis=(1, 2, 3)表示沿着(通道数，高度，宽度)方向求和

        # 添加偏置
        Z += self.b.reshape((1, -1, 1, 1))

        self.A = A  # 保存输入数据用于反向传播
        return Z

    def backward(self, dLdZ):
        """
        反向传播函数
        参数:
            dLdZ (np.array): 损失对输出Z的梯度，形状为(batch_size, out_channels,
output_height, output_width)
        返回:
            dLdA (np.array): 损失对输入A的梯度，形状为(batch_size, in_channels,
input_height, input_width)
        """
        # 初始化梯度数组
        self.dLdW.fill(0)
        self.dLdb.fill(0)
        batch_size, _, input_height, input_width = self.A.shape
        _, _, output_height, output_width = dLdZ.shape
```

```
        # 计算dLdA
        dLdA = np.zeros_like(self.A)
        for i in range(output_height):
            for j in range(output_width):
                for k in range(self.out_channels):
                    dLdA[:, :, i:i+self.kernel_size, j:j+self.kernel_size] += dLdZ[:,
k, i, j][:, None, None, None] * self.W[k, :, :, :]

        # 计算dLdW
        for k in range(self.out_channels):
            for i in range(output_height):
                for j in range(output_width):
                    self.dLdW[k, :, :, :] += np.sum(dLdZ[:, k, i, j][:, None, None,
None] * self.A[:, :, i:i+self.kernel_size, j:j+self.kernel_size], axis=0)

        # 计算dLdb
        self.dLdb = np.sum(dLdZ, axis=(0, 2, 3))

        return dLdA
```

4.4 Conv2d（2D卷积，任意步长）

2D卷积的前向传播，可以先求步长为1的2D卷积结果，然后对结果进行下采样。

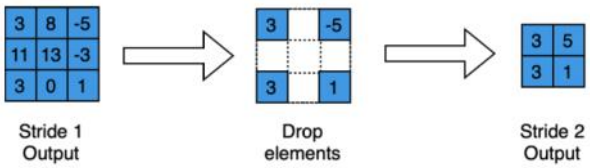


Figure 19: Stride 2 convolution as stride 1 convolution and downsampling with factor = 2

对于反向传播：

顺序要颠倒过来，先求下采样的反向传播结果，然后对结果求步长为1的2D卷积反向传播结果。

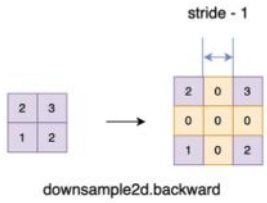
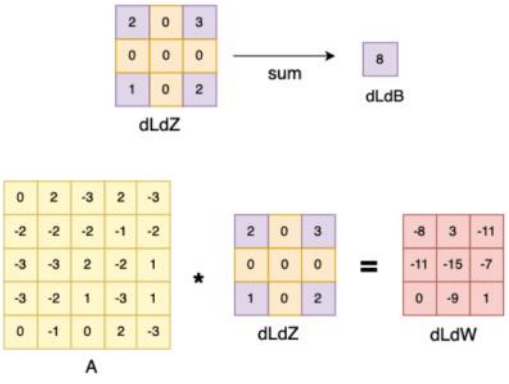


Figure 20: Downsample2d Backward: Conv2d Example



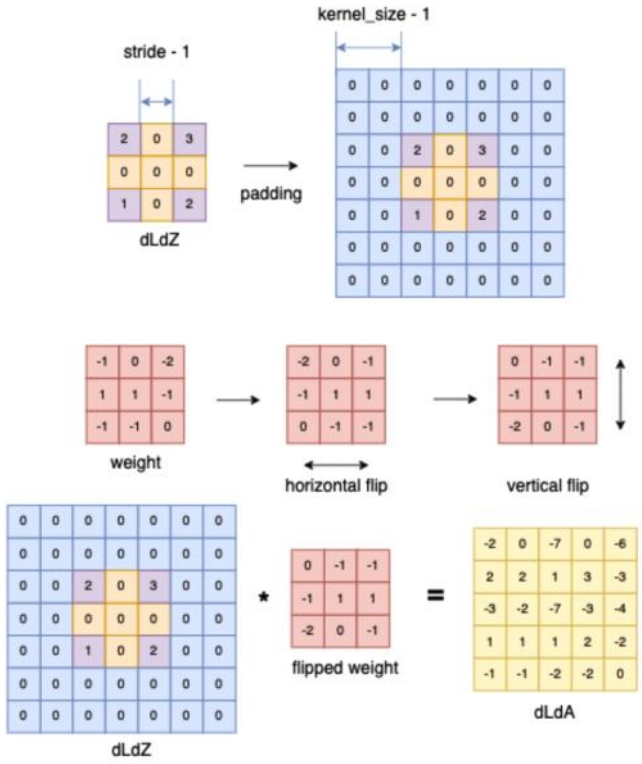


Figure 21: Conv2d Backward Example (stride ¿ 1)

代码如下：（S23版本作业不要求带padding）

```
class Conv2d():
    def __init__(self, in_channels, out_channels, kernel_size, stride,
                 weight_init_fn=None, bias_init_fn=None):
        # 初始化2维卷积层，可以接受任意步长
        self.stride = stride

        self.conv2d_stride1 = Conv2d_stride1(in_channels, out_channels,
kernel_size,weight_init_fn, bias_init_fn)
        self.downsample2d = Downsample2d(stride)  # TODO

    def forward(self, A):
        """
        参数:
            A (np.array): 输入数据，形状为(batch_size, in_channels, input_height,
input_width)
        返回:
            Z (np.array): 卷积操作后的输出，形状为(batch_size, out_channels,
output_height, output_width)
        """
        # 首先执行步长为1的2维卷积
        conv_A = self.conv2d_stride1.forward(A)

        # 然后进行下采样以实现所需的步长
        Z = self.downsample2d.forward(conv_A)

        return Z

    def backward(self, dLdZ):
        """
        参数:
            dLdZ (np.array): 对输出Z的损失梯度，形状为(batch_size, out_channels,
```

```
output_height, output_width)
        返回:
            dLdA (np.array): 对输入A的损失梯度，形状为(batch_size, in_channels,
input_height, input_width)
        """

        # 这个顺序很重要，首先处理下采样层的梯度，然后再处理卷积层的梯度

        # 首先处理下采样层的梯度
        down_dLdZ = self.downsample2d.backward(dLdZ)

        # 将梯度传递给步长为1的卷积层
        dLdA = self.conv2d_stride1.backward(down_dLdZ)

        return dLdA
```

带padding版本的2D 卷积：

```
class Conv2d_padding():
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding=0,
                 weight_init_fn=None, bias_init_fn=None):
        # 初始化2维卷积层，可以接受任意步长
        self.stride = stride
        self.pad = padding

        self.conv2d_stride1 = Conv2d_stride1(in_channels, out_channels, kernel_size, weight_init_fn,
bias_init_fn)
        self.downsample2d = Downsample2d(stride)

    def forward(self, A):
        """
        参数:
            A (np.array): 输入数据，形状为(batch_size, in_channels, input_height, input_width)
        返回:
            Z (np.array): 卷积操作后的输出，形状为(batch_size, out_channels, output_height, output_width)
        """

