作业一：

1.KNN

kNN分类器包括两个阶段：

在训练过程中，分类器获取训练数据并简单地记住它

在测试过程中，kNN通过与所有训练图像进行比较，并传递k个最相似训练样例的标签，对每个测试图像进行分类

代码：

```
from builtins import range  from builtins import object import numpy as np
from past.builtins import xrange
class KNearestNeighbor(object):
  """ a kNN classifier with L2 distance """
  def init(self):
    pass
```

训练方法：记忆数据

```
def train(self, X, y):
  self.X_train = X
  self.y_train = y
```

预测流程：

```
def predict(self, X, k=1, num_loops=0):
  if num_loops == 0:
    dists = self.compute_distances_no_loops(X)
  elif num_loops == 1:
    dists = self.compute_distances_one_loop(X)
  elif num_loops == 2:
    dists = self.compute_distances_two_loops(X)
  else:
    raise ValueError("Invalid value %d for num_loops" % num_loops)
  return self.predict_labels(dists, k=k)#predict_labels进行标签预测
```

距离实现方式：

```
def compute_distances_two_loops(self, X):
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        for j in range(num_train):
            dists[i, j] = np.sqrt(np.sum((X[i] - self.X_train[j]) ** 2))
    return dists
```

时间复杂度为O(num_test * num_train)

```
def compute_distances_one_loop(self, X):
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        dists[i, :] = np.sqrt(np.sum((X[i] - self.X_train) ** 2, axis=1))
    return dists
```

无循环向量化：

```
def compute_distances_no_loops(self, X):
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test,num_train))#X[:,np.newaxis]将测试数据维度从(num_test, D)变为(num_test, 1, D)
    dists=np.sqrt(np.sum((X[:,np.newaxis]-self.X_train)**2,axis=2))#axis沿哪个维度求和开根号
    return dists
```

标签预测：多数表决

```
def predict_labels(self, dists, k=1):
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in range(num_test):
        closest_y = self.y_train[np.argsort(dists[i])[:k]]
```

```
        y_pred[i] = np.argmax(np.bincount(closest_y))
    return y_pred
```

1. 对每个测试样本`i`，使用`np.argsort(dists[i])`对距离数组进行排序，返回的是索引值。
2. 取前`k`个最小距离的索引，用这些索引从训练标签`self.y_train`中获取对应的标签`closest_y`。
3. `np.bincount`统计`closest_y`中每个标签出现的次数。
4. `np.argmax`找出出现次数最多的标签，作为该测试样本的预测结果 [2] [4] [5] 。

- **投票机制**：这就是KNN中的"多数表决"规则，k个最近邻居中占多数的类别获胜 [7] 。

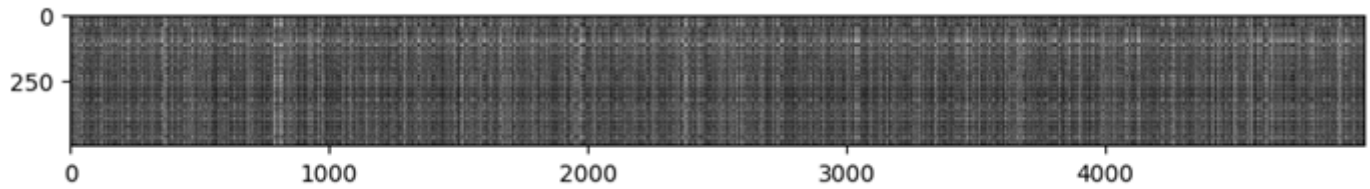
KNN的优势：

- **简单直观**：算法原理容易理解，实现简单 [8] 。
- **无需训练**：训练阶段只是存储数据，非常快速。
- **自然处理多分类**：天生支持多类别分类问题。

KNN的缺点：

- **预测速度慢**：测试时需要计算与所有训练样本的距离，当训练集很大时，计算成本高 [6] [8] 。
- **内存消耗大**：需要存储整个训练数据集。
- **对不平衡数据敏感**：如果某个类的样本数量过多，可能会主导预测结果 [6] 。
- **特征尺度敏感**：如果特征量纲不一，需要先进行标准化处理 [7] 。

KNN：



Q1：Notice the structured patterns in the distance matrix, where some rows or columns are visibly brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

What in the data is the cause behind the distinctly bright rows?

