# 作业二：

一.BatchNormalization

基础知识：

1.deep networks容易训练两种办法：1.使用的复杂的优化程序SGD+momentum、RMSProp或Adam2.改变网络结构，使得更容易训练（Batch Normalization）

2.机器学习方法往往在由均值和单位方差为零的不相关特征组成的输入数据中表现更好

这种层使用一个小批量的数据来估计每个特征的均值和标准差。这些估计的平均值和标准偏差然后用于集中和标准化小批量的特征。在训练期间保存这些均值和标准差的运行平均值，在测试时使用这些运行平均值来集中和规范化特征。这种归一化策略可能会降低网络的表示能力，因为对于某些层来说，具有非零均值或单位方差的特征有时可能是最优的。为此，批归一化层包括每个特征维度的可学习移位和缩放参数。、

3.batchnorm_forward函数 train：mean std x归一化，outpu=h*x+b（缩放，偏移），test时由于数量太少不适合求mean，我们用

running_mean = momentum * running_mean + (1 - momentum) * sample_mean

running_var = momentum * running_var + (1 - momentum) * sample_var

4

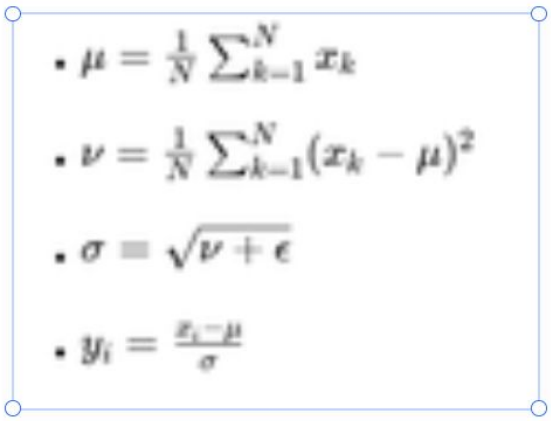
| 特性 | 批量归一化（BN）反向传播 | 普通层（如全连接层）反向传播 |
| --- | --- | --- |
| **梯度来源** | **多路径依赖**。输入$x$的梯度（dx）来源于三条路径：直接来自$y$的梯度、通过**方差**$\sigma^2$的梯度、通过**均值**$\mu$的梯度 [2] [4] 。 | **单一样本依赖**。每个样本的梯度计算相对独立，主要依赖于当前样本的输入、权重和上游梯度。 |
| **参数梯度** | 需要计算对可学习参数 **gamma和beta** 的梯度（dgamma, dbeta），这是BN层特有的 [1] [7] 。 | 只需计算对权重W和偏置b的梯度（dW, db）。 |
| **计算复杂度** | **高**。需要计算并合并来自均值路径和方差路径的中间梯度，公式较为复杂 [2] [3] 。 | **相对较低**。主要是矩阵乘法和加法。 |
| **缓存需求** | **需要缓存大量中间变量**，如x_hat, sample_mean, sample_var, gamma等，用于反向传播中的梯度计算 [3] 。 | 通常只需缓存前向传播的输入$x$和权重W等少量变量。 |

5.Alternative Backward Pass

常规归一化手段：前向与后向

5.Alternative Backward Pass

常规归一化手段：前向与后向



- $\mu = \frac{1}{N}\sum_{k=1}^{N} x_k$
- $v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2$
- $\sigma = \sqrt{v + \epsilon}$
- $y_i = \frac{x_i - \mu}{\sigma}$

```python
# 传统实现 - 逐步计算
def batchnorm_backward_naive(dout, cache):
    # 1. 反向传播通过归一化操作
    dx_hat = dout * gamma

    # 2. 反向传播通过方差计算
    dvar = np.sum(dx_hat * (x - mean) * (-0.5) * (var + eps)**(-1.5), axis=0)

    # 3. 反向传播通过均值计算
    dmean = np.sum(dx_hat * (-1) / std, axis=0) + dvar * np.sum(-2*(x-mean))/N

    # 4. 最终合并所有路径
    dx = dx_hat/std + dvar*2*(x-mean)/N + dmean/N

    return dx, dgamma, dbeta
```

xhat是直接利用链式法则求解，dL/dy*dy/dxhat=上述式子

均值 μ = (1/N) * Σx_i

方差 v = (1/N) * Σ(x_i - μ)²

标准差 σ = √(v + ε)

归一化 y_i = (x_i - μ) / σ

```
dv = ∂L/∂y_i × ∂y_i/∂σ × ∂σ/∂v
```

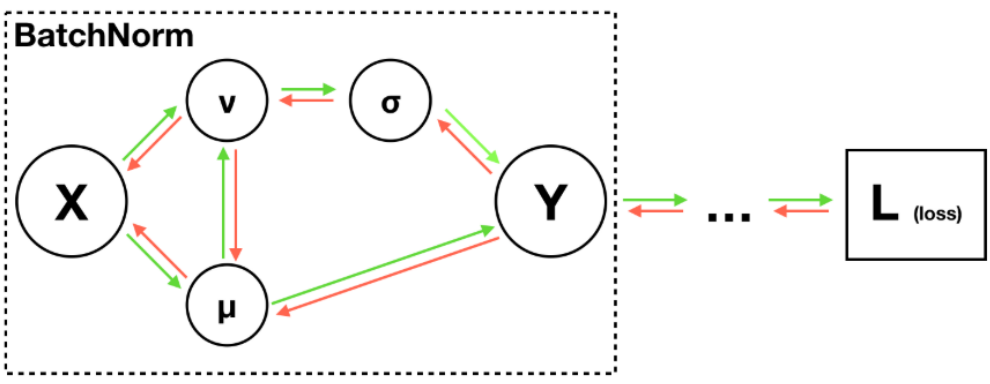
**分步计算**:

1. **∂y_i/∂σ** = (x_i - μ) × (-1)/σ²  (因为 y_i = (x_i - μ)/σ)
2. **∂σ/∂v** = 0.5 × (v + ε)^(-0.5) = 0.5/σ
3. **合并**: ∂y_i/∂σ × ∂σ/∂v = (x_i - μ) × (-1)/σ² × 0.5/σ = (x_i - μ) × (-0.5)/σ³

最终得到:

```
dv = Σ[dx_hat_i × (x_i - μ) × (-0.5) / σ³]
```

升级：




```python
def batchnorm_backward_alt(dout, cache):
    """
    优化的批量归一化反向传播
    """
    # 解包缓存
    x, x_hat, mean, var, gamma, beta, eps = cache
    N, D = x.shape

    # 计算 gamma 和 beta 的梯度（与传统方法相同）
    dgamma = np.sum(dout * x_hat, axis=0)
    dbeta = np.sum(dout, axis=0)

    # 使用优化公式直接计算 dx
    std = np.sqrt(var + eps)
    dy = dout * gamma  # 上游梯度乘以 gamma

    # 应用优化公式
    term1 = N * dy
    term2 = np.sum(dy, axis=0, keepdims=True)
    term3 = x_hat * np.sum(dy * x_hat, axis=0, keepdims=True)

    dx = (term1 - term2 - term3) / (N * std)

    return dx, dgamma, dbeta
```

去掉中间变量均值、方差等，从gamma bais入手求解dx

6.FullyConnectedNet的归一化

BN层（有一定正则化效果，减小Dropout的比例甚至移除Dropout）通常的插入位置是在全连接层之后，非线性激活函数（之前。其核心思路是：先通过全连接层进行线性变换，紧接着使用BN层来稳定这个变换结果的分布，最后再通过激活函数引入非线性。需要注意的是，网络的最后一层（输出层）通常不添加BN层。这是因为输出层需要直接输出最终的预测结果，对其分布有特定的要求（例如分类任务需要输出概率），不适合再做归一化。