        # 在输入张量 A 的高度和宽度两个维度上进行零填充，填充的宽度为 self.pad
        # 假设 A 的形状为 (10, 3, 32, 32)，self.pad = 1，那么执行这行代码后，
        # A 的形状将变为 (10, 3, 34, 34)，即在每个输入图像的四周各添加了一圈零填充。
        A = np.pad(A, ((0, 0), (0, 0), (self.pad, self.pad), (self.pad, self.pad)), mode='constant')

        # 首先执行步长为1的2维卷积
        conv_A = self.conv2d_stride1.forward(A)

        # 然后进行下采样以实现所需的步长
        Z = self.downsample2d.forward(conv_A)

        return Z

    def backward(self, dLdZ):
        # 首先处理下采样层的梯度
        down_dLdZ = self.downsample2d.backward(dLdZ)

        # 将梯度传递给步长为1的卷积层
        dLdA = self.conv2d_stride1.backward(down_dLdZ)

        if self.pad > 0:
```

```
            dLdA = dLdA[:, :, self.pad:-self.pad, self.pad:-self.pad]

        return dLdA
```

5 转置卷积

转置卷积的功能：对输入数据进行上采样（即尺度变大）。

实现方法：即先通过上采样（upsampling）增加输入的尺寸，然后应用常规的卷积（这里我们使用步长为1的卷积）

1. **填充**：在输入特征图的元素之间插入空行和空列（通常是0）。插入的数量取决于希望扩大的倍数。
2. **卷积**：使用标准卷积步骤（步长为1）应用卷积核。

转置卷积的前向传播和反向传播示意图如下：

1D转置卷积：

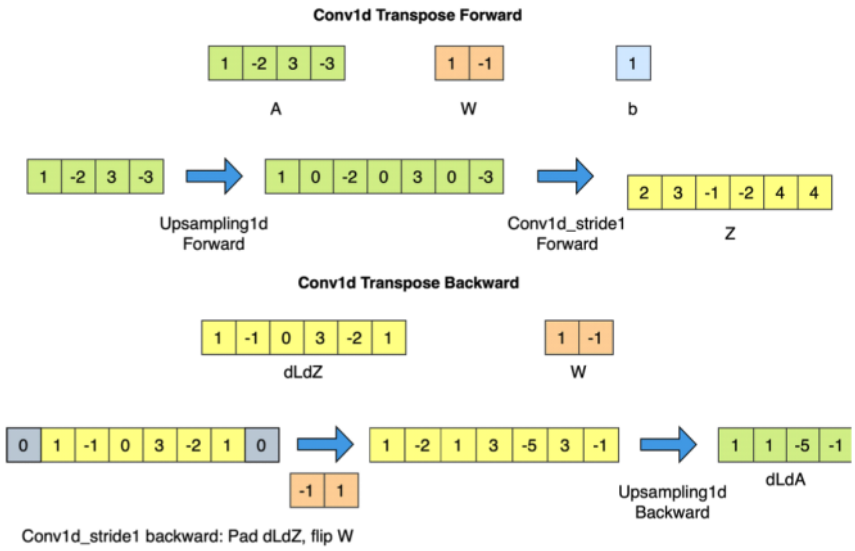


Figure 22: 1d Transpose Convolution

2D转置卷积：

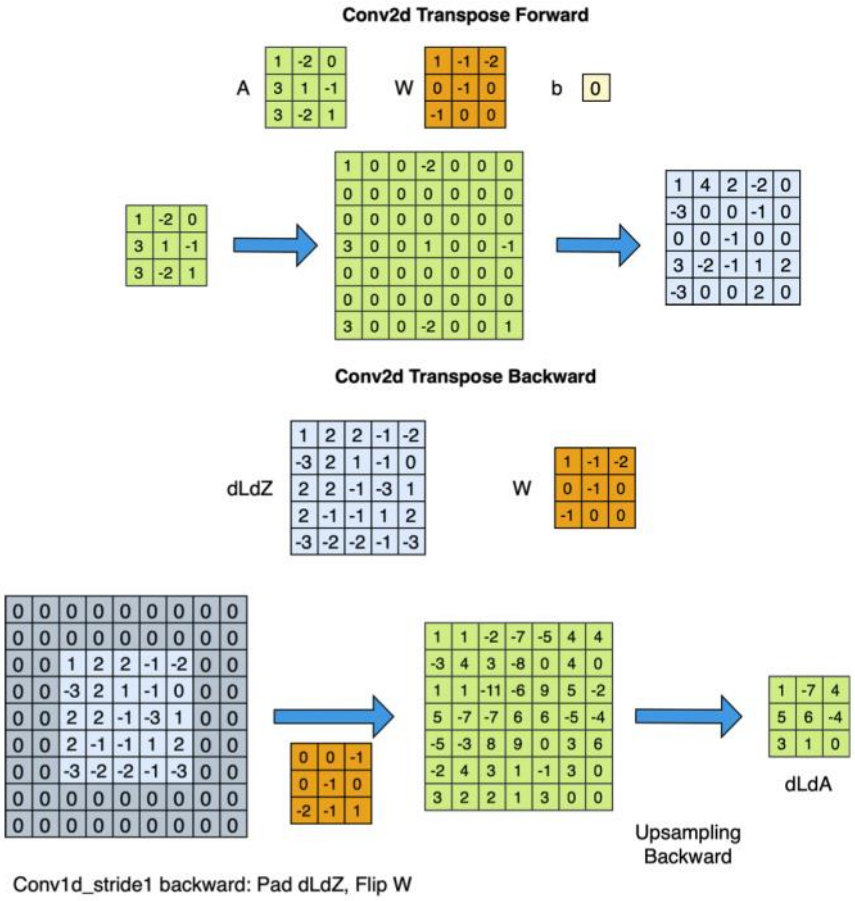


Figure 23: 2d Transpose Convolution

代码如下：

```
import numpy as np
from resampling import *
from Conv1d import *
from Conv2d import *


class ConvTranspose1d():
    """
    实现一维转置卷积类。
    """
    def __init__(self, in_channels, out_channels, kernel_size, upsampling_factor,
                 weight_init_fn=None, bias_init_fn=None):
        """
        初始化一维转置卷积层。
        :param in_channels: 输入通道数。
        :param out_channels: 输出通道数。
        :param kernel_size: 卷积核大小。
        :param upsampling_factor: 上采样因子。
        :param weight_init_fn: 权重初始化函数。
        :param bias_init_fn: 偏置初始化函数。
        """
        self.upsampling_factor = upsampling_factor  # 设置上采样因子

        # 初始化一维上采样和一维卷积实例，这里假设Upsample1d和Conv1d_stride1类已经实现
        self.upsample1d = Upsample1d(upsampling_factor)
        self.conv1d_stride1 = Conv1d_stride1(in_channels, out_channels, kernel_size,
                                             weight_init_fn, bias_init_fn)
```

```
    def forward(self, A):
        """
        前向传播过程。
        :param A: 输入数组，维度为(batch_size, in_channels, input_size)。
        :return: 输出数组，维度为(batch_size, out_channels, output_size)。
        """
        # 上采样输入
        A_upsampled = self.upsample1d.forward(A)

        # 应用步长为1的一维卷积
        Z = self.conv1d_stride1.forward(A_upsampled)

        return Z

    def backward(self, dLdZ):
        """
        反向传播过程。
        :param dLdZ: 损失关于输出Z的梯度。
        :return: 损失关于输入A的梯度。
        """
        # 首先反向传播通过步长为1的一维卷积层
        delta_out = self.conv1d_stride1.backward(dLdZ)