What causes the columns?

A1：量行原因：

1. 测试集样本，比较模糊，或者比较特别导致不属于任何训练样本
2. 测试集样本的同一类在训练集数量很少代表性不足

量列原因：

3. 异常训练样本(标注错误，含噪音，物体特征不明显，背景等)

Q2：

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i, j) of some image I_k,

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw} \sum_{k=1}^{n} \sum_{i=1}^{h} \sum_{j=1}^{w} p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n} \sum_{k=1}^{n} p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. To clarify, both training and test examples are preprocessed in the same way.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

选择123

1. Subtracting the mean μ（减去全局均值μ）

- 操作：每个像素值减去相同的全局均值μ，即 $\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$。
- 对L1距离的影响：对于两个图像x和y，预处理后L1距离为∑|(x_i - μ) - (y_i - μ)| = ∑|x_i - y_i|，距离保持不变。
- **性能不变**：因为距离值不变，最近邻关系不变。

2. Subtracting the per pixel mean μ_ij（减去每个像素的均值μ_ij）

- 操作：每个像素位置减去对应的均值μ_ij，即 $\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$。
- 对L1距离的影响：预处理后L1距离为∑|(x_i - μ_i) - (y_i - μ_i)| = ∑|x_i - y_i|，距离保持不变（μ_i是每个像素的常数）。
- **性能不变**：因为距离值不变，最近邻关系不变。

3. Subtracting the mean μ and dividing by the standard deviation σ（减去全局均值μ并除以全局标准差σ）

- 操作：全局标准化，即 $\tilde{p}_{ij}^{(k)} = (p_{ij}^{(k)} - \mu)/\sigma$，其中σ是全局标准差。
- 对L1距离的影响：预处理后L1距离为∑|(x_i - μ)/σ - (y_i - μ)/σ| = ∑|(x_i - y_i)/σ| = (1/σ) ∑|x_i - y_i|。所有距离缩放相同因子1/σ（σ>0），相对顺序不变。
- **性能不变**：因为距离缩放一致，最近邻关系不变。

4. Subtracting the pixel-wise mean μ_ij and dividing by the pixel-wise standard deviation σ_ij（减去每个像素的均值μ_ij并除以每个像素的标准差σ_ij）

- 操作：像素级标准化，即 $\tilde{p}_{ij}^{(k)} = (p_{ij}^{(k)} - \mu_{ij})/\sigma_{ij}$。
- 对L1距离的影响：预处理后L1距离为∑|(x_i - μ_i)/σ_i - (y_i - μ_i)/σ_i| = ∑|x_i - y_i| / σ_i。每个维度除以不同的σ_i，改变了不同像素对距离的权重（高标准差像素的贡献变小）。
- **性能可能改变**：因为距离权重变化，最近邻关系可能改变。

5. Rotating the coordinate axes of the data（旋转坐标轴，即旋转图像）

- 操作：所有图像旋转相同角度，空白区域用相同像素值填充，不插值。
- 对L1距离的影响：旋转改变了像素位置，L1距离基于像素值对应位置计算。旋转后，像素对应关系破坏（例如，原本相同位置的像素可能不再对齐），距离计算无效。
- **性能可能改变**：因为像素错位，距离值显著变化，最近邻关系可能改变。

Q3：

1.k-NN分类器的决策边界是非线性

2.使用k-NN分类器对测试样例进行分类所需的时间随着训练集的大小而增长。

3.1-NN的训练误差总是小于或等于5-NN的训练误差。

4.1-NN的测试误差总是小于5-NN的测试误差。

判断：错误。

理由：测试误差取决于过拟合和欠拟合。k=1时，容易过拟合训练数据，导致测试误差可能很高；k=5时，由于平滑效应，可能减少过拟合，从而测试误差可能更低。因此，1-NN的测试误差并不总是小于5-NN的测试误差；实际上，对于许多数据集，5-NN的测试误差更小。