7.Batch Normalization and Initialization

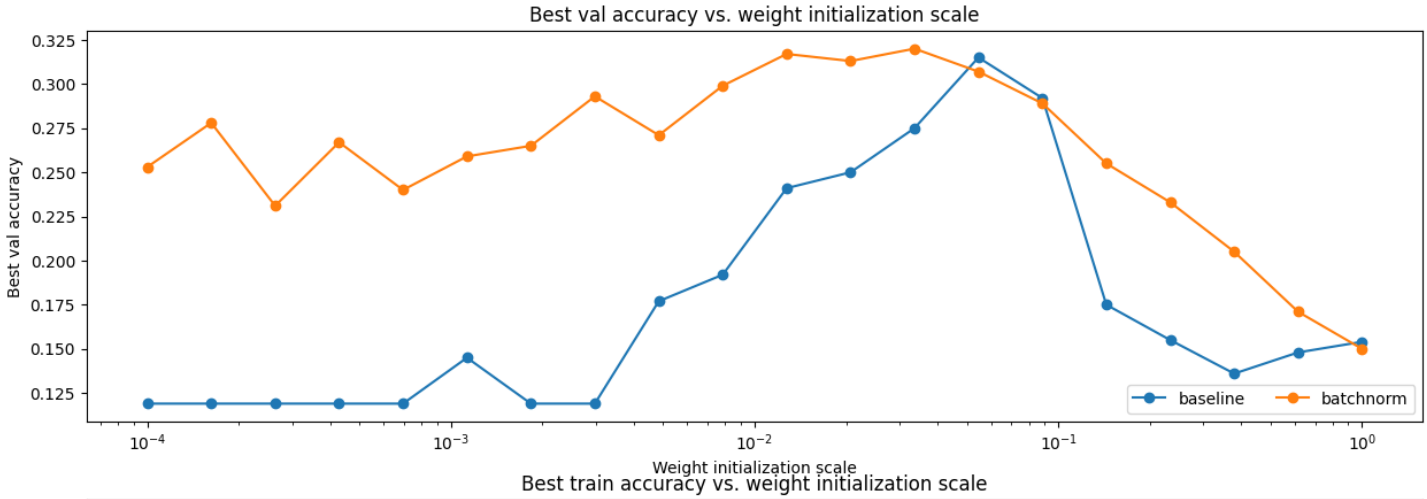
Q1实验1：批量归一化能否降低深度网络对权重初始化尺度的敏感性

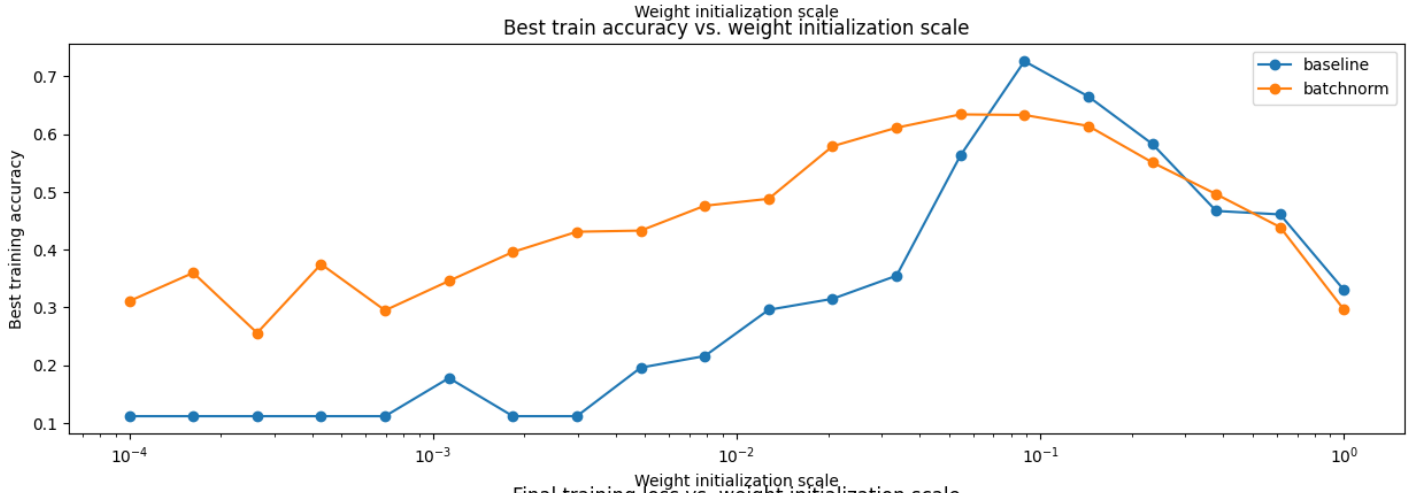
权重：过大，可能在反向传播时导致梯度呈指数级增长；如果初始权重过小，梯度消失，使得网络底层的参数几乎无法更新。反向传播算法进行更新，损失函数关于权重的梯度指示了权重调整的方向和幅度。优化器（如SGD、Adam）则利用这些梯度信息来更新权重。

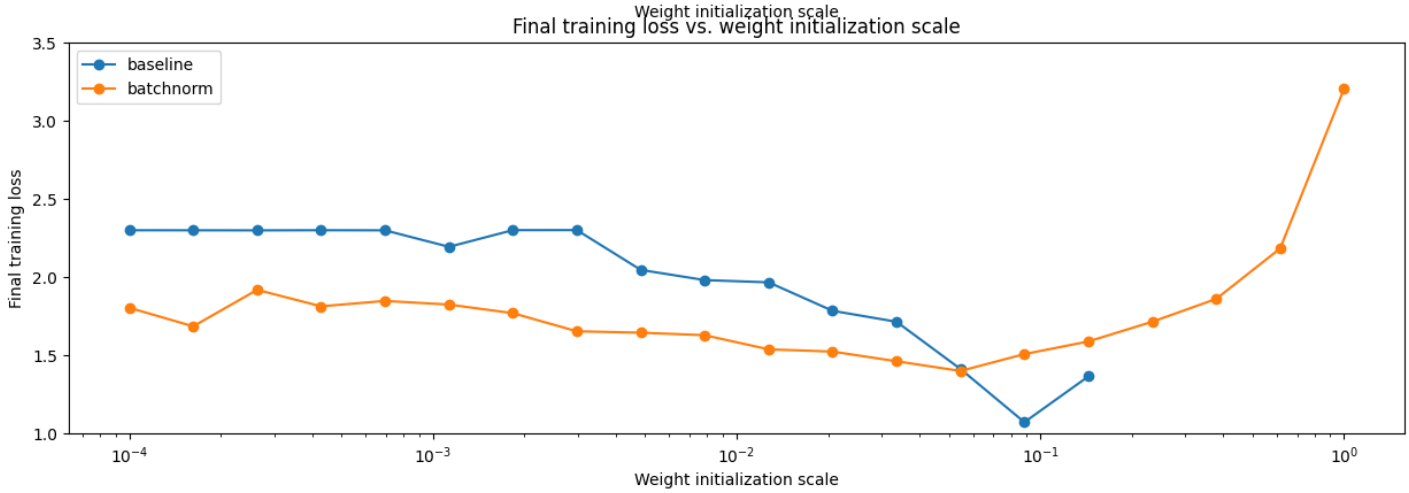
正则化技术（如L1、L2正则化）通过对权重的大小进行惩罚，将其向零压缩，有助于防止过拟合，提升模型的泛化能力。

迁移学习则提供了另一种高效的权重利用方式：我们可以将在大型数据集（如ImageNet）上预训练好的模型权重（通常包含通用的特征提取能力）作为起点，然后用自己的数据对模型进行微调，这尤其适用于自身数据量有限的场景。

实验结果：



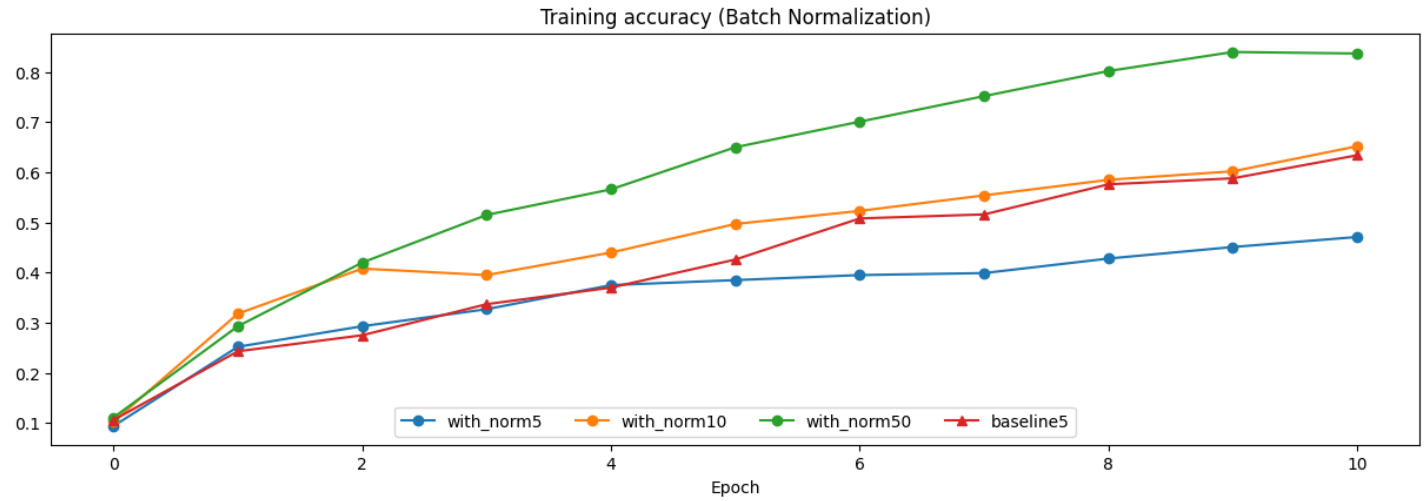



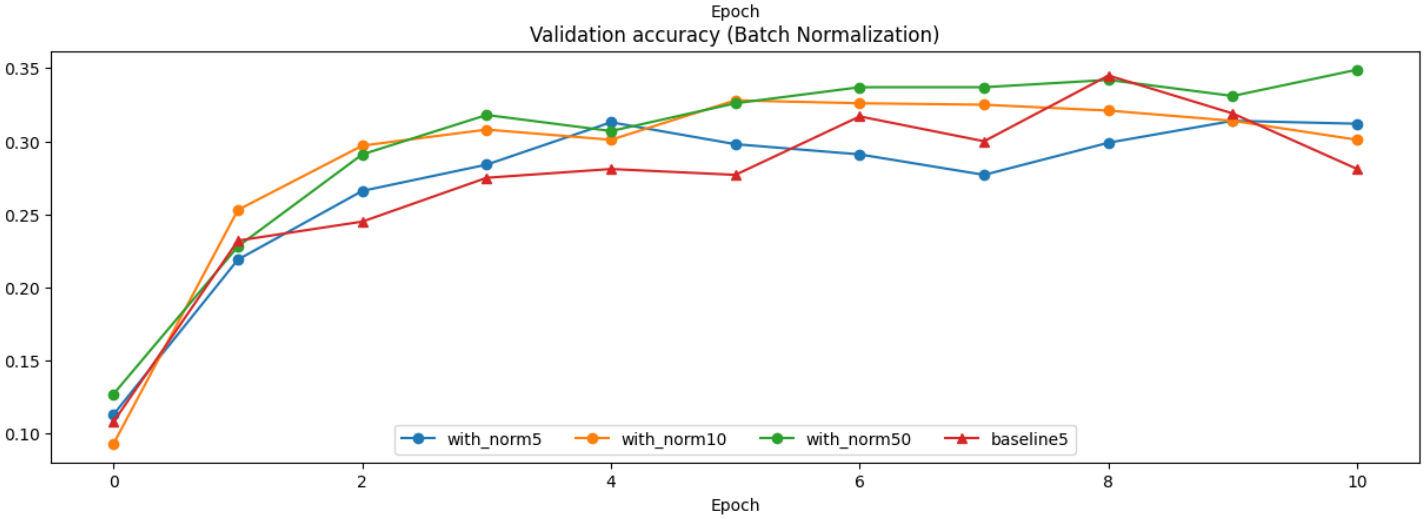




- 训练/验证准确率曲线：对于没有BN的网络，你可能只会看到在某个特定的初始化尺度附近，准确率突然达到一个高峰，而在其他区域则很低。而对于有BN的网络，准确率在很宽的初始化尺度范围内都应该维持在一个较高的水平，曲线会显得更“平缓”和“稳定”。
- 训练损失曲线：没有BN的网络，其训练损失可能在大多数初始化尺度下下降非常缓慢（消失梯度）或者出现NaN（爆炸梯度）。而有BN的网络，其训练损失在各种初始化下都应能平稳下降。
  8.Batch Normalization and Batch Size
- Q2实验2：批量归一化能否对Batch Size有关系