        # 然后反向传播通过上采样层
        dLdA = self.upsample1d.backward(delta_out)

        return dLdA



class ConvTranspose2d():
    def __init__(self, in_channels, out_channels, kernel_size, upsampling_factor,
                 weight_init_fn=None, bias_init_fn=None):
        """
        二维转置卷积的初始化方法。
        :param in_channels: 输入的通道数。
        :param out_channels: 输出的通道数。
        :param kernel_size: 卷积核的大小，可以是一个整数或一个由两个整数构成的元组。
        :param upsampling_factor: 上采样因子，即输出相对于输入在空间维度上的放大倍数。
        :param weight_init_fn: 权重初始化函数，用于初始化卷积核的权重。
        :param bias_init_fn: 偏置初始化函数，用于初始化卷积核的偏置。
        """
        # 存储上采样因子
        self.upsampling_factor = upsampling_factor

        # 初始化二维卷积实例，这里Conv2d_stride1类应该实现了常规的二维卷积操作，其中步长被设
置为1。
        # 此卷积用于在上采样后的数据上应用。
        self.conv2d_stride1 = Conv2d_stride1(in_channels, out_channels, kernel_size,
                                             weight_init_fn, bias_init_fn)
        # 初始化上采样实例，Upsample2d类应该实现了二维数据的上采样操作，按照指定的上采样因子放
大数据。
        self.upsample2d = Upsample2d(upsampling_factor)

    def forward(self, A):
        """
        对输入数据执行前向传播。
```

```
        :param A: 输入的numpy数组，其形状应该是(batch_size, in_channels, height, width)。
        :return: 输出的numpy数组，其形状是(batch_size, out_channels, new_height,
new_width)，其中
                 new_height和new_width由上采样因子和卷积核大小共同决定。
        """
        # 首先对输入数据A执行上采样操作，得到上采样后的数据A_upsampled。
        A_upsampled = self.upsample2d.forward(A)

        # 然后在上采样后的数据A_upsampled上执行步长为1的二维卷积操作，得到最终的输出Z。
        Z = self.conv2d_stride1.forward(A_upsampled)

        return Z

    def backward(self, dLdZ):
        """
        对损失关于输出Z的梯度执行反向传播。
        :param dLdZ: 损失关于输出Z的梯度，其形状应该与forward方法的输出Z相同。
        :return: 损失关于输入A的梯度dLdA，其形状与输入A相同。
        """
        # 首先对损失梯度dLdZ执行二维卷积的反向传播操作，得到经过卷积层反向传播后的梯度
delta_out。
        delta_out = self.conv2d_stride1.backward(dLdZ)

        # 然后对delta_out执行上采样的反向传播操作，得到最终的损失关于输入A的梯度dLdA。
        dLdA = self.upsample2d.backward(delta_out)

        return dLdA
```

6 池化（最大池化、平均池化）

池化层（Pooling layer）在神经网络中主要用于减少数据的空间大小（降维），以减少计算量和防止过拟合。池化操作有多种，如最大池化（Max Pooling）和平均池化（Mean/Average Pooling）。尽管池化层没有学习参数（如权重和偏置），但它们在训练过程中仍然需要进行反向传播，以允许损失函数关于输入数据的梯度通过网络反向传播，进而更新前面层的参数。

6.1 池化层的反向传播原理

最大池化（Max Pooling）的反向传播

- **原理**: 最大池化层的前向传播会从输入的子区域（窗口）中选择最大的元素进行输出。在反向传播时，损失相对于该层输出的梯度（即上游梯度）只会回传给在前向传播中被选为最大值的输入元素，而其他元素的梯度将被置为0。简单来说，只有对应于每个池化窗口最大值的输入位置会接收到梯度，其他位置的梯度为0。
- **步骤**:
  1. 在前向传播中记录每个池化窗口中最大元素的位置。
  2. 在反向传播时，将损失函数的梯度只传递给这些记录位置的元素，其他位置填充0。

平均池化（Mean Pooling）的反向传播

- **原理**: 平均池化层的前向传播会计算输入的子区域（窗口）中所有元素的平均值。在反向传播时，损失相对于该层输出的梯度会均匀地分配给输入的子区域中的每个元素。这意味着每个输入元素接收到的梯度是该池化窗口对应的输出梯度除以窗口内元素的数量。
- **步骤**:
  1. 无需记录特定位置的信息。

2. 在反向传播时，将上游的梯度均匀分配到对应池化窗口的每个输入元素上。

6.2 代码

```
import numpy as np
from resampling import *


class MaxPool2d_stride1():
    def __init__(self, kernel):
        """
        初始化一个最大池化层。
        :param kernel: 池化核的大小，此处假设核为正方形。
        """
        self.kernel = kernel

    def forward(self, A):
        """
        前向传播函数：执行最大池化操作。
        :param A: 输入数据，形状为(batch_size, in_channels, input_width, input_height)。
        :return: 池化后的结果，形状为(batch_size, in_channels, output_width,
output_height)。
        """
        self.A = A
        self.batch_size, self.in_channels, self.input_width, self.input_height =
A.shape
        self.out_width = self.input_width - self.kernel + 1
        self.out_height = self.input_height - self.kernel + 1

        # 初始化输出矩阵
        Z = np.zeros((self.batch_size, self.in_channels, self.out_width,
self.out_height))

        # 存储最大值位置的索引
        self.max_indices = np.zeros((self.batch_size, self.in_channels, self.out_width,
self.out_height, 2), dtype=int)

        for n in range(self.batch_size):
            for c in range(self.in_channels):
                for i in range(self.out_width):
                    for j in range(self.out_height):
                        window = A[n, c, i:i+self.kernel, j:j+self.kernel]
                        max_val = np.max(window)
                        Z[n, c, i, j] = max_val
                        # 找到最大值的位置并存储
                        max_pos = np.unravel_index(np.argmax(window, axis=None),
window.shape)
                        self.max_indices[n, c, i, j] = (i + max_pos[0], j + max_pos[1])

        return Z

    def backward(self, dLdZ):
        """
        反向传播函数：根据最大值的位置将梯度回传。
        :param dLdZ: 损失关于池化层输出的梯度，形状为(batch_size, in_channels,
output_width, output_height)。
        :return: 损失关于池化层输入的梯度，形状与输入A相同。
```

```
        """
        dLdA = np.zeros_like(self.A)

        for n in range(self.batch_size):
            for c in range(self.in_channels):
                for i in range(self.out_width):
                    for j in range(self.out_height):
                        # 获取最大值位置的索引
                        (max_i, max_j) = self.max_indices[n, c, i, j]
                        # 只有最大值位置的梯度非零
                        dLdA[n, c, max_i, max_j] += dLdZ[n, c, i, j]

        return dLdA



class MeanPool2d_stride1:
    """
    前向传播（forward 方法）
    1 获取输入尺寸：从输入A中获取批量大小（batch_size）、通道数（in_channels）、输入宽度（input_width）和输入高度（input_height）。
    2 计算输出尺寸：根据内核大小（kernel）和步长（stride=1）计算输出宽度（output_width）和输出高度（output_height）。
    3 初始化输出张量：根据计算出的输出尺寸初始化输出张量Z。
    4 执行均值池化：对于每个位置，计算覆盖的区域内元素的平均值，并将这个平均值赋值给输出张量的相应位置。