2.softmax

目标

1.为Softmax分类器实现一个完全矢量化的损失函数。

2.实现其解析梯度的完全矢量化表达式

3.使用数值梯度检查您的实现

4.使用验证集来调整学习率和正则化强度

5.用SGD优化损失函数

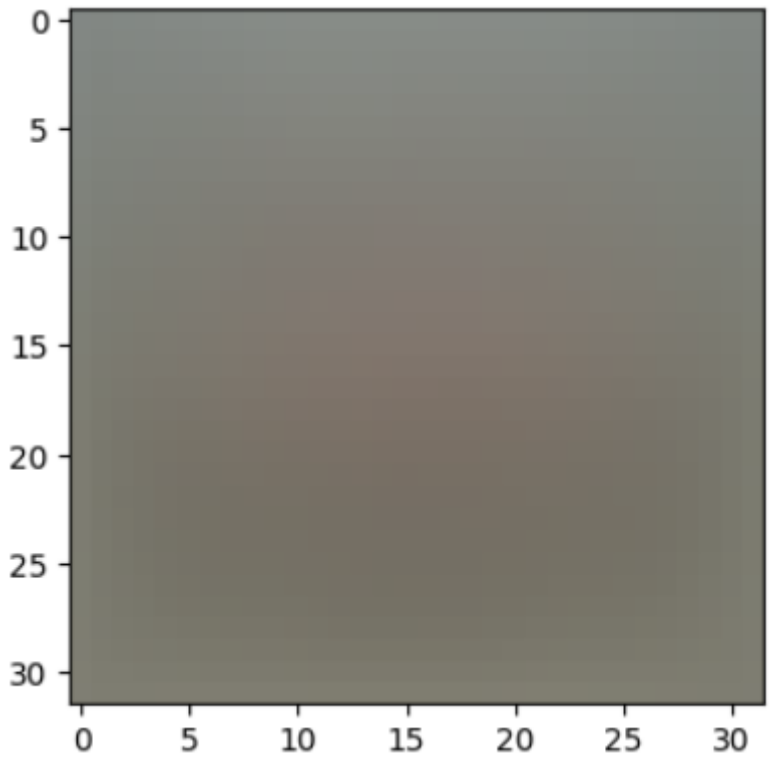
6.想象最后学到的权重

数据集形状(CIFAR-10 data)：

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)

Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

训练集	49,000	从原始训练集的前num_training个样本中提取。	**模型学习的教材**：用于**训练模型参数**，让模型从中学习特征与标签之间的映射关系。 6 8
验证集	1,000	从原始训练集的第49,000到50,000个样本中提取。	**模型调试的考官**：用于**调整超参数**（如KNN中的K值）和**模型选择**，评估模型在未参与训练的数据上的表现，防止过拟合。 6 7
测试集	1,000	从原始测试集的前num_test个样本中提取。	**最终成果的检验官**：仅在模型完全确定后使用，用于**最终、无偏地评估模型的泛化能力**，模拟模型在真实世界中的表现。 6 8
开发集	500	通过随机无放回抽样从最终的训练集中抽取num_dev个样本。	**开发阶段的"快进"键**：作为训练集的一个小型子集，在代码开发和调试阶段使用，以**大幅缩短实验迭代时间**。 7

对数据进行减去均值（可视化如下）



加入偏置项：

X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])

X的矩阵多加一列进行拼接起来，最后一列全为1即是偏置项

原始公式：`y = Wx + b`

新公式：`y = W'x'`

只需要优化权重矩阵W,不用分开优化了

Softmax分类器：

Softmax 函数与交叉熵损失

Softmax函数将分类模型的原始输出分数转换为概率分布，再结合交叉熵损失，共同衡量模型预测与真实标签的差异 6 。

1. **概率转换**：对于每个样本，其每个类别 j 的预测概率为 $p_j = \frac{e^{s_j}}{\sum_k e^{s_k}}$，其中 s_j 是该样本在类别 j 上的得分。
2. **损失计算**：交叉熵损失关注正确类别的预测概率，损失值为 $L_i = -\log(p_{y_i})$，其中 y_i 是样本 i 的真实标签。我们希望正确类别的预测概率尽可能大（接近1），这样损失就会小。

循环版本：

```
def softmax_loss_naive(W, X, y, reg):
  loss = 0.0
  dW = np.zeros_like(W)  # 初始化梯度矩阵，形状同W
  num_classes = W.shape[1]
  num_train = X.shape[0]
```