针对两个批次：1.归一化小批次2.SGD中的批次大小

实验结果：




实验结果：

| 观察维度 | 训练准确率图（上图） |
| --- | --- |
| **收敛速度** | **带BN的模型**（蓝、橙、绿线）起点高，快速上升。**基线模型**（红线）起点低，上升缓慢。 |
| **最终性能** | 带BN的模型最终稳定在更高的平台（如~0.95），基线模型性能显著偏低。 |
| **稳定性** | 带BN的曲线**更平滑**，波动小。基线模型（红线）曲线**抖动明显**。 |
| **不同BN设置对比** | `with_norm50`（绿）性能最佳，收敛最快最稳。`with_norm5`和`with_norm10`稍次之。 |

1. 解决内部协变量偏移：深层网络中，前面层参数的微小变化会导致后面层输入分布的剧烈变化，迫使网络不断适应，减慢学习。BN 通过强制每一层的输入具有稳定的分布（固定均值和方差），解决了这个问题，这是收敛加速和训练稳定的主因。
2. 平滑优化地形：BN 使得损失函数对参数更新的敏感度降低。这意味着可以使用更大的学习率而不用担心训练发散，从而更快地穿越平坦区域，找到更优的解。
3. 轻微的正则化效果：由于 BN 在训练时使用小批次的统计量而非整个数据集的统计量，这些统计量带有噪声。这种噪声类似于 Dropout，给模型增加了轻微扰动，防止过拟合，从而提升了验证准确率。

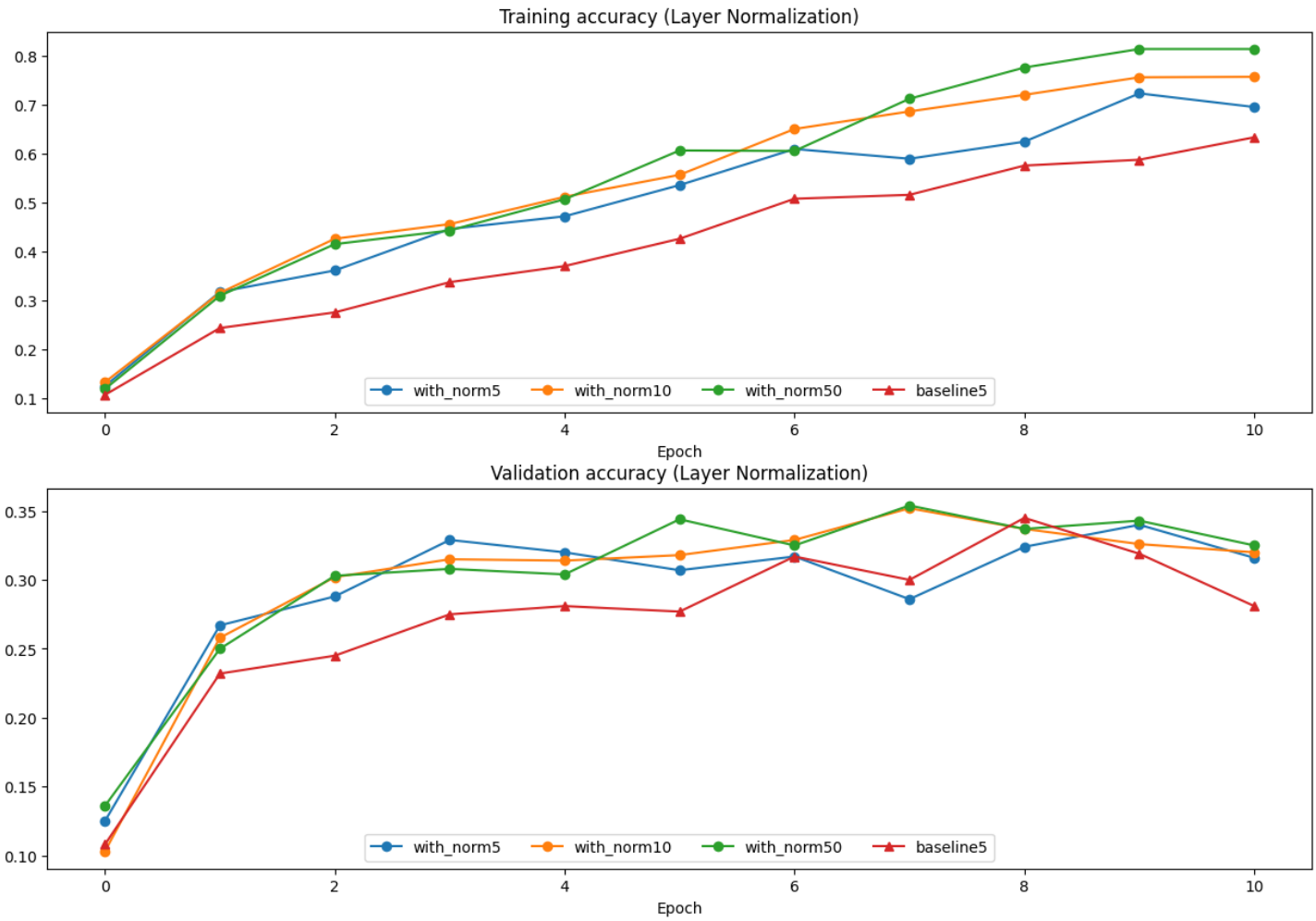
9.Layer Normalization and Batch Size

No normalization: batch size = 5

Normalization: batch size = 5

Normalization: batch size = 10

Normalization: batch size = 50




Q3：层归一化（Layer Normalization, LN）在大多数情况下能有效稳定训练过程，但在某些场景下可能效果不佳。基于您的选项，以下是详细分析：

## 1. 特征维度非常小

- 原因：层归一化的核心是对每个样本的所有特征维度计算均值和方差，然后进行归一化。如果特征维度非常小（例如，只有几个特征），则计算出的均值和方差可能不可靠或噪声较大。这会导致归一化过程不稳定，无法有效调整特征分布，甚至可能引入偏差，从而削弱层归一化的好处。
- 具体机制：当特征维度较小时，统计估计的方差较大，归一化后的特征可能无法代表真实分布。例如，如果特征维度为1，归一化会使特征变为常数（均值为0），丢失所有信息。因此，层归一化在特征维度足够大时工作最佳（如Transformer模型中的隐藏层维度通常较大）。

## 2. 在非常深的网络中使用

- 分析：层归一化实际上在非常深的网络中通常工作良好，甚至是被设计用于解决深层网络中的内部协变量偏移问题。例如，在Transformer模型（如BERT、GPT）中，层归一化是核心组件，能稳定训练极深的网络。相反，它可能改善深层网络的训练。

## 3. 有高正则化项

- 分析：正则化项（如L2正则化）主要用于防止过拟合，而层归一化本身具有一定的正则化效果（通过引入噪声）。高正则化项可能会与层归一化相互作用，但不会直接导致层归一化失败。实际上，层归一化可以允许使用更高的学习率，并可能减少对正则化的依赖。

## 总结

层归一化最可能效果不佳的情况是特征维度非常小。这是因为层归一化依赖于足够大的特征维度来计算稳定的统计量。在其他情况下，层归一化通常能提升模型性能。在实际应用中，建议确保特征维度足够大（例如，至少几十维）以发挥层归一化的优势。

二、Dropout

1.前向传播函数 `dropout_forward`

p, mode = dropout_param["p"], dropout_param["mode"]

主要参数：p保留的概率，模型分为训练与测试

训练集，利用布尔掩码，导致不丢失神经元其输出放大1/p，保证期望与test的一样。

```python
def dropout_forward(x, dropout_param):
    """Dropout前向传播（倒置Dropout）"""
    p, mode = dropout_param["p"], dropout_param["mode"]

    if mode == "train":
        # 生成掩码：以概率p保留神经元
        mask = (np.random.rand(*x.shape) < p) / p  # 除以p保持期望值不变
        out = x * mask
    elif mode == "test":
        out = x  # 测试时不使用Dropout
```

**倒置Dropout原理：**

- 训练时：随机屏蔽神经元，但将保留的神经元放大1/p倍
- 测试时：直接使用所有神经元，无需缩放
- 优点：测试时计算更简单，性能更好

2.反向传播函数 `dropout_backward`

```python
def dropout_backward(dout, cache):
    """Dropout反向传播"""
    dropout_param, mask = cache
    mode = dropout_param["mode"]

    if mode == "train":
        dx = dout * mask  # 只传播未被屏蔽的神经元的梯度
    elif mode == "test":
        dx = dout
```

**反向传播逻辑：**

- 训练时：只对未被屏蔽的神经元传播梯度
- 被屏蔽的神经元梯度为0
- 测试时：直接传播所有梯度

3.原理解析

防止过拟合；防止某些神经元只在其他特定神经元存在时才有效（共适应）；

| 阶段 | Dropout 行为 | 输出处理 | 数学期望（单个神经元） |
| --- | --- | --- | --- |
| **训练** | 随机丢弃部分神经元（概率1-p） | 对保留的神经元输出 **乘以 1/p** | `E = p * (x / p) + (1-p) * 0 = x` |
| **测试** | 使用所有神经元，不进行丢弃 | **不做任何缩放**，直接输出 | `E = 1 * x = x` |

4.位置放置：

p是保留概率，一般隐藏层设置为0.5，输入层设置会很高0.8防止一开始就丢失很多信息，输出层不放。Dropout通常放置在全连接层之后、激活函数（如ReLU）之后。

5.Q1为什么使用p的原因

| 阶段 | 错误实现的输出期望值 | 导致的结果 |
| --- | --- | --- |
| **训练阶段** | `E_train = p * a` | 网络每一层的输入信号强度会随着Dropout的应用而**减弱**（变为原来的 p倍）。 |
| **测试阶段** | `E_test = a` | 使用完整的网络，信号强度正常。 |