    反向传播（backward 方法）
    1 获取梯度尺寸：从dLdZ中获取批量大小、通道数、输出宽度和输出高度。
    2 初始化输入梯度张量：初始化一个与输入A同样尺寸的张量dLdA，用于存放传回输入的梯度。
    3 计算输入梯度：对于输出的每个梯度值，将其等分散布到对应的输入区域内的每个元素上。
    """
    def __init__(self, kernel):
        # 内核大小
        self.kernel = kernel

    def forward(self, A):
        batch_size, in_channels, input_width, input_height = A.shape
        output_width = input_width - self.kernel + 1
        output_height = input_height - self.kernel + 1

        # 初始化输出张量
        Z = np.zeros((batch_size, in_channels, output_width, output_height))

        # 执行均值池化
        for i in range(output_width):
            for j in range(output_height):
                patch = A[:, :, i:i+self.kernel, j:j+self.kernel]
                Z[:, :, i, j] = np.mean(patch, axis=(2, 3))

        return Z

    def backward(self, dLdZ):
        batch_size, out_channels, output_width, output_height = dLdZ.shape
        dLdA = np.zeros((batch_size, out_channels, output_width + self.kernel - 1, output_height + self.kernel - 1))

        # 计算输入的梯度
```

```
        for i in range(output_width):
            for j in range(output_height):
                dLdA[:, :, i:i+self.kernel, j:j+self.kernel] += dLdZ[:, :, i, j][:, :,
None, None] / (self.kernel * self.kernel)

        return dLdA



class MaxPool2d():
    def __init__(self, kernel, stride):
        self.kernel = kernel
        self.stride = stride

        # 假设存在MaxPool2d_stride1和downsample2d，并进行初始化
        self.maxpool2d_stride1 = MaxPool2d_stride1(kernel)
        self.downsample2d = Downsample2d(stride)

    def forward(self, A):
        """
        使用步长为1的最大池化，然后下采样到期望的步长
        """
        # 首先进行步长为1的最大池化
        Z = self.maxpool2d_stride1.forward(A)

        # 根据stride进行下采样
        Z = self.downsample2d.forward(Z)

        return Z

    def backward(self, dLdZ):
        """
        反向传播同样需要考虑下采样和最大池化的梯度传递
        """
        # 首先根据stride对梯度进行上采样
        dLdZ_upsampled = self.downsample2d.backward(dLdZ)

        # 然后将上采样的梯度通过步长为1的最大池化的反向传播
        dLdA = self.maxpool2d_stride1.backward(dLdZ_upsampled)

        return dLdA



class MeanPool2d:
    def __init__(self, kernel, stride):
        self.kernel = kernel
        self.stride = stride

        # 假设MeanPool2d_stride1和Downsample2d已经定义，并进行初始化
        self.meanpool2d_stride1 = MeanPool2d_stride1(kernel)
        self.downsample2d = Downsample2d(stride)

    def forward(self, A):
        """
        使用步长为1的均值池化，然后下采样到期望的步长
        """
        # 首先进行步长为1的均值池化
        Z = self.meanpool2d_stride1.forward(A)
```

```
        # 根据stride进行下采样
        Z = self.downsample2d.forward(Z)

        return Z

    def backward(self, dLdZ):
        """
        反向传播同样需要考虑下采样和均值池化的梯度传递
        """
        # 首先根据stride对梯度进行上采样
        dLdZ_upsampled = self.downsample2d.backward(dLdZ)

        # 然后将上采样的梯度通过步长为1的均值池化的反向传播
        dLdA = self.meanpool2d_stride1.backward(dLdZ_upsampled)

        return dLdA
```

7 Flatten层

Flatten层是神经网络中的一种结构，它没有学习参数，其主要作用是将多维的输入数据“展平”成一维的形式。这通常在卷积神经网络（CNN）中用于卷积层和全连接层之间，将卷积层产生的多维特征图（feature maps）转换成一维的特征向量，以便后续的全连接层可以处理。

7.1 Flatten层的原理

假设Flatten层的输入是一个多维数组（例如，在CNN中，一个batch的特征图的尺寸可能为[batch_size, channels, height, width]），Flatten层会保持第一维（batch_size）不变，而将其余的维度展开成一个大的一维数组。例如，如果输入的尺寸是[32, 10, 28, 28]（即32个样本，每个样本有10个通道，每个通道的特征图尺寸是28x28），Flatten层之后的输出尺寸将是[32, 7840]（即32个样本，每个样本是7840个特征值的一维向量）。

7.2 Flatten层的反向传播

反向传播通过网络传播误差以更新网络参数的过程。由于Flatten层不涉及权重和偏置等学习参数，它的反向传播相对简单：

- **原理**: Flatten层的反向传播只需要将梯度的形状从一维变回Flatten之前的多维形状。这是一个形状变换操作，不涉及数据内容的变化或计算。
- **操作**: 假设损失函数相对于Flatten层输出的梯度已知（设其形状为[batch_size, features]），那么在反向传播时，这个梯度需要被重新塑形（reshape）回Flatten层输入时的形状（例如[batch_size, channels, height, width]）。这样，梯度就可以继续传播到前面的层（例如卷积层）。

7.3 代码实现

```
import numpy as np

class Flatten():

    def forward(self, A):
        """
        Argument:
            A (np.array): (batch_size, in_channels, in_width)
        Return:
            Z (np.array): (batch_size, in_channels * in width)
        """
```

```
        self.A_shape = A.shape # 保存输入形状以便在反向传播时使用
        Z = A.reshape(A.shape[0], -1)  # A.reshape(batch_size, in_channels * in_width)

        return Z

    def backward(self, dLdZ):
        """
        Argument:
            dLdZ (np.array): (batch size, in channels * in width)
        Return:
            dLdA (np.array): (batch size, in channels, in width)
        """

        dLdA = dLdZ.reshape(self.A_shape)  # 将梯度重新塑形为原始输入形状

        return dLdA
```

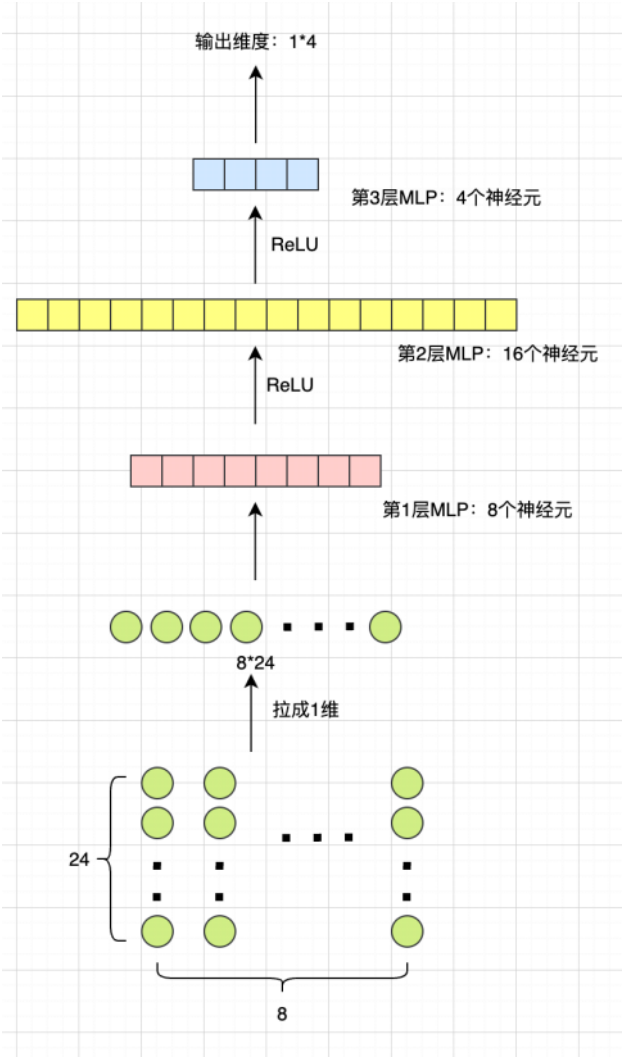
8 用1D卷积复现MLP网络

备注：本人对HW2P1讲义中的结构描述要求实在难以读懂，主要是第一层的核大小为什么要是8，而后续全为1。具体不细究需求了，直接看答案。当然，要注意权重转换问题。

MLP网络结构如下：