W初始化为与权重矩阵 W同形的全零矩阵，用于累积每个样本贡献的梯度。

```
for i in range(num_train):
    scores = X[i].dot(W)  # 计算单个样本的得分向量
    scores -= np.max(scores, keepdims=True)  # 数值稳定性处理
    p = np.exp(scores)  # 计算指数
    p /= p.sum()  # 计算概率
    logp = np.log(p)  # 计算对数概率
    loss -= logp[y[i]]  # 累加损失
```

- 对于每个样本 `i`，先计算其各类别得分 `scores`。
- 数值稳定性处理：得分向量减去其最大值（`scores -= np.max(scores)`），防止指数运算时数值过大溢出。

- 计算Softmax概率 `p`，即归一化的指数得分。
- 损失累加正确类别 *y[i]*的负对数概率 `-logp[y[i]]`。

def softmax_loss_vectorized(W, X, y, reg):

  loss = 0.0

  dW = np.zeros_like(W)

  num_train = X.shape[0]

  scores = X.dot(W) # 向量化计算所有样本得分矩阵 (N, C)

  scores -= np.max(scores, axis=1, keepdims=True) # 按行求最大值进行数值稳定处理

- 使用矩阵乘法 `X.dot(W)` 一次性计算所有样本的得分矩阵 `scores`。
- 数值稳定性处理时，`np.max(scores, axis=1, keepdims=True)` 计算每个样本的得分最大值，`keepdims=True` 保持维度便于广播。

exp_scores = np.exp(scores) # 计算指数

  probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # 计算概率矩阵 (N, C)

  correct_class_probs = probs[np.arange(num_train), y] # 提取正确类别的概率

  loss = -np.sum(np.log(correct_class_probs)) # 计算总交叉熵损失

  loss = loss / num_train + 0.5 * reg * np.sum(W * W) # 平均损失加正则项

probs是所有样本的Softmax概率矩阵。

probs[np.arange(num_train), y]使用高级索引提取每个样本对应真实标签的概率。

总损失为所有样本正确类别概率的负对数之和。

dscores = probs.copy() # 梯度中间变量

  dscores[np.arange(num_train), y] -= 1 # 正确类别梯度调整

  dW = X.T.dot(dscores) / num_train # 向量化梯度计算

  dW += reg * W # 正则化梯度

- 构造梯度矩阵 `dscores`，其形状与 `probs` 相同。通过索引 `dscores[np.arange(num_train), y] -= 1` 一次性完成所有样本正确类别的梯度调整（*pj*−1），其他类别梯度即为 *pj*。
- 权重梯度 `dW` 通过矩阵乘法 `X.T.dot(dscores)` 一次性计算，再求平均并加上正则化梯度。

梯度计算核心在于：

链式法则，针对损失函数L对权重求导：

$$\frac{\partial L}{\partial \text{scores}} \times \frac{\partial \text{scores}}{\partial W}$$

前者利用softmax的函数求导当取分数正确分数为P-1，不是则为P，后者则是X[i],score=X❌W，求导就是X，故总的来说梯度为

```
for j in range(num_classes):
    dW[:, j] += dscores[j] * X[i]  # 计算每个类别的梯度
```

dW = dW / num_train + reg * W # 平均梯度加正则项梯度

针对梯度去平均梯度，使得学习率大小独立于训练集的大小、

加入L2正则化（求导后）

Q1:

在Softmax分类器的初始阶段，我们期望损失值接近 `-log(0.1)`，这主要基于模型初始预测的概率分布特性。下表总结了导致这一现象的核心原因：

随机初始化的权重	模型参数 W初始时是随机生成的小数值（例如 `W = np.random.randn(3073, 10) * 0.0001`），导致模型在训练初期未学习到有区分度的模式。
均匀的概率分布	由于初始权重很小，所有类别的得分（scores）都很接近。经过Softmax函数处理后，每个类别的预测概率大致相等。对于一个10分类问题，每个类别的概率约为 **0.1**（即 1/10）。
交叉熵损失的计算	交叉熵损失的计算公式为 `Loss = -log(正确类别的预测概率)` 3 7 。当正确类别的预测概率约为0.1时，损失值自然就是 `-log(0.1)`。

Q2:

进行梯度检查为什么SVM梯度检查会失效：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

在于在0处是不可导的，大于0时候导数为1，小于0为0，故在0不可导，但是数值梯度检查时候(f(w + h) - f(w)) / h，但是解析梯度通常会选择一个0或者1作为点梯度、