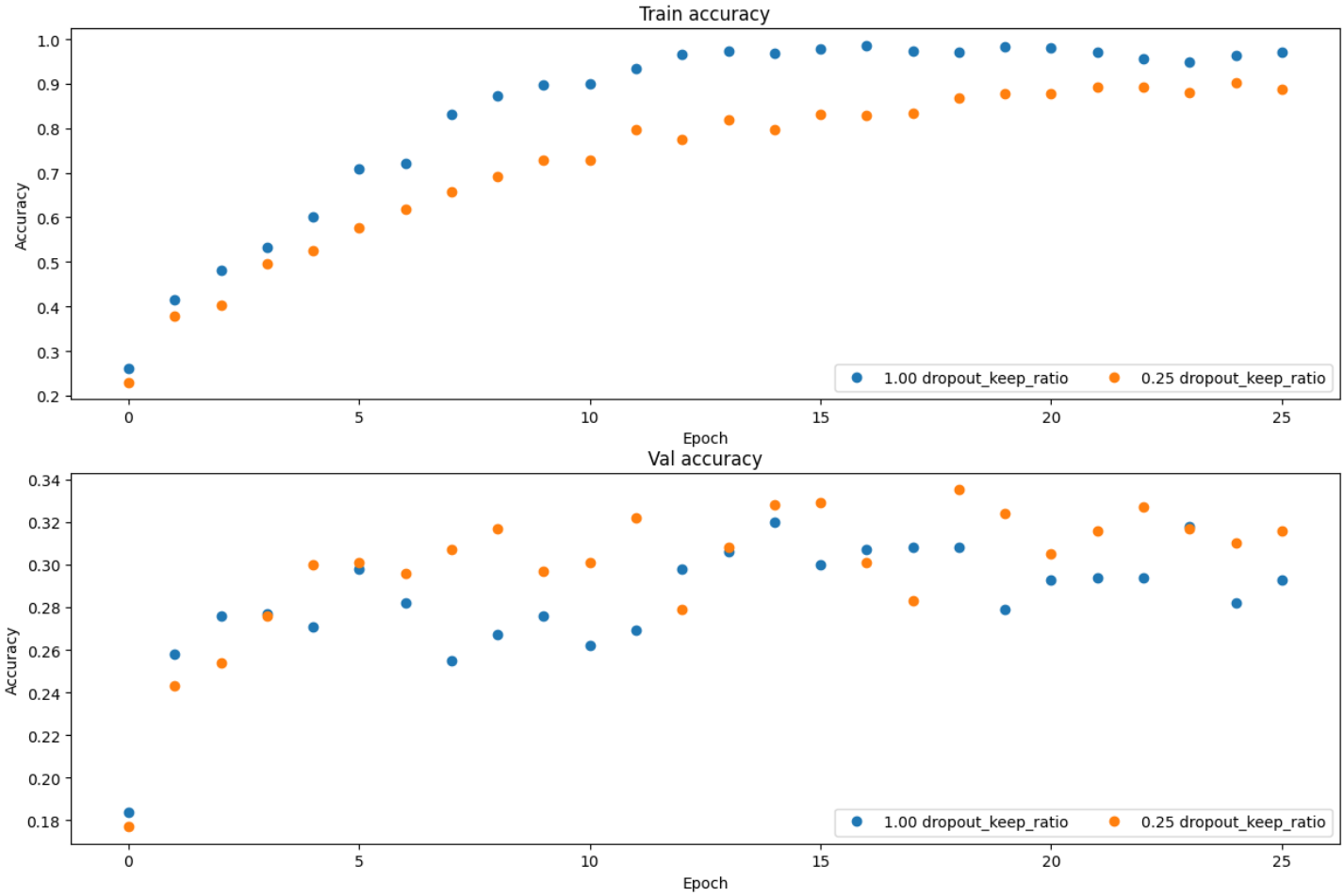
模型训练集适应弱信号，测试却要用强信号

训练数据数值范围发生改变，导致梯度改变，最后使得难以收敛

可能导致过拟合功能失效，数据其往往不稳定

Q2：比较有dropout和没有dropout的验证和训练准确性——你的结果表明dropout是一个正则化器吗？

模拟结果：




### 训练准确率对比（上图）

- 蓝色曲线（无Dropout）：准确率从约0.25开始，急速上升，在10个训练周期后就接近了100%（1.0），并一直保持。
- 橙色曲线（有Dropout）：准确率虽然也在稳步上升，但始终显著低于蓝色曲线，最终停留在90%（0.9）左右。这是因为Dropout在训练时随机地“破坏”了网络结构。

结论一：在训练集上，不使用Dropout的模型表现远优于使用Dropout的模型。这显示了Dropout的“副作用”——它会故意降低模型在训练数据上的拟合能力。

### 验证准确率对比（下图）

这才是衡量模型好坏的关键指标，因为它反映了模型的真实泛化能力。

- 蓝色曲线（无Dropout）：验证准确率虽然也有所上升，但远低于其训练准确率。这表明模型出现了严重的过拟合。
- 橙色曲线（有Dropout）：验证准确率整体上优于蓝色曲线，达到了约0.32的水平。更重要的是，其验证准确率与训练准确率之间的差距要小得多。

结论二：在验证集上，使用Dropout的模型泛化能力反而更好。它虽然“放弃”了部分训练精度，但换来了更强的处理新数据的能力。

图表是“过拟合”与“正则化”概念的经典示例。蓝色曲线展示了典型的过拟合：模型在训练集上表现完美，在验证集上表现平平。而Dropout的加入（橙色曲线）成功地抑制了过拟合，使得模型在验证集上取得了更好的效果。

三、ConvolutionalNetworks

1.conv_forward_naive

```
# 获取输入参数
N, C, H, W = x.shape  # 批次大小，通道数，高度，宽度
F, _, HH, WW = w.shape  # 滤波器数量，通道数，滤波器高，滤波器宽
stride = conv_param['stride']  # 滑动步长
pad = conv_param['pad']  # 填充大小

# 计算输出特征图尺寸
H_out = 1 + (H + 2 * pad - HH) // stride
W_out = 1 + (W + 2 * pad - WW) // stride
```

卷积实际计算：

```
for n in range(N):  # 遍历每个样本
    for f in range(F):  # 遍历每个滤波器
        for i in range(H_out):  # 遍历输出高度
            for j in range(W_out):  # 遍历输出宽度
                # 计算当前感受野的位置
                h_start = i * stride
                h_end = h_start + HH
                w_start = j * stride
                w_end = w_start + WW

                # 提取输入的感受野区域
                x_slice = x_pad[n, :, h_start:h_end, w_start:w_end]

                # 计算卷积结果：元素乘积累加后加上偏置
                out[n, f, i, j] = np.sum(x_slice * w[f]) + b[f]
```

2.conv_backward_naive

首先对偏置梯度计算：

```
for f in range(F):
    db[f] = np.sum(dout[:, f, :, :])
```

$$\frac{\partial L}{\partial b[f]} = \sum_{n=1}^{N} \sum_{i=1}^{H'} \sum_{j=1}^{W'} \frac{\partial L}{\partial y[n,f,i,j]}$$

其次对权重和输入的梯度：

```
for n in range(N):  # 遍历每个样本
    for f in range(F):  # 遍历每个滤波器
        for i in range(H_out):  # 遍历输出高度
            for j in range(W_out):  # 遍历输出宽度
                # 计算当前感受野的位置
                h_start = i * stride
                h_end = h_start + HH
                w_start = j * stride
                w_end = w_start + WW

                # 提取输入的感受野区域
                x_slice = x_pad[n, :, h_start:h_end, w_start:w_end]

                # 计算权重的梯度：输入区域与上游梯度的乘积
                dw[f] += x_slice * dout[n, f, i, j]

                # 计算输入的梯度：权重与上游梯度的乘积
                dx_pad[n, :, h_start:h_end, w_start:w_end] += w[f] * dout[n, f, i, j]
```

公式如下：

1. **权重梯度**：$\frac{\partial L}{\partial w[f]} = \sum_{n,i,j} x_{pad} \times \frac{\partial L}{\partial y[n,f,i,j]}$

   • 每个滤波器的梯度是其感受野输入与对应输出梯度的乘积累加 6

2. **输入梯度**：$\frac{\partial L}{\partial x_{pad}} = \sum_{f,i,j} w[f] \times \frac{\partial L}{\partial y[n,f,i,j]}$

   • 输入梯度是滤波器权重与上游梯度的卷积操作 4

去掉填充：

if pad > 0:

  dx = dx_pad[:, :, pad:-pad, pad:-pad]

else:

  dx = dx_pad

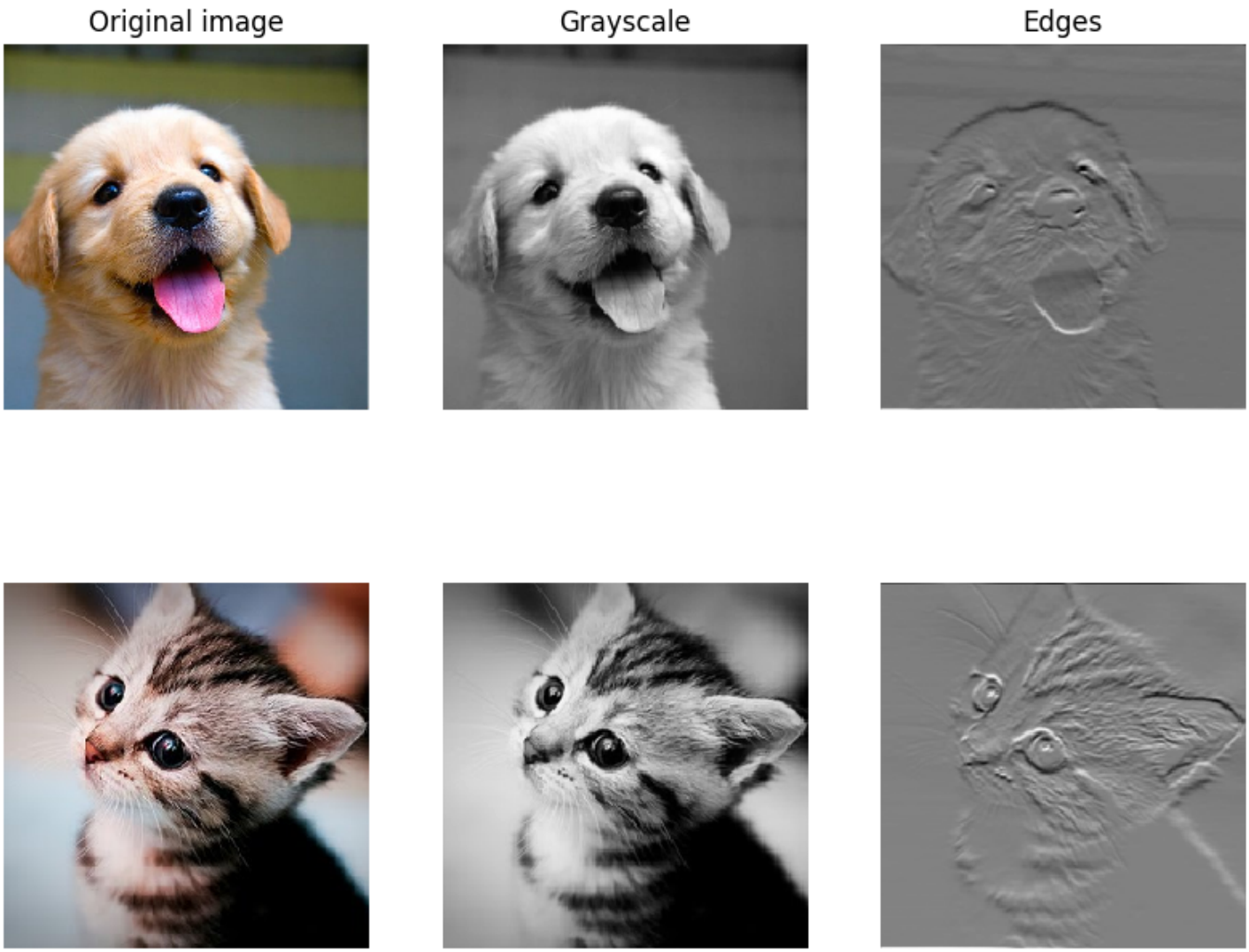
3.检查正确性（设置灰度转换与边缘检测）

第一个卷积核 `w[0]`被设计用于将彩色图像转换为灰度图：

```python
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]  # 红色通道权重 0.3
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]  # 绿色通道权重 0.6
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]  # 蓝色通道权重 0.1
```

第二个卷积核 `w[1]`被设计用于检测图像中的**水平边缘**：

```python
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]  # 权重仅设置在蓝色通道
```




4.Max-Pooling: Naive Forward Pass

最大池化，下采样技术，降低尺寸，保留最显著特征。

```
# 计算输出特征图的尺寸
H_out = 1 + (H - pool_height) // stride  # 输出高度
W_out = 1 + (W - pool_width) // stride   # 输出宽度
```

根据下边公式求解输出尺寸：

$$H' = 1 + \frac{H - HH}{\text{stride}}$$

核心计算：

```
# 初始化输出数组
out = np.zeros((N, C, H_out, W_out))

# 四重循环实现最大池化
for n in range(N):                  # 遍历每个样本
    for c in range(C):              # 遍历每个通道
        for i in range(H_out):      # 遍历输出高度
            for j in range(W_out):  # 遍历输出宽度
                # 计算当前池化窗口位置
                h_start = i * stride
                h_end = h_start + pool_height
                w_start = j * stride
                w_end = w_start + pool_width

                # 提取当前池化窗口区域
                window = x[n, c, h_start:h_end, w_start:w_end]

                # 计算窗口内最大值作为输出
                out[n, c, i, j] = np.max(window)
```

5.Max-Pooling: Naive Backward Pass

```
    for n in range(N):                # 遍历每个样本
        for c in range(C):            # 遍历每个通道
            for i in range(H_out):    # 遍历输出高度
                for j in range(W_out):  # 遍历输出宽度
                    # 定位前向传播时的池化窗口
                    h_start = i * stride
                    h_end = h_start + pool_height
                    w_start = j * stride
                    w_end = w_start + pool_width

                    # 提取前向传播的窗口区域
                    window = x[n, c, h_start:h_end, w_start:w_end]

                    # 找到前向传播时的最大值
                    max_val = np.max(window)

                    # 创建最大值位置的布尔掩码
                    mask = (window == max_val)

                    # 将上游梯度分配给最大值位置
                    dx[n, c, h_start:h_end, w_start:w_end] += mask * dout[n, c, i, j]
```

$$\frac{\partial L}{\partial x_{ij}} = \begin{cases} \frac{\partial L}{\partial y_{kl}}, & \text{if } x_{ij} \text{ 是前向传播的最大值} \\ 0, & \text{otherwise} \end{cases}$$

```
mask = (window == max_val)  # 创建布尔掩码
dx[...] += mask * dout[...]  # 梯度选择性分配
```

6.快速卷积与池化

二者相比结果，加速七百倍左右

```
Testing conv_forward_fast:
Naive: 5.545781s
Fast: 0.026726s
Speedup: 207.505018x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 9.027658s
Fast: 0.012300s
Speedup: 733.955088x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  3.1393858025571252e-15
```

快速池化的技巧：

| 优化条件 | 具体含义 | 对加速效果的影响 |
| --- | --- | --- |
| **非重叠池化** | 池化窗口的步长（stride）等于其尺寸（如 2x2 窗口，步长为 2）。 | **核心条件。**确保计算区域无重叠，实现规整的内存访问和并行计算。 |
| **完整覆盖输入** | 输入特征的宽度和高度必须能被池化窗口的尺寸整除。 | **关键条件。**避免边界处理带来的条件判断和零填充，保持计算流程的整齐划一。 |
| **内存布局友好** | 输入数据在内存中连续存储，并且访问模式符合缓存友好原则。 | **重要条件。**大幅减少缓存未命中（Cache Miss）次数，提升数据读取效率。 |

7.Three-Layer Convolutional Network

主要结构：conv-relu-pool-affine-relu-affline

全连接层：两个全连接层进行高级特征学习和分类输出

输出层：通过softmax函数产生分类概率分布

步骤1：init：初始化卷积层权重w1（标准正态生成随机数，利用weight-scale控制初始化范围（全连接层也做一样处理）

```
self.params['W1'] = weight_scale * np.random.randn(num_filters, C, filter_size, filter_size)
```

步骤二：loss中前向传播

首先参数设置（滤波器、池化层等），卷积，非线性，池化操作。

```
# 卷积-ReLU-池化组合操作
conv_relu_pool_out, cache_conv_relu_pool = conv_relu_pool_forward(
    X, W1, b1, conv_param, pool_param
)
```

之后为全连接层与激活函数，第二个全连接层。

```
# 第一个全连接层 + ReLU
affine1_out, cache_affine1 = affine_forward(conv_relu_pool_out, W2, b2)
affine_relu_out, cache_affine_relu = relu_forward(affine1_out)

# 输出层（第二个全连接层）
scores, cache_affine2 = affine_forward(affine_relu_out, W3, b3)
```

步骤三：损失计算与反向传播

首先计算损失函数（softmax函数），引入L2正则化加入损失函数中。

```
# 计算数据损失（softmax损失）
data_loss, dscores = softmax_loss(scores, y)

# L2正则化损失
reg_loss = 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2) + np.sum(W3**2))

# 总损失
loss = data_loss + reg_loss
```

利用BP计算梯度，包括每个层梯度以及正则化梯度：

```
# 输出层梯度
daffine_relu, grads['W3'], grads['b3'] = affine_backward(dscores, cache_affine2)

# ReLU层梯度
daffine1 = relu_backward(daffine_relu, cache_affine_relu)

# 隐藏层梯度
dconv_relu_pool, grads['W2'], grads['b2'] = affine_backward(daffine1, cache_affine1)

# 卷积层梯度
dX, grads['W1'], grads['b1'] = conv_relu_pool_backward(dconv_relu_pool, cache_conv_relu_pool)

# 添加正则化梯度
grads['W3'] += self.reg * W3
grads['W2'] += self.reg * W2
grads['W1'] += self.reg * W1
```

BP本质上就是每层梯度=上层梯度✖本地梯度

正则化梯度：

- L2正则化的梯度为 $\lambda W$，直接加到原始梯度上
- 确保优化过程中权重向零收缩，实现权重衰减

涉及原理：

> **尺寸计算原理**：
>
> - **池化输出尺寸**：`H // 2`因为使用2×2池化，步长为2，尺寸减半
> - **全连接输入维度**：将多维特征图展平为一维向量的长度
> - 公式：$H_{out} = \lfloor \frac{H}{2} \rfloor, W_{out} = \lfloor \frac{W}{2} \rfloor$ 6

**卷积参数原理**：

• **步长1**：卷积核每次移动1像素，保持高分辨率特征图

• **填充计算**：`pad = (filter_size - 1) // 2`确保输入输出尺寸相同（"SAME"卷积）

• 尺寸保持公式：$H_{out} = \lfloor \frac{H+2p-f}{s} \rfloor + 1$，当$p = \frac{f-1}{2}$且$s = 1$时，$H_{out} = H$ [4]

**池化参数作用**：

• **2×2窗口，步长2**：非重叠池化，每次将特征图尺寸减半

• **最大池化**：保留最显著特征，提供平移不变性 [6]

**全连接层功能**：

• affine_forward：执行线性变换 $y = Wx + b$

• **维度变换**：将空间特征转换为类别得分向量

• **缓存机制**：存储中间结果供反向传播使用

ReLU**激活函数**：

• 公式：$f(x) = \max(0, x)$

• **优点**：解决梯度消失问题，计算高效

• **生物学合理性**：近似神经元的稀疏激活特性 [3]

**Softmax损失原理**：

- 公式：$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$
- **作用**：将类别得分转换为概率分布，衡量预测与真实标签的差异
- **数值稳定性**：实现中通常使用log-sum-exp技巧避免数值溢出 1

**L2正则化**：

- 公式：$R(W) = \frac{1}{2}\lambda \sum w^2$
- **作用**：惩罚大权重，防止过拟合，提高模型泛化能力
- **系数0.5**：简化梯度表达式，求导后系数变为1 1

8.过拟合小数据集（训练模型）

1. 快速验证模型实现是否正确。如果能过拟合小数据集，则强有力地证明你的模型实现基本正确，各组件工作正常，优化器也能成功更新权重。
2. 确认模型拥有足够的容量（Capacity）：模型容量指的是其拟合复杂函数的能力。一个三层卷积网络应该具备足够的能力去学习你任务中的潜在模式。如果能过拟合小数据集，说明模型的容量是足够的，甚至可能对于完整数据集来说也足够了。后续工作的重点就可以从“增加模型复杂度”转向“如何防止过拟合”和“提高泛化能力”。
3. 高效调试，快速迭代。