```
import numpy as np
from layers import *

class MLP():
    def __init__(self, layer_sizes):
        self.layers = []

        for i in range(len(layer_sizes) - 1):
            in_size, out_size = layer_sizes[i], layer_sizes[i + 1]
            self.layers.append(Linear(in_size, out_size))
            self.layers.append(ReLU())

        self.layers = self.layers[:-1]  # remove final ReLU

    def init_weights(self, weights):
        for i in range(len(weights)):
            self.layers[i * 2].W = weights[i].T

    def forward(self, A):
        Z = A
        for layer in self.layers:
            Z = layer(Z)
        return Z

    def backward(self, dLdZ):
        dLdA = dLdZ
        for layer in self.layers[::-1]:
            dLdA = layer.backward(dLdA)
        return dLdA
```

```
if __name__ == '__main__':
    D = 24  # length of each feature vector
    mlp = MLP([8 * D, 8, 16, 4])
```



使用1D卷积复现：

```
from flatten import Flatten
from Conv1d import Conv1d
from linear import Linear
from activation import ReLU
import numpy as np
import sys

sys.path.append('mytorch')


class CNN_SimpleScanningMLP():
    def __init__(self):
        # 第一层卷积层，模拟MLP中对输入数据的首次处理。
        # in_channels=24：作业讲义中提到总共128个数据点，每个数据有24个特征，即输入数据通道数是24。注意这里没有batchsize。
        # out_channels=8： 表示此卷积层生成8个特征映射，类似于MLP中第一层有8个神经元
        # kernel_size=8：1D卷积每次处理8个数据点。
        # stride=4： 每次滑动4个数据点距离。
        self.conv1 = Conv1d(in_channels=24, out_channels=8, kernel_size=8, stride=4)

        # 第二层卷积层，使用1x1卷积核实现类似全连接层的效果。
```

```
        # out_channels=16，相当于MLP中第二层的16个神经元。
        # kernel_size=1 和 stride=1 保证这一层完全连接到前一层的所有输出。
        self.conv2 = Conv1d(in_channels=8, out_channels=16, kernel_size=1, stride=1)

        # 第三层卷积层，同样使用1x1卷积核。
        # out_channels=4，对应MLP中最后一层的4个输出神经元。
        self.conv3 = Conv1d(in_channels=16, out_channels=4, kernel_size=1, stride=1)

        # 经过3层1D卷积后，128*24的数据，最终输出维度是 31*4，其中31是128个数据点进行窗口为8步长为4的滑动结果。

        # 构建激活层和最后的扁平化层
        self.layers = [
            self.conv1,
            ReLU(),
            self.conv2,
            ReLU(),
            self.conv3,
            Flatten()  # 将多维输出扁平化成单一维度，模拟MLP的行为
        ]

    def init_weights(self, weights):
        # 初始化权重，将MLP的权重转置并赋给相应的卷积层
        # 注意，这里需要将MLP的权重矩阵转置，因为在MLP中权重矩阵的维度是(out_features, in_features)
        w1, w2, w3 = weights
        w1 = np.transpose(w1).reshape((self.conv1.conv1d_stride1.out_channels, self.conv1.conv1d_stride1.kernel_size, self.conv1.conv1d_stride1.in_channels))
        w2 = np.transpose(w2).reshape((self.conv2.conv1d_stride1.out_channels, self.conv2.conv1d_stride1.kernel_size, self.conv2.conv1d_stride1.in_channels))
        w3 = np.transpose(w3).reshape((self.conv3.conv1d_stride1.out_channels, self.conv3.conv1d_stride1.kernel_size, self.conv3.conv1d_stride1.in_channels))
        self.conv1.conv1d_stride1.W = np.transpose(w1, (0,2,1))
        self.conv2.conv1d_stride1.W = np.transpose(w2, (0,2,1))
        self.conv3.conv1d_stride1.W = np.transpose(w3, (0,2,1))

    def forward(self, A):
        """
        执行模型的前向传播。
        参数:
            A (np.array): 输入数组，形状为 (batch size, in channel, in width)
        返回:
            Z (np.array): 输出数组，形状为 (batch size, out channel, out width)
        """
        Z = A
        for layer in self.layers:
            Z = layer.forward(Z)
        return Z

    def backward(self, dLdZ):
        """
        执行模型的后向传播。
        参数:
            dLdZ (np.array): 损失关于输出的梯度
        返回:
            dLdA (np.array): 损失关于输入的梯度
        """
        dLdA = dLdZ
```

```
        for layer in self.layers[::-1]:
            dLdA = layer.backward(dLdA)
        return dLdA
```

关于1D卷积的效果可以参考如下图：

11.1 Scanning MLP : Illustration

Consider a 1-D CNN model (This explanation generalizes to any number of dimensions).
Specifically consider a CNN with the following architecure:

- Layer 1: 2 filters of kernel width 2, with stride 2
- Layer 2: 3 filters of kernel width 3, with stride 3
- Layer 3: 2 filters of kernel width 3, with stride 3
- Finally a single softmax unit which combines all the outputs of the final layer.

This is a regular, if simple, 1-D CNN. The computation performed can be visualized by the figure below.

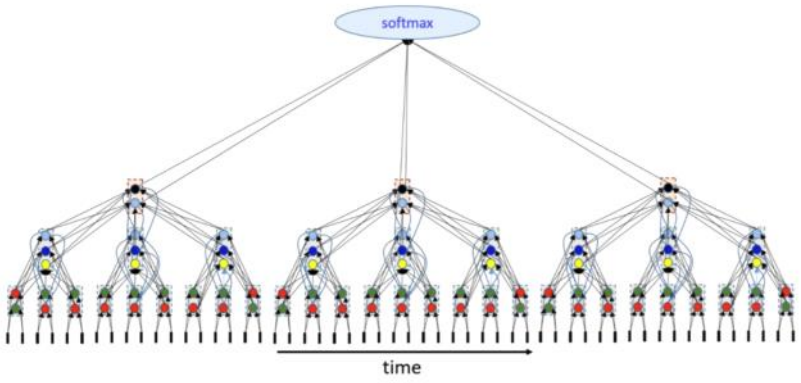
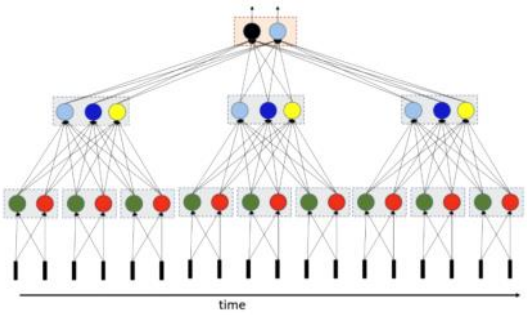


Figure 29: Scanning MLP Figure 1

上述竖排列等价于下图，只是为了让图没那么密集：



同理，还有个CNN_DistributedScanningMLP的需求并没理解，仅放上答案：