边界值如何影响频率;

边界值 Δ越大，这种梯度检查失败发生的频率越低。

- **原因**：Δ定义了正确类别得分需要比错误类别得分高出的"安全边际"。
- 当 Δ较小时，$s_j - s_{y_i} + \Delta = 0$这个条件（即 $s_{y_i} = s_j + \Delta$）更容易满足，因为得分不需要相差太多就能到达边界。
- 当 Δ较大时，正确类别的得分需要远高于错误类别得分才能满足边界条件。在模型训练的初期，权重是随机的，得分也是随机的，正确类别得分恰好比错误类别得分高出**很多**（一个大 Δ）的概率要比"高出一点点"（一个小 Δ）的概率**低得多**。

SGD梯度下降：

SGD的优化过程如下表所示，其核心是**通过小批量数据近似整体梯度**，以高效更新参数 [2] [8]：

步骤	代码关键操作	作用与说明
1. 初始化权重	`self.W = 0.001 * np.random.randn(dim, num_classes)`	使用高斯分布随机初始化权重，缩放因子 `0.001`防止初始梯度爆炸
2. 迭代循环	`for it in range(num_iters):`	进行 `num_iters`次优化迭代
3. 小批量采样	`indices = np.random.choice(num_train, batch_size)`	随机选择 `batch_size`个样本索引（有放回抽样，提高效率）
4. 计算损失与梯度	`loss, grad = self.loss(X_batch, y_batch, reg)`	调用子类实现的 `loss`方法（如SVM或Softmax）返回当前批次的损失和梯度
5. 参数更新	`self.W -= learning_rate * grad`	SGD核心：沿梯度反方向更新权重，步长由 `learning_rate`控制

SGD每次迭代仅使用一个小批量（`batch_size`）计算梯度，而非全部训练数据，这显著降低了计算开销，尤其适合大规模数据集

learning_rate：控制更新步长。过大会导致震荡，过小则收敛慢。

batch_size：影响梯度估计的稳定性。较小的批量增加随机性，可能帮助逃离局部最优；较大的批量梯度更稳定，但计算成本高

reg：正则化强度，通过惩罚大权重值防止过拟合（代码中使用L2正则化）

输出：loss_history记录每次迭代的损失值，用于监控收敛情况。

超参数选取：
给定学习率，正则化程度字典记录

```
# 超参数搜索范围定义
learning_rates = [1e-7, 1e-6]
regularization_strengths = [2.5e4, 1e4]

# 结果存储字典
results = {}
best_val = -1  # 初始最佳验证准确率设为-1
best_softmax = None  # 存储最佳模型实例
```

关键变量说明:

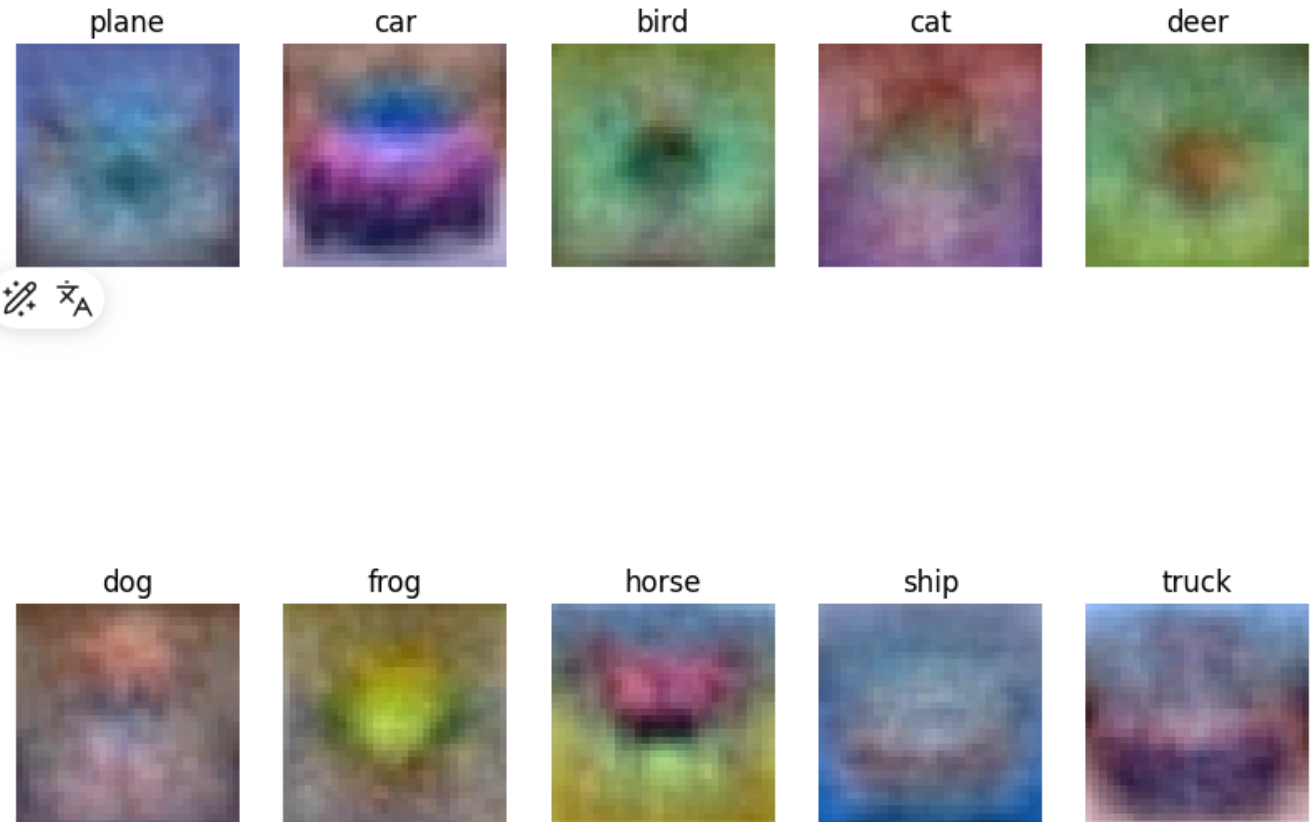
- `results`: 字典，键为`(learning_rate, regularization_strength)`元组，值为`(training_accuracy, validation_accuracy)`元组
- `best_val`: 跟踪当前找到的最佳验证准确率
- `best_softmax`: 存储达到最佳验证准确率的模型实例

超参数网格搜索循环

for lr in learning_rates:

```
    for reg in regularization_strengths:
        # 创建Softmax分类器实例
        softmax_classifier = Softmax()

        # 使用当前超参数训练模型
        softmax_classifier.train(X_train, y_train,
                                 learning_rate=lr,
                                 reg=reg,
                                 num_iters=1000)  # 训练迭代次数

        # 计算训练集准确率
        y_train_pred = softmax_classifier.predict(X_train)
        train_accuracy = np.mean(y_train_pred == y_train)

        # 计算验证集准确率
        y_val_pred = softmax_classifier.predict(X_val)
        val_accuracy = np.mean(y_val_pred == y_val)

        # 存储结果
        results[(lr, reg)] = (train_accuracy, val_accuracy)

        # 更新最佳模型
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax_classifier
```

可视化学习的权重：



Q3：描述你可视化的Softmax分类器权重是什么样子的，并提供一个简短的解释为什么它们看起来是这样的。

特征：模糊的模板形状 颜色主导 低空间分辨率

原因：

- 线性模型的限制：Softmax是线性分类器，只能学习输入特征的线性组合
- 颜色主导模式：在CIFAR-10数据集中，不同类别的颜色分布差异往往比形状差异更容易被线性模型捕捉
- 模板匹配特性：每个类别的权重可以看作是一个"模板"，通过与输入图像进行点积来测量相似度
- 缺乏空间不变性：线性分类器对物体的平移、旋转等变化敏感，导致学习到的特征模糊

Q4:

假设整体训练损失被定义为所有训练样本上每个数据点损失的总和。可以在训练集中添加一个新的数据点，这将改变softmax损失，但保持SVM损失不变。

损失函数	SVM（合页损失）	Softmax（交叉熵损失）
计算方式	max(0, 错误得分-正确得分+1)	-log(正确类别概率)
敏感性	只关心得分相对大小	关心所有类别的概率分布
边界情况	当边际≥1时损失为0	永远追求概率最大化