空间批归一化：

空间批归一化是批归一化在卷积神经网络中的扩展。卷积层的输出是4D张量（N, C, H, W），其中N是批次大小，C是通道数，H和W是空间维度。标准批归一化处理2D数据（N, D），因此需要将空间维度展平，使每个通道的统计量基于所有空间位置计算。

前向传播：

def spatial_batchnorm_forward(x, gamma, beta, bn_param):

N, C, H, W = x.shape # 获取输入张量的维度

x_flat = x.transpose(0, 2, 3, 1).reshape(-1, C)

将4D输入转换为2D矩阵。转置操作将通道维度C移到最后，得到（N, H, W, C），然后重塑为（N*HW*, C）。这样，每个通道的所有空间位置被视为独立样本，符合标准批归一化的输入要求。

**实现**：`transpose(0, 2, 3, 1)` 重新排列维度，`reshape(-1, C)` 自动计算第一维大小（N*HW*）。

out_flat, cache_flat = batchnorm_forward(x_flat, gamma, beta, bn_param)

调用标准批归一化前向传播函数。输入是展平后的2D数据，输出是归一化后的2D数据及缓存（用于反向传播）。

批归一化对每个通道计算均值和方差，进行归一化后应用缩放（gamma）和平移

out = out_flat.reshape(N, H, W, C).transpose(0, 3, 1, 2)

cache = (cache_flat, N, C, H, W)

恢复原来的维度保存cache

反向传播：

def spatial_batchnorm_backward(dout, cache):

cache_flat, N, C, H, W = cache # 解包缓存

dout_flat = dout.transpose(0, 2, 3, 1).reshape(-1, C)

将上游梯度dout（形状为（N, C, H, W））重塑为2D形式（N*H*W, C），以匹配标准批归一化的输入维度。转置和重塑操作与前向传播对称

dx_flat, dgamma, dbeta = batchnorm_backward(dout_flat, cache_flat)

计算梯度x-flat，gamma，beta

dx = dx_flat.reshape(N, H, W, C).transpose(0, 3, 1, 2)

空间组归一化

组归一化将通道分成G组，每组内部计算均值和方差进行归一化。它不依赖批次大小，适用于小批次或动态网络结构（如RNN）。与批归一化不同，组归一化的统计量基于单个样本的组内数据计算。

前向传播：

def spatial_groupnorm_forward(x, gamma, beta, G, gn_param):

eps = gn_param.get('eps', 1e-5) # 获取数值稳定性常数

N, C, H, W = x.shape

assert C % G == 0, "通道数C必须被组数G整除"

检查通道数能否被组数整除，确保每组通道数相等。

x_grouped = x.reshape(N, G, C // G, H, W)

将输入重塑为（N, G, C//G, H, W），其中G是组数，C//G是每组通道数。这样，每组包含连续通道的空间数据。

mean = x_grouped.mean(axis=(2, 3, 4), keepdims=True)

var = x_grouped.var(axis=(2, 3, 4), keepdims=True)

计算每组数据的均值和方差。沿通道（索引2）和空间维度（索引3,4）求平均，保持维度以便广播。

```
x_norm_grouped = (x_grouped - mean) / np.sqrt(var + eps)  # 重新计算归一化数据
```

- **原理**：重新计算归一化值，确保与缓存一致（也可直接缓存）。

python 下载 复制 运行

```
dgamma = np.sum(dout * x_norm, axis=(0, 2, 3), keepdims=True)
    dbeta = np.sum(dout, axis=(0, 2, 3), keepdims=True)
```

- **作用**：计算gamma和beta的梯度。沿批次和空间维度求和，保持维度以匹配参数形状（1, C, 1, 1）。
- **原理**：dgamma是损失对gamma的梯度，等于归一化输入x_norm与上游梯度dout的乘积和；dbeta是dout的和 8 。

python 下载 复制 运行

```
dx_norm = dout * gamma  # 上游梯度通过gamma缩放
    dx_norm_grouped = dx_norm.reshape(N, G, C // G, H, W)
```

- **原理**：将梯度分组处理，以便按组计算统计量梯度。

```python
sigma = np.sqrt(var + eps)  # 标准差
    sum_dx_norm = np.sum(dx_norm_grouped, axis=(2, 3, 4), keepdims=True)
    sum_dx_norm_x_norm = np.sum(dx_norm_grouped * x_norm_grouped, axis=(2, 3, 4), keepdims=True)
```

- **作用**：计算组内梯度的求和项。这些项用于合并梯度路径（均值、方差贡献） [8] 。
- **原理**：类似批归一化，但限制在组内。

```python
dx_grouped = (1 / (M * sigma)) * (
        M * dx_norm_grouped -
        sum_dx_norm -
        x_norm_grouped * sum_dx_norm_x_norm
    )
```

- **原理**：这是组归一化的梯度简化公式。三项分别对应：
  - $M \cdot dx\_norm\_grouped$：直接路径梯度。
  - $-sum\_dx\_norm$：均值路径的梯度贡献。
  - $-x\_norm\_grouped \cdot sum\_dx\_norm\_x\_norm$：方差路径的梯度贡献。

  公式确保梯度正确传播通过归一化操作 [8] 。

```python
dx = dx_grouped.reshape(N, C, H, W)
```

- **作用**：将组梯度恢复为原始4D形状。

| 技术 | 计算维度 | 适用场景 | 优点 |
| --- | --- | --- | --- |
| **批归一化** (BatchNorm) | 跨样本的同一特征通道 | 大批次训练，稳定训练过程 | 加速收敛，减少内部协变量偏移 |
| **层归一化** (LayerNorm) | 单样本的所有特征 | RNN、Transformer、小批次场景 | 不依赖批次大小，适合序列模型 |
| **组归一化** (GroupNorm) | 单样本的特征组 | 卷积网络、小批次训练、检测/分割任务 | 结合两者优点，平衡统计稳定性与假设合理性 |

层归一化在卷积网络中的表现不如批归一化，**根本原因**：在全连接层中，所有隐藏单元对最终预测的贡献相似，但在卷积网络中，**靠近图像边界的隐藏单元很少被激活**，与同一层中其他单元具有非常不同的统计特性（样本某些部分不重要不应该关注，而层归一化很关注）。

组归一化提出了一个**折中方案**：

- 不像层归一化那样在整个特征上进行归一化
- 而是将每个样本的特征**分成G个组**，在每个组内进行归一化

组归一化的优点：

1.平衡了统计假设层归一化假设（一个样本内所有特征贡献相似❌（在CNN中不成立）），批归一化假设（不同样本的同一特征统计特性相似✅（但需要大批次））组归一化取中间（一个样本内相关特征组贡献相似✅（更合理的折中））

2.视觉识别中特征存在内在分组

**HOG特征**：在传统计算机视觉中，先计算局部块的直方图，然后对每个块进行归一化

**卷积特征**：相邻的通道往往检测相关的模式（如不同方向的边缘）

3.不依赖于批次

四、pytorch

1.pytorch作用与学习

介绍：PyTorch是一个在Tensor对象上执行动态计算图的系统，其行为类似于numpy narray。它配备了一个强大的自动区分引擎，消除了手动反向传播的需要。

优点：直接运行在GPU，模型训练的快

直接使用框架而无需编写相应的代码

目录：

Barebones PyTorch：抽象级别1，我们将使用最低级别的PyTorch张量。

第三部分，PyTorch模块API：抽象级别2，我们将使用nn。模块来定义任意的神经网络架构。

第四部分，PyTorch顺序API：抽象级别3，我们将使用nn。顺序定义一个线性前馈网络非常方便。

| API | Flexibility | Convenience |
| --- | --- | --- |
| Barebone | High | Low |
| `nn.Module` | High | Medium |
| `nn.Sequential` | Low | High |

2. Barebones PyTorch

Flatten函数：

```
def flatten(x):
    N = x.shape[0] # 读取N。N为批次大小
    return x.view(N, -1) # 将每个图像的C * H * W值展平为单个向量
```

`x.view(N, -1)`：这是展平操作的核心。

- `view()`是PyTorch中用于改变张量形状的方法，它返回一个与原始张量共享数据的新张量，但具有不同的形状。
- 参数 `N` 表示保持批处理维度不变。

- 参数 `-1` 是一个特殊值，表示让PyTorch自动计算该维度的大小，计算依据是确保新张量的总元素数与原始张量一致。例如，如果一个张量形状为 `(2, 3, 32, 32)`（总元素数 = 2×3×32×32 = 6144），使用 `x.view(2, -1)` 后，形状将变为 `(2, 3072)`，因为 2 × 3072 = 6144。

x = torch.arange(12).view(2, 1, 3, 2)

首先前部分生成0到11的一维张量，重塑为2×1×3×2张量

- 卷积层：需要保持数据的空间结构（`C, H, W`），以便识别局部特征，如图像中的边缘、纹理。
- 全连接层：需要将每个数据点表示为单个向量，不再需要通道、行、列的分离结构。展平操作将每个图像的 `C × H × W` 个值合并成一个一维向量，作为全连接层的输入。

PyTorch提供了几种展平张量的方式，以下是主要方法的对比 1 7 ：

| 方法 | 类型 | 特点 | 适用场景 |
| --- | --- | --- | --- |
| **`x.view(N, -1)`** | 函数/方法 | 高效，共享内存，但要求张量连续 | 已知张量连续，追求性能 |
| **`torch.flatten(x, start_dim=1)`** | 函数 | 高级API，更安全，自动处理维度 | 通用场景，代码清晰 1 |
| **`nn.Flatten(start_dim=1)`** | 网络模块 | 可作为神经网络的一部分 | 在 `nn.Sequential`中使用 2 |

## Two-Layer Network:

代码：

def two_layer-fc(x,params):

`全连接层 → ReLU激活函数 → 全连接层`。

输出十分类的分类得分，输入是批次图像数据和模型参数列表

只需要前向传播，pytorch会自动处理反向传播

x=flatten（x）全连接层

w1，w2=params

前向过程：

x=F.relu(x.mm(w1))

x=x.mm(w2)

return x

下边代码可以进行测试：

def two_layer_fc_test():

hidden_layer_size = 42

x = torch.zeros((64, 50), dtype=dtype) # 模拟一个批量的数据

```
w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
scores = two_layer_fc(x, [w1, w2])
print(scores.size())  # 应该输出: torch.Size([64, 10])
```