```
class CNN_DistributedScanningMLP():
    def __init__(self):
        self.conv1 = Conv1d(in_channels=24, out_channels=2, kernel_size=2, stride=2)
        self.conv2 = Conv1d(in_channels=2, out_channels=8, kernel_size=2, stride=2)
        self.conv3 = Conv1d(in_channels=8, out_channels=4, kernel_size=2, stride=1)
        self.layers = [self.conv1, ReLU(), self.conv2, ReLU(), self.conv3, Flatten()]

    def __call__(self, A):
        # Do not modify this method
        return self.forward(A)

    def init_weights(self, weights):
        w1, w2, w3 = weights
```

```
        w1 = np.reshape(w1[:48,:2].T, (2,2,24)) #2,2,24
        w2 = np.reshape(w2[:4,:8].T, (8,2,2))   #8,2,2
        w3 = np.reshape(w3[:16,:4].T, (4,2,8))  #4,2,8

        w1 = np.transpose(w1, (0,2,1))  #2,24,2
        w2 = np.transpose(w2, (0,2,1))  #8,2,2
        w3 = np.transpose(w3, (0,2,1))  #4,8,2
        self.conv1.conv1d_stride1.W = w1
        self.conv2.conv1d_stride1.W = w2
        self.conv3.conv1d_stride1.W = w3

    def forward(self, A):
        """
        Do not modify this method

        Argument:
            A (np.array): (batch size, in channel, in width)
        Return:
            Z (np.array): (batch size, out channel , out width)
        """

        Z = A
        for layer in self.layers:
            Z = layer.forward(Z)
        return Z

    def backward(self, dLdZ):
        """
        Do not modify this method

        Argument:
            dLdZ (np.array): (batch size, out channel, out width)
        Return:
            dLdA (np.array): (batch size, in channel, in width)
        """
        dLdA = dLdZ
        for layer in self.layers[::-1]:
            dLdA = layer.backward(dLdA)
        return dLdA
```

9 构建CNN模型

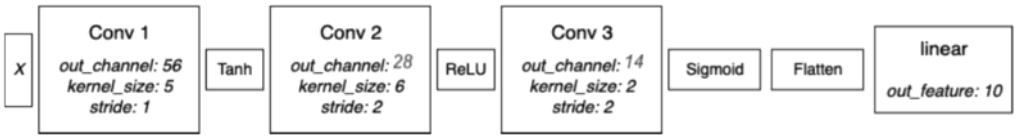


Figure 28: CNN Architecture to implement.

```
# DO NOT import any additional 3rd party external libraries as they will not
# be available to AutoLab and are not needed (or allowed)

from flatten import *
from Conv1d import *
from linear import *
```

```
from activation import *
from loss import *
import numpy as np
import os
import sys



class CNN(object):
    """
    实现一个简单的卷积神经网络模型。
    你需要在下面的 get_cnn_model 函数中指定详细的模型架构，其架构应与第3.3节图3相同。
    """

    def __init__(self, input_width, num_input_channels, num_channels, kernel_sizes,
strides,
                 num_linear_neurons, activations, conv_weight_init_fn, bias_init_fn,
                 linear_weight_init_fn, criterion, lr):
        """
        初始化CNN模型的参数。

        参数:
        input_width           : int     : 第一个卷积层输入的宽度
        num_input_channels    : int     : 输入层的通道数
        num_channels          : [int]   : 每个卷积层的输出通道数列表
        kernel_sizes          : [int]   : 每个卷积层的核宽度列表
        strides               : [int]   : 每个卷积层的步长列表
        num_linear_neurons    : int     : 线性层的神经元数
        activations           : [obj]   : 每个卷积层对应的激活函数对象列表
        conv_weight_init_fn   : fn      : 初始化卷积层权重的函数
        bias_init_fn          : fn      : 初始化所有卷积层和线性层偏置的函数
        linear_weight_init_fn : fn      : 初始化线性层权重的函数
        criterion             : obj     : 使用的损失函数对象（例如SoftMaxCrossEntropy）
        lr                    : float   : 学习率

        注意: activations, num_channels, kernel_sizes, strides 的长度必须一致。
        """

        # Don't change this -->
        self.train_mode = True
        self.nlayers = len(num_channels)

        self.activations = activations
        self.criterion = criterion

        self.lr = lr
        # <---------------------

        # Don't change the name of the following class attributes,
        # the autograder will check against these attributes. But you will need to
change
        # the values in order to initialize them correctly

        # Your code goes here -->
        # self.convolutional_layers (list Conv1d) = []
        # self.flatten              (Flatten)     = Flatten()
        # self.linear_layer         (Linear)      = Linear(???)
        # <---------------------
```

```
        # 初始化卷积层
        self.convolutional_layers = [Conv1d(num_input_channels, num_channels[0],
kernel_sizes[0], strides[0]),
                                     Conv1d(num_channels[0], num_channels[1],
kernel_sizes[1], strides[1]),
                                     Conv1d(num_channels[1], num_channels[2],
kernel_sizes[2], strides[2])]
        self.flatten = Flatten()

        # 计算每个卷积层的输出尺寸
        conv1_out = (input_width-kernel_sizes[0])/strides[0]+1
        conv2_out = (conv1_out-kernel_sizes[1])/strides[1]+1
        conv3_out = (conv2_out-kernel_sizes[2])/strides[2]+1
        self.linear_layer = Linear(int(conv3_out*num_channels[2]), num_linear_neurons)

        # 将所有层组合到一个列表中
        self.layers = [self.convolutional_layers[0],
                       activations[0],
                       self.convolutional_layers[1],
                       activations[1],
                       self.convolutional_layers[2],
                       activations[2],
                       self.flatten,
                       self.linear_layer
                      ]

    def forward(self, A):
        """
        Argument:
            A (np.array): (batch_size, num_input_channels, input_width)
        Return:
            Z (np.array): (batch_size, num_linear_neurons)
        """

        # Your code goes here -->
        # Iterate through each layer
        # <---------------------

        # Save output (necessary for error and loss)
        Z = A
        for layer in self.layers:
            Z = layer.forward(Z)

        self.Z = Z

        return self.Z

    def backward(self, labels):
        """
        Argument:
            labels (np.array): (batch_size, num_linear_neurons)
        Return:
            grad (np.array): (batch size, num_input_channels, input_width)
        """

        m, _ = labels.shape
        self.loss = self.criterion.forward(self.Z, labels).sum()
        grad = self.criterion.backward()