只要新加数据点波动小于1即可保持不变

3.two_layer_net

神经网路每一层都需要两个核心函数：

forward传播

接收输入数据和本层参数，进行计算并输出结果，同时保存计算过程中产生的、反向传播所必需的数据到 `cache` 中

反向传播 (`backward`)：

接收来自上一层回传的梯度 `dout` 和前向传播保存的 `cache` ，计算本层参数的梯度和传递给前一层的梯度

只要我们定义好了每一层的接口，就可以将它们像搭积木一样组合起来。训练整个网络时，只需要依次调用各层的 `forward` 函数进行前向计算，再反向依次调用各层的 `backward` 函数，梯度就能通过链式法则自动地从输出层传播回输入层。

Affine Layer (全连接层)：

```python
def affine_forward(x, w, b):
    # x: 输入数据，形状为 (N, d_1, ..., d_k)，代表N个样本
    # w: 权重矩阵，形状为 (D, M)，D是输入特征总数，M是输出特征数
    # b: 偏置向量，形状为 (M,)

    # 核心计算步骤:
    N = x.shape[0]
    x_reshape = x.reshape(N, -1)  # 将每个样本展平为向量，形状变为 (N, D)
    out = x_reshape.dot(w) + b    # 计算输出: X * W + b，形状为 (N, M)

    cache = (x, w, b)  # 缓存输入、权重、偏置，反向传播时使用
    return out, cache
```

- **思路解析**：全连接层本质上是进行一次线性变换 `XW + b`。输入数据 `x`可能具有多维结构（如图片的高、宽、通道数），但全连接层需要将其视为一维向量。因此，代码中首先通过 `reshape`操作将每个样本展平。`x.reshape(N, -1)`里的 `-1`会让NumPy自动计算该维度的大小，确保总元素数不变。之后的矩阵乘法和偏置加法就是标准的线性代数运算 [1] [4] [5] 。
- **Cache作用**：缓存输入 `x`、权重 `w`和偏置 `b`是因为在反向传播计算梯度时，公式中会用到这些值。例如，权重的梯度计算需要输入数据，输入的梯度计算需要权重矩阵。

```python
def affine_backward(dout, cache):
    # dout: 上一层传回的梯度，形状为 (N, M)
    # cache: 前向传播时保存的元组 (x, w, b)

    x, w, b = cache  # 解包缓存
    dx, dw, db = None, None, None

    # 核心计算步骤：
    N = x.shape[0]
    x_reshape = x.reshape(N, -1)  # 同样需要展平输入数据

    dx = dout.dot(w.T)              # 计算关于x的梯度
    dx = dx.reshape(x.shape)        # 将dx的形状还原为x的原始形状
    dw = x_reshape.T.dot(dout)      # 计算关于w的梯度
    db = np.sum(dout, axis=0)       # 计算关于b的梯度

    return dx, dw, db
```

- **思路解析**：反向传播的核心是**链式法则**。dout是损失函数对本层输出的梯度。我们需要计算损失函数对本层输入 (dx) 和参数 (dw, db) 的梯度。
  - `dx = dout.dot(w.T)`：根据链式法则，损失对 x的梯度等于损失对输出 out的梯度 (dout) 乘以输出对 x的梯度。由于 `out = x*w + b`，所以 out对 x的梯度就是 w的转置 `w.T`。
  - `dw = x_reshape.T.dot(dout)`：类似地，out对 w的梯度是输入 x的转置。
  - `db = np.sum(dout, axis=0)`：因为偏置 b是直接加到每个输出特征上的，所以梯度是 dout沿着样本维度 (axis=0) 求和 [1] [4] [5] 。

Relu层：

def relu_forward(x):

  # x: 输入数据，可以是任何形状

  out = np.maximum(0, x) # ReLU函数：输出为输入和0之间的最大值

  cache = x # 缓存输入，用于反向传播

  return out, cache

反向（梯度一般都是L对输出求导（dout基本上都是自动微分，输出在对输入求导×起来）

```python
def relu_backward(dout, cache):
    x = cache  # 取出前向传播时缓存的输入x
    dx = dout * (x > 0)  # 计算梯度：当x>0时，梯度为dout；当x<=0时，梯度为0
    return dx
```