判断张量形状，有助于发现维度不匹配等错误

## Three-Layer ConvNet：

| 代码层 | 操作目的 | 输入数据形状 (N, C, H, W) | 输出数据形状 | 关键变化说明 |
|---|---|---|---|---|
| 输入 | 图像数据输入 | (N, 3, H, W) | - | N:批大小, 3:RGB通道, H:高, W:宽 |
| **Conv1** | 特征提取(初级) | (N, 3, H, W) | (N, 32, H, W) | 使用32个5x5滤波器, padding=2保持尺寸 |
| **ReLU1** | 引入非线性 | (N, 32, H, W) | (N, 32, H, W) | 激活函数, 保持形状不变 |
| **Conv2** | 特征提取(高级) | (N, 32, H, W) | (N, 16, H, W) | 使用16个3x3滤波器, padding=1保持尺寸 |
| **ReLU2** | 引入非线性 | (N, 16, H, W) | (N, 16, H, W) | 激活函数, 保持形状不变 |
| **Flatten** | 展平特征图 | (N, 16, H, W) | (N, 16*H*W) | 将空间特征转换为全连接层所需的向量 |
| **FC Layer** | 计算分类得分 | (N, 16*H*W) | **(N, 10)** | 输出10个类别对应的得分(raw logits) |

代码:

```
def three_layer_convnet(x, params):
conv_w1,conv_b1,conv_w2,conv_b2,fc_w,fc_b=params
```

网络层的权重与偏置，可以使用kaiming正态分布权重和0偏置进行初始化

```
conv1=F.conv2d(x,weight=conv_w1,bias=conv_b1,stride=1,padding=2)
```

权重为channe1，3，k1，w1 偏置为channe1padding为2在5×5的卷积核保证输出高度宽度一样

```
relu1=F.relu(conv1)
conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, stride=1, padding=1)
```

权重（滤波器大小）:channel2,channel1,KH2,KW2 padding为1保证3×3卷积之后尺寸保持不变

```
relu2 = F.relu(conv2)
relu2_flat = flatten(relu2)
scores = relu2_flat.mm(fc_w) + fc_b
```

最终输出是"原始得分"。在训练时，这些得分会直接传递给 `F.cross_entropy` 损失函数，该函数会同时完成Softmax和交叉熵损失的计算

## 初始化代码：

def random_weight(shape):

if len(shape) == 2:  # FC weight一般为输出乘以输入维度

    fan_in = shape[0] #（输入单元数）

  else:

    fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH, kW]，fan_in=inchannel*kH*kW

Kaiming 初始化通过 `fan_in` 来调整权重的方差，旨在解决深度网络中的梯度消失或爆炸问题，其核心思想是保持数据在层间流动时方差稳定

w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)

`torch.randn(shape, device=device, dtype=dtype)`：从标准正态分布（均值为0，方差为1）中随机采样，生成指定形状的张量。

`np.sqrt(2. / fan_in)`：这是 Kaiming 初始化的关键缩放因子。对于使用 ReLU 及其变体的激活函数，为了补偿 ReLU 将一半负值置零导致的前向传播信号方差减半效应，需要将权重的方差设置为 `2 / fan_in`，对应的标准差就是 `sqrt(2 / fan_in)`。这样做的目标是确保网络各层的激活值在正向传播过程中保持稳定的方差，通常设为1 。

w.requires_grad = True

  return w

偏置的初始化：

def zero_weight(shape):

  return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

设计理由：对于偏置项，常见的做法是初始化为零。这样在训练初期，每个神经元的输出由其权重和输入决定，偏置项从零开始学习调整 。将偏置初始化为零并与权重采用不同的初始化策略，是一种打破对称性、促进稳定训练的有效方法 。

## 检查准确率代码：

def check_accuracy_part2(loader, model_fn, params):

判断是测试集还是训练集

split = 'val' if loader.dataset.train else 'test'

print('Checking accuracy on the %s set' % split)

初始化

num_correct, num_samples = 0, 0

with torch.no_grad():#禁用梯度

循环：

for x, y in loader:

  x = x.to(device=device, dtype=dtype)  # 迁移到指定设备

  y = y.to(device=device, dtype=torch.int64)

确保数据与模型在同一设备

scores = model_fn(x, params)

_, preds = scores.max(1)

选择分数最大作为类被，1代表按照第一个维度取最大值

num_correct += (preds == y).sum()布尔掩码

num_samples += preds.size(0)

acc = float(num_correct) / num_samples

print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * acc))

## Training Loop：

| | | |
|---|---|---|
| 1. **数据准备** | `for t, (x, y) in enumerate(loader_train):` | 从数据加载器中获取批次的训练数据 |
| 2. **设备迁移** | `x.to(device=device, dtype=dtype)` | 将数据移动到指定设备（如GPU） |
| 3. **前向传播** | `scores = model_fn(x, params)` | 计算模型的预测输出 |
| 4. **损失计算** | `loss = F.cross_entropy(scores, y)` | 计算预测与真实值之间的差异 |
| 5. **反向传播** | `loss.backward()` | 自动计算梯度（PyTorch自动求导） |
| 6. **参数更新** | `w -= learning_rate * w.grad` | 使用梯度下降更新模型参数 |
| 7. **梯度清零** | `w.grad.zero_()` | 重置梯度为零，准备下一轮计算 |

代码：

def train_part2(model_fn, params, learning_rate):

迭代循环：

for t, (x, y) in enumerate(loader_train):

x = x.to(device=device, dtype=dtype)

y = y.to(device=device, dtype=torch.long)

将数据与标签移动到指定的计算设备GPU，保证在同一个设备中

scores = model_fn(x, params)

loss = F.cross_entropy(scores, y)，执行前向，并且利用softmax作为loss函数

loss.backward()

with torch.no_grad():#

- 上下文管理器，确保参数更新操作不被记录在计算图中，节省内存

  for w in params:

    w -= learning_rate * w.grad

    w.grad.zero_()#将梯度重置为零，防止梯度在不同批次间累积

if t % print_every == 0:

  print('Iteration %d, loss = %.4f' % (t, loss.item()))

  check_accuracy_part2(loader_val, model_fn, params)

  print()

- `with torch.no_grad()` 上下文管理器在参数更新时非常重要，它防止不必要的计算图构建，节省内存
- 梯度清零（`zero_grad()`）是必要的，因为默认情况下PyTorch会累积梯度

Barebone PyTorch要求我们手动跟踪所有参数张量。这对于只有几个张量的小型网络来说是很好的，但是在大型网络中跟踪数十或数百个张量将是非常不方便和容易出错的。

PyTorch提供了nn。模块的API为您定义任意网络架构，同时跟踪每一个可学习的参数，可以实现了所有常见的优化器

3. PyTorch Module API

1.子类“nn.Module”。给你的网络类起一个直观的名字，比如“TwoLayerFC”。

2.在构造函数‘**init**()’中，将所需的所有层定义为类属性。像nn这样的图层对象。nn.linear、nn.Conv2d是nn.module块的子类，并包含可学习的参数，因此您不必自己实例化原始张量。nn.module块将为您跟踪这些内部参数。

3.在‘forward()’方法中，定义网络的连接性。您应该使用‘**init**’中定义的属性作为函数调用，将张量作为输入并输出“转换”的张量。不要在‘forward()’中创建任何具有可学习参数的新层！所有这些都必须在‘**init**’中预先声明

TwoLayerFC：

代码：

class TwoLayerFC(nn.Module):

```
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
```

网络层定义与初始化：

```
self.fc1 = nn.Linear(input_size, hidden_size)
nn.init.kaiming_normal_(self.fc1.weight)
```

- **nn.Linear**：全连接层（线性变换），数学形式为 y = xW^T + b
  - input_size：输入特征维度
  - hidden_size：隐藏层大小（神经元数量）
- **nn.init.kaiming_normal_**：使用 Kaiming He 正态分布初始化权重
  - 特别适合 ReLU 激活函数，能保持信号方差稳定
  - 初始化公式：W \sim \mathcal{N}(0, \sqrt{\frac{2}{\text{fan_in}}})

```
self.fc2 = nn.Linear(hidden_size, num_classes)
nn.init.kaiming_normal_(self.fc2.weight)
```

前向传播方法：

```
def forward(self, x):
    x = flatten(x)
    scores = self.fc2(F.relu(self.fc1(x)))
    return scores
```

| 特性 | Barebone PyTorch | nn.ModuleAPI |
| --- | --- | --- |
| **参数管理** | 手动维护参数列表 | 自动追踪所有参数 |
| **设备迁移** | 手动移动每个参数 | model.to(device)一键迁移 |
| **序列化** | 手动保存/加载 | torch.save(model.state_dict()) |
| **层定义** | 手动定义权重矩阵 | 使用内置层如 nn.Linear |
| **前向传播** | 自定义函数 | 在 forward方法中定义 |
| **子模块** | 难以嵌套 | 支持嵌套子模块 |

1. **层定义在 __init__中**:
   - 所有包含可学习参数的层必须在 __init__中定义
   - 这样 PyTorch 才能正确追踪参数
2. **计算逻辑在 forward中**:
   - forward定义数据流动路径
   - 可以使用 Python 控制流（条件、循环）
3. **参数初始化**:
   - 良好的初始化对训练成功至关重要
   - Kaiming 初始化适合 ReLU 激活函数
4. **模块化设计**:
   - 可以嵌套其他 nn.Module构建复杂网络
   - 支持模型复用和共享