        # Your code goes here -->
```

```
        # Iterate through each layer in reverse order
        # <--------------------
        for layer in reversed(self.layers):
            grad = layer.backward(grad)
        return grad

    def zero_grads(self):
        # Do not modify this method
        # 清零所有梯度
        for i in range(self.nlayers):
            self.convolutional_layers[i].conv1d_stride1.dLdW.fill(0.0)
            self.convolutional_layers[i].conv1d_stride1.dLdb.fill(0.0)

        self.linear_layer.dLdW.fill(0.0)
        self.linear_layer.dLdb.fill(0.0)

    def step(self):
        # Do not modify this method
        # # 更新权重
        for i in range(self.nlayers):
            self.convolutional_layers[i].conv1d_stride1.W =
(self.convolutional_layers[i].conv1d_stride1.W -
                                                              self.lr *
self.convolutional_layers[i].conv1d_stride1.dLdW)
            self.convolutional_layers[i].conv1d_stride1.b =
(self.convolutional_layers[i].conv1d_stride1.b -
                                                              self.lr *
self.convolutional_layers[i].conv1d_stride1.dLdb)

        self.linear_layer.W = (
            self.linear_layer.W -
            self.lr *
            self.linear_layer.dLdW)
        self.linear_layer.b = (
            self.linear_layer.b -
            self.lr *
            self.linear_layer.dLdb)

    def train(self):
        # Do not modify this method
        self.train_mode = True

    def eval(self):
        # Do not modify this method
        self.train_mode = False
```

10 额外作业（HW2P1_Bonus）

主要就是实现如下3个模块：

1. dropout 2D（原理可参考HW1P1中的 dropout 1D实现）
2. BN 2D（原理可参考HW1P1中的 BN 1D实现）
3. ResNet block

测试方法：python autograder/hw2_bonus_autograder/runner.py

10.1 实现dropout 2d

```
import numpy as np

np.random.seed(11785)

class Dropout2d(object):
    def __init__(self, p=0.5):
        # 初始化dropout概率p，默认为0.5
        self.p = p
        self.mask = None  # 初始化掩码，用于在训练过程中随机关闭部分神经元

    def __call__(self, *args, **kwargs):
        # 使得类实例可以像函数一样被调用，内部调用forward方法
        return self.forward(*args, **kwargs)

    def forward(self, x, eval=False):
        """
        前向传播方法：
        参数：
          x (np.array): 输入数据，其形状为(batch_size, in_channel, input_width,
input_height)
          eval (boolean): 表示模型是否处于评估模式
        返回：
          和输入形状相同的np.array
        """
        if eval:
            # 如果是评估模式，直接返回输入数据x，不进行任何dropout操作
            return x
        else:
            # 如果是训练模式，进行dropout操作
            self.mask = np.random.binomial(1, 1-self.p, size=(x.shape[0], x.shape[1],
1, 1))
            # 生成一个随机掩码，掩码中的元素非0即1，1的概率为1-p
            return x * self.mask * 1/(1-self.p)
            # 应用掩码，并通过1/(1-p)缩放以保持激活值的总量不变

    def backward(self, delta):
        """
        反向传播方法：
        参数：
          delta (np.array): 上游传来的梯度，其形状为(batch_size, in_channel, input_width,
input_height)
        返回：
          和输入梯度形状相同的np.array
        """
        # 训练模式下，进行梯度传播
        return delta * self.mask * 1/(1-self.p)
        # 通过已保存的掩码对梯度进行调整，再通过1/(1-p)缩放
```

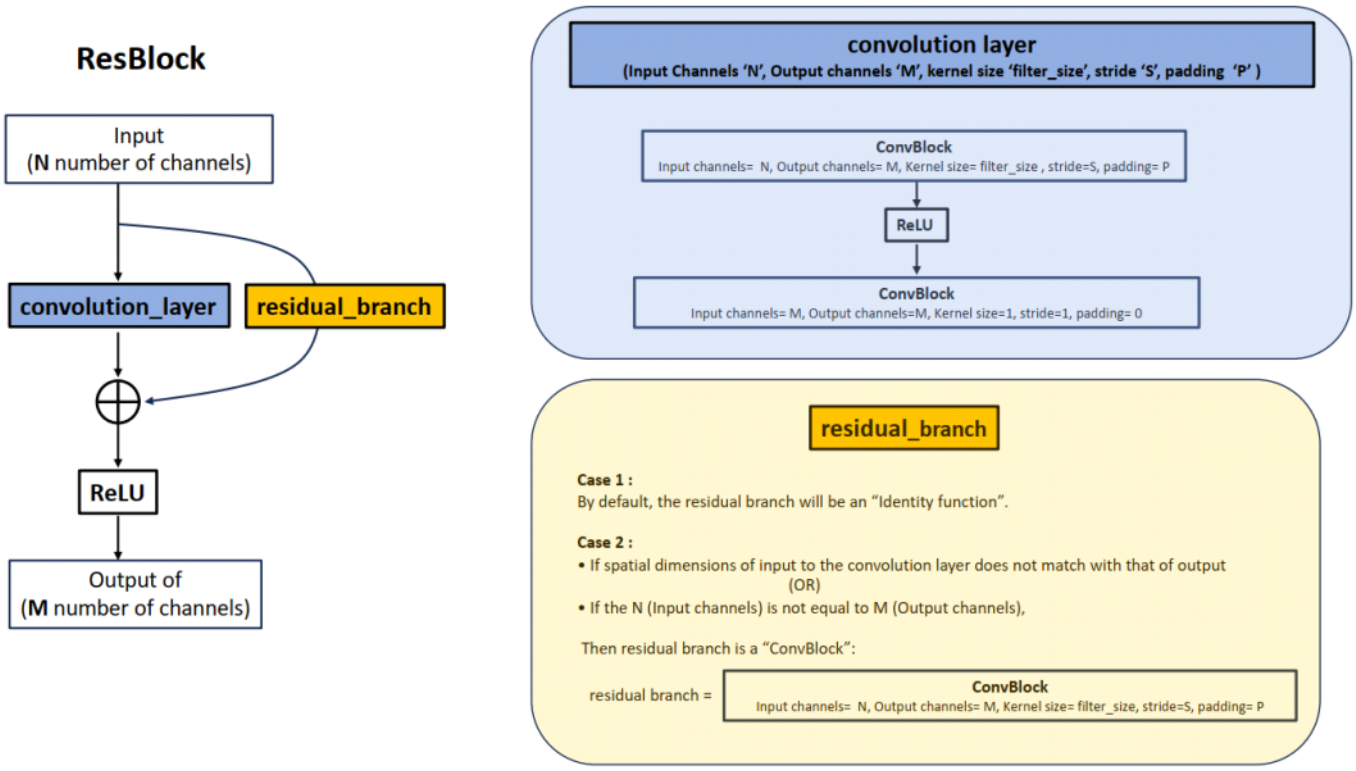
10.2 实现BN 2d

```python
import numpy as np

class BatchNorm2d:
    def __init__(self, num_features, alpha=0.9):
        # 初始化批量归一化层
        # num_features: 特征数，即通道数
        # alpha: 用于运行时均值和方差的移动平均系数
        self.alpha = alpha  # 移动平均的衰减因子
        self.eps = 1e-8  # 用于防止除以零的小量

        # 归一化后的值、缩放后的值、偏移后的值
        self.Z = None  # 原始输入
        self.NZ = None  # 归一化后的值
        self.BZ = None  # 缩放和偏移后的值

        # 权重和偏置
        self.BW = np.ones((1, num_features, 1, 1))  # 缩放权重
        self.Bb = np.zeros((1, num_features, 1, 1))  # 偏置

        # 梯度
        self.dLdBW = np.zeros((1, num_features, 1, 1))  # 权重梯度
        self.dLdBb = np.zeros((1, num_features, 1, 1))  # 偏置梯度

        # 均值和方差
        self.M = np.zeros((1, num_features, 1, 1))  # 当前批次的均值
        self.V = np.ones((1, num_features, 1, 1))  # 当前批次的方差

        # 运行时均值和方差
        self.running_M = np.zeros((1, num_features, 1, 1))  # 运行时均值
        self.running_V = np.ones((1, num_features, 1, 1))  # 运行时方差

    def __call__(self, *args, **kwargs):
        # 类实例可以像函数一样被调用，内部调用forward方法
        return self.forward(*args, **kwargs)

    def forward(self, Z, eval=False):
        """
        前向传播:
        参数:
          Z (np.array): 输入数据
          eval (boolean): 是否处于评估模式
        """
        if eval:
            # 评估模式下，使用运行时均值和方差进行归一化
            NZ = (Z - self.running_M) / np.sqrt(self.running_V + self.eps)
            BZ = self.BW * NZ + self.Bb
            return BZ

        # 保存原始输入
        self.Z = Z
        # 计算当前批次中所有元素的总数
        self.N = Z.shape[0] * Z.shape[2] * Z.shape[3]

        # 计算当前批次的均值和方差
        self.M = np.mean(Z, axis=(0, 2, 3), keepdims=True)
```

```
        self.V = np.var(Z, axis=(0, 2, 3), keepdims=True)
        # 根据均值和方差进行归一化
        self.NZ = (Z - self.M) / np.sqrt(self.V + self.eps)
        # 应用权重和偏置进行缩放和偏移
        self.BZ = self.BW * self.NZ + self.Bb

        # 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
        return self.BZ

    def backward(self, dLdBZ):
        """
        反向传播:
        参数:
          dLdBZ (np.array): 上游传递的梯度
        """
        # 计算关于偏置的梯度
        self.dLdBb = np.sum(dLdBZ, axis=(0, 2, 3), keepdims=True)
        # 计算关于权重的梯度
        self.dLdBW = np.sum(dLdBZ * self.NZ, axis=(0, 2, 3), keepdims=True)

        # 根据权重计算输入归一化的梯度
        dLdNZ = dLdBZ * self.BW
        # 计算关于方差的梯度
        dLdV = -0.5 * np.sum(dLdNZ * (self.Z - self.M) * ((self.V + self.eps) **
(-1.5)), axis=(0, 2, 3), keepdims=True)

        # 计算均值对归一化的影响
        dNZdM = -(self.V + self.eps) ** (-0.5) - 0.5 * (self.Z - self.M) * (self.V +
self.eps) ** (-1.5) * (-2 / self.N * np.sum(self.Z - self.M, axis=(0, 2, 3),
keepdims=True))

        # 计算关于均值的梯度
        dLdM = np.sum(dLdNZ * dNZdM, axis=(0, 2, 3), keepdims=True)
        # 综合计算输入数据的梯度
        dLdZ = dLdNZ * (self.V + self.eps) ** (-0.5) + dLdV * (2 / self.N * (self.Z -
self.M)) + 1 / self.N * dLdM

        return dLdZ
```

10.3 实现ResNet block

原理图如下:



类的结构：

```python
class ConvBlock(object):

    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        self.layers = []  # TODO

    def forward(self, A):
        # TODO
        return NotImplemented

    def backward(self, grad):
        # TODO
        return NotImplemented


class ResBlock(object):

    def __init__(self, in_channels, out_channels, filter_size, stride=3, padding=1):
        self.convolution_layers = []  # TODO
        self.final_activation = None  # TODO
        if stride != 1 or in_channels != out_channels or filter_size != 1 or padding != 0:
            self.residual_connection = None  # TODO
        else:
            self.residual_connection = None  # TODO

    def forward(self, A):
        # TODO
        return NotImplemented

    def backward(self, grad):
        # TODO
        return NotImplementedError
```

最终代码：

```
import sys
import numpy as np
import os

sys.path.append("mytorch")
from Conv2d import Conv2d_padding


from activation import *
from batchnorm2d import *



class ConvBlock(object):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        # 初始化一个卷积块，包括卷积层和批归一化层
        self.layers = [
            Conv2d_padding(in_channels, out_channels, kernel_size, stride, padding, bias_init_fn=None),
            BatchNorm2d(out_channels),
        ]

    def forward(self, A):
        # 前向传播，依次通过卷积层和批归一化层
        for layer in self.layers:
            A = layer.forward(A)
        return A

    def backward(self, grad):
        # 反向传播，依次逆序通过批归一化层和卷积层
        for layer in reversed(self.layers):
            grad = layer.backward(grad)
        return grad


class IdentityBlock(object):
    def forward(self, A):
        # 恒等块的前向传播，直接返回输入
        return A

    def backward(self, grad):
        # 恒等块的反向传播，直接返回梯度
        return grad


class ResBlock(object):
    def __init__(self, in_channels, out_channels, filter_size, stride=3, padding=1):
        # 初始化残差块，包括卷积块、激活函数和残差连接
        self.convolution_layers = [
            ConvBlock(in_channels, out_channels, filter_size, stride, padding),
            ReLU(),
            ConvBlock(out_channels, out_channels, 1, 1, 0),
        ]
        self.final_activation = ReLU()


        # 决定残差连接是使用卷积块还是恒等块
        # 当步幅（stride）不为1时，卷积层的输出大小会发生变化。因此，如果步幅不为1，残差连接就不能直接使用输入数据，需要
```

```
通过卷积调整输入的尺寸。
        # 当输入通道数（in_channels）和输出通道数（out_channels）不相同时，输入数据与输出数据的形状不匹配，需要通过卷积调整通道数。
        # 当滤波器大小（filter_size）不为1时，意味着卷积核尺寸变大。一般情况下，这个条件会影响到卷积操作的范围.
        # 当填充（padding）不为0时，输入数据的尺寸会在卷积操作中保持不变或增大。因此，这个条件可能影响输出尺寸.

        # 在很多实现中，常见的残差块在判断是否需要卷积块来处理残差连接时，重点是步幅和通道数是否匹配。
        # 这两个条件（stride != 1 或 in_channels != out_channels）已经可以覆盖绝大多数情况下残差连接需要调整的场景。
        # 在S24版测试用例中，只使用stride != 1 or in_channels != out_channels也能通过。
        if stride != 1 or in_channels != out_channels or filter_size != 1 or padding != 0:
            # 当输入和输出维度不匹配时，使用卷积块作为残差连接
            self.residual_connection = ConvBlock(in_channels, out_channels, filter_size, stride, padding)
        else:
            # 当输入和输出维度匹配时，使用恒等块作为残差连接
            self.residual_connection = IdentityBlock()


    def forward(self, A):
        # 前向传播，通过卷积块和激活函数，并加上残差连接的输出
        Z = A
        for layer in self.convolution_layers:
            Z = layer.forward(Z)

        # 通过残差连接获取残差输出
        residual_out = self.residual_connection.forward(A)
        # 将卷积块的输出与残差连接的输出相加，并通过最终的ReLU激活函数
        return self.final_activation.forward(Z + residual_out)

    def backward(self, grad):
        # 残差块的反向传播，首先通过最终的ReLU激活函数计算梯度
        grad = self.final_activation.backward(grad)

        # 计算残差连接的梯度
        residual_grad = self.residual_connection.backward(grad)

        # 依次通过卷积块计算梯度
        for layer in reversed(self.convolution_layers):
            grad = layer.backward(grad)

        # 将卷积块的梯度与残差连接的梯度相加，得到最终的梯度
        return grad + residual_grad
```