Three-Layer ConvNet：

```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight, mode='fan_out', nonlinearity='relu')#fan_in关注前一次输入当前连接数量主要稳定前向传播，fanout有助于稳定反向传播，卷积层一般都是fanout决定了反向传播梯度规模，是梯度更稳定，nonlinearity告诉使用什么激活函数，调正相应的方差保证稳定，reLU激活函数需要Kaming初始化，引入一个增益因子
        nn.init.constant_(self.conv1.bias, 0)#将卷积层的偏置初始化为0
        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1)
        self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)
    def forward(self, x):
        scores = None
        x = F.relu(self.conv1(x))     # 卷积1 + ReLU
        x = F.relu(self.conv2(x))     # 卷积2 + ReLU
        x = flatten(x)              # 展平操作
        scores = self.fc(x)         # 全连接层
        return scores
def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

4.PyTorch Sequential API

> **Sequential API 的核心优势**
>
> `nn.Sequential`是 PyTorch 中的一个**顺序容器**，它可以按顺序将多个网络层组合成一个完整的模型。与传统的 `nn.Module`子类化方式相比，它的主要优势在于**代码简洁性**——您不需要分别定义 `__init__`和`forward`方法，而是直接按顺序列出所有层即可 [1] [2] 。
>
> | 特性 | 传统 nn.Module | nn.Sequential |
> | --- | --- | --- |
> | **代码量** | 较多（需定义类和forward方法） | 极少（直接列出层） |
> | **灵活性** | 高（支持复杂连接） | 有限（仅顺序连接） |
> | **可读性** | 中等 | 高（结构一目了然） |
> | **适用场景** | 复杂网络架构 | 简单顺序结构 |

```
class Flatten(nn.Module):
    def forward(self, x):
        return x.view(x.size(0), -1)


model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2, bias=True),  # 卷积层1
    nn.ReLU(),                                  # 激活函数1
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1, bias=True),  # 卷积层2
    nn.ReLU(),                                  # 激活函数2
    Flatten(),                                 # 展平层
    nn.Linear(channel_2 * 32 * 32, 10, bias=True)            # 全连接层
)
optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, nesterov=True)
```

通过 `padding=2` （5×5卷积）和 `padding=1` （3×3卷积）确保卷积操作不改变特征图空间尺

全连接层输入维度必须准确计算为 `channel_2 × H × W = 16 × 32 × 32 = 16384`

由于PyTorch的 `nn.Sequential` 需要所有组件都是 `nn.Module` 子类，因此需要将展平操作封装为自定义层

五、RNN_Captioning_pytorch

1.针对图像描述，首先数据集选择，图像特征的提取与处理，文本描述编码方式：

| 组成部分 | 描述 | 作用与目的 |
| --- | --- | --- |
| **数据集 (COCO 2014)** | 包含8万张训练图像和4万张验证图像，每张图有5句人工标注的描述 [1] 。 | 为模型提供高质量的图像-文本配对数据，是模型学习的基础 |
| **图像特征 (VGG-16 fc7)** | 使用在ImageNet上预训练好的VGG-16网络，提取图像的特征向量（原始为4096维，后经PCA降至512维） [1] 。 | 将图像转化为计算机可处理的数值向量，降维后能显著减少存占用。 |
| **描述编码 (整数ID)** | 将每个单词映射为一个唯一的整数ID，从而将文本描述转换为整数序列 [1] 。 | 将文本转化为模型能够直接处理的数值形式，提高处理效率。 |
| **特殊标记** | 引入<START>, <END>, <UNK>, <NULL>等特殊标记 [1] 。 | 标示句子开始/结束，处理稀有词汇，统一批次内序列长度，进行的关键。 |

- `<START>` 和 `<END>`：为模型提供明确的句子边界。在训练时，模型学习以上一个单词预测下一个单词。`<START>` 作为生成第一个词的“上一个词”，`<END>` 让模型学会在何时停止生成。
- `<UNK>`：任何在词汇表中出现频率低于某个阈值的词都会被替换为此标记。这能有效控制词汇表的大小，避免模型为大量生僻词分配参数，同时防止模型遇到生僻词时“不知所措”。
- `<NULL>`：为了将不同长度的句子组合成一个批次（Batch）进行并行计算，需要将所有句子填充到同一长度。`<NULL>` 就是用于填充的占位符。关键的一点是，在计算损失时，会屏蔽（Mask）这些填充位置，确保模型不会因为学习了无意义的填充符而降低性能。

2.Vanilla RNN: Step Forward

仿射变换（全连接层）

```
import torch

def affine_forward(x, w, b):
    out = x.reshape(x.shape[0], -1) @ w + b
    return out
```

`x.reshape(x.shape[0], -1)` 将输入 `x`（通常形状为 `(N, d1, d2, ...)`）重塑为二维矩阵 `(N, D)`，其中 `D` 是所有维度特征的乘积。这样可以将每个样本转换为一个向量，以便进行矩阵乘法。

RNN在一个时间步上的前向计算

```
def rnn_step_forward(x, prev_h, Wx, Wh, b):
    affline = x @ Wx + prev_h @ Wh + b
    next_h = torch.tanh(affline) 压缩值到-1到1，利用梯度稳定
    return next_h
```

整个序列展开RNN

```
def rnn_forward(x, h0, Wx, Wh, b):
    N, T, D = x.shape
    H = h0.shape[1]
```

h = torch.zeros((N, T, H), dtype=x.dtype, device=x.device)

prev_h = h0

for t in range(T):

x_t = x[:, t, :] # 获取第t个时间步的输入，形状为(N, D)

prev_h = rnn_step_forward(x_t, prev_h, Wx, Wh, b)

h[:, t, :] = prev_h

return h

- 它接收整个输入序列 `x` （形状为 `(N, T, D)`）和初始隐藏状态 `h0` （形状为 `(N, H)`）。
- 函数初始化一个全零张量 `h` 来存储所有时间步的隐藏状态。
- 通过循环遍历每个时间步 `t`，依次调用 `rnn_step_forward`，并将每个时间步得到的隐藏状态存入 `h` 的对应位置。
- 最终返回所有时间步的隐藏状态 `h`，形状为 `(N, T, H)`

词嵌入查找层：

def word_embedding_forward(x, W):

out = W[x]

return out

- 输入 `x` 是一个整数张量，包含单词的索引，形状通常为 `(N, T)`。
- `W` 是词嵌入矩阵，形状为 `(V, D)`，其中 `V` 是词汇表大小，`D` 是词向量的维度。
- `W[x]` 使用高级索引操作，根据 `x` 中的每个索引从 `W` 中取出对应的 `D` 维词向量。输出 `out` 的形状为 `(N, T, D)`

辅助函数sigmoid：

def sigmoid(x):

return torch.sigmoid(x)

LSTM在一个时间步的前向计算:

def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b):

H = prev_h.shape[1]

a = x @ Wx + prev_h @ Wh + b # (N, 4H)

# 将激活结果分割成四个部分，对应输入门、遗忘门、输出门和候选细胞状态

ai = a[:, :H] # 输入门激活

af = a[:, H:2*H] # 遗忘门激活*

*ao = a[:, 2*H:3*H] # 输出门激活*

```
    ag = a[:, 3H:4*H] # 候选细胞状态激活
    i = sigmoid(ai)  # 输入门
    f = sigmoid(af)  # 遗忘门
    o = sigmoid(ao)  # 输出门
    g = torch.tanh(ag)  # 候选细胞状态
    next_c = f * prev_c + i * g  # 新细胞状态：遗忘部分旧状态，加入部分新信息
    next_h = o * torch.tanh(next_c)  # 新隐藏状态：基于新细胞状态输出
    return next_h, next_c
```

- 首先，将当前输入 `x` 和上一隐藏状态 `prev_h` 通过仿射变换并结合偏置，得到一个大的激活张量 `a` （形状为 `(N, 4H)` ）。
- 将这个激活张量分割成四部分，分别对应输入门（i）、遗忘门（f）、输出门（o）和候选细胞状态（g）。
- 三个门使用 sigmoid 函数，输出在0到1之间，控制信息流通的比例。候选细胞状态使用 tanh 函数，提供新的候选信息。
- 新细胞状态 `next_c` 由两部分组成： `f * prev_c` （遗忘门控制保留多少旧记忆）和 `i * g` （输入门控制加入多少新信息）。
- 新隐藏状态 `next_h` 由输出门 `o` 控制，基于新细胞状态 `next_c` 的 tanh 激活值输出。

```
def lstm_forward(x, h0, Wx, Wh, b):
    N, T, D = x.shape
    H = h0.shape[1]
    h = torch.zeros((N, T, H), dtype=x.dtype, device=x.device)
    prev_h = h0
    prev_c = torch.zeros((N, H), dtype=h0.dtype, device=h0.device) # 细胞状态初始化为零
    for t in range(T):
        x_t = x[:, t, :]
        next_h, next_c = lstm_step_forward(x_t, prev_h, prev_c, Wx, Wh, b)
        h[:, t, :] = next_h
        prev_h = next_h
        prev_c = next_c
    return h
```

`temporal_affine_forward` 函数是循环神经网络（RNN）中一个非常关键的操作，它的核心作用是在每个时间步上将RNN的隐藏状态映射到整个词汇表上，从而生成每个单词的得分。这些得分后

续会通过Softmax函数转换为概率，以决定在当前位置最可能生成哪个词。

| | | |
|---|---|---|
| **输入** | `x` | RNN所有时间步的隐藏状态，形状为 (N, T, D)。其中 N是批次大小，T是序列长度（时间步数），D是隐藏层的维度。 |
| | `w` | 权重矩阵，形状为 (D, M)。M是词汇表的大小。它的作用是将一个 D维的隐藏向量变换为 M维的分数向量，每个维度对应词汇表中一个单词的得分。 |
| | `b` | 偏置项，形状为 (M,)。为每个单词的得分添加一个基础值。 |
| **核心计算** | `x.reshape(N * T, D)` | **形状变换**：将三维的输入张量 x重塑为二维矩阵 (N * T, D)。这一步的目的是将批次和时间的维度合并，以便利用高效的矩阵乘法，对所有样本和所有时间步进行**并行计算**。 |
| | `@ w` | **矩阵乘法**：变换后的输入与权重矩阵 w相乘，结果是一个形状为 (N * T, M)的矩阵。这相当于对 N*T个 D维向量分别进行了仿射变换。 |
| | `.reshape(N, T, M)` | **形状恢复**：将矩阵乘法的结果重新调整为 (N, T, M)的三维张量，恢复批次和时间步的维度。现在，`out[i, t, :]`就代表了第 i个样本在第 t个时间步上，词汇表所有 M个单词的得分。 |
| | `+ b` | **添加偏置**：最后，将偏置项 b加到每个单词的得分上。这里利用了PyTorch/TensorFlow等框架的**广播机制**，偏置会自动扩展到 (N, T, M)的维度。 |
| **输出** | `out` | 每个时间步上所有单词的得分，形状为 (N, T, M)。 |