- **思路解析**：ReLU函数的导数更加简单。从函数图像上看，在 `x > 0`的区域，其斜率为1；在 `x <= 0`的区域，斜率为0。因此，在反向传播时，对于输入 x中大于0的位置，梯度 dx直接等于上游传来的梯度 dout；对于输入 x中小于等于0的位置，梯度 dx为0。代码 `(x > 0)`会生成一个布尔矩阵，在NumPy中与 dout相乘时，True被视为1，False被视为0，完美实现了这个逻辑 [2] [3] [5] 。

```python
def affine_relu_forward(x, w, b):
    a, fc_cache = affine_forward(x, w, b)  # 全连接层前向传播
    out, relu_cache = relu_forward(a)       # ReLU层前向传播
    cache = (fc_cache, relu_cache)          # 合并缓存
    return out, cache

def affine_relu_backward(dout, cache):
    fc_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)    # ReLU层反向传播
    dx, dw, db = affine_backward(da, fc_cache) # 全连接层反向传播
    return dx, dw, db
```

一个完整的神经网络（如两层网络：Affine - ReLU - Affine - Softmax）的训练过程如下：

1. **前向传播**：按顺序调用各层的 `forward`函数，例如 `affine_relu_forward`和 `affine_forward`，并逐层传递数据和缓存。
2. **计算损失**：在输出层使用Softmax等损失函数计算总损失。
3. **反向传播**：从损失函数开始，反向依次调用各层的 `backward`函数。每一层根据其缓存的中间数据和接收到的上游梯度，计算并传递本地梯度。优化器（如SGD）利用这些梯度更新网络参数 (`w = w - learning_rate * dw`) [1] [4] 。

Q1：激活函数在反向传播又是会获得0的梯度（哪个函数），因为一维情况考虑函数，什么输入会导致该行为

1.sigmoid函数（有严重问题）

1. Sigmoid 函数

- **函数公式**：$f(x) = \frac{1}{1+e^{-x}}$
- **导数公式**：$f'(x) = f(x) \cdot (1 - f(x))$
- **问题分析**：
  Sigmoid函数将输入挤压到(0, 1)之间。从其导数图像可以看出：
  - 当输入x的绝对值很大时（无论是非常大的正数还是负数），其导数$f'(x)$都**接近于0**。
  - 即使输入在中间区域，导数的最大值也只有0.25。

2.ReLU（有问题）

ReLU在正区间的梯度恒为1，这缓解了梯度消失问题。但它有一个致命弱点：

- 对于**所有负的输入**，其导数严格为0。
- **一维情况下的问题输入**：
  任何小于或等于0的输入（例如 $x \leq 0$）都会导致梯度精确地为0。
- **后果**：一旦一个神经元在训练过程中其加权输入（$Wx + b$）稳定地变为负数，它的梯度将永远为0，权重将再也无法更新。这个神经元就"死亡"了，以后永远输出0，不再对学习有任何贡献。

3.Leaky ReLU（基本没问题）

• **函数公式**：$f(x) = max(\alpha x, x)$，其中α是一个很小的常数（如0.01）。

保证了即使为负数，也可以收到一个微小的梯度信号，从而避免神经元的死亡

FC分类器：

架构：`输入层 -> 全连接层 (Affine) -> ReLU激活函数 -> 全连接层 (Affine) -> 输出层`

参数准备：

```
def __init__(self, input_dim=3 * 32 * 32, hidden_dim=100, num_classes=10, weight_scale=1e-3, reg=0.0):
    self.params = {}
    self.reg = reg
    self.params['W1'] = np.random.randn(input_dim, hidden_dim) * weight_scale#乘以weight主要让权重充分小，避免导致梯度消失
    self.params['b1'] = np.zeros(hidden_dim)#偏置一边初始化为0
    self.params['W2'] = np.random.randn(hidden_dim, num_classes) * weight_scale
    self.params['b2'] = np.zeros(num_classes)
```

forward：

```
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']
out1, cache1 = affine_forward(X, W1, b1)   # 全连接层1
out2, cache2 = relu_forward(out1)          # ReLU激活函数
scores, cache3 = affine_forward(out2, W2, b2) # 全连接层2（输出得分）
```

backward：

```
loss, dscores = softmax_loss(scores, y)
loss += 0.5 * self.reg * (np.sum(W1 * W1) + np.sum(W2 * W2))
dout2, dW2, db2 = affine_backward(dscores, cache3) # 输出层Affine反向
dout1 = relu_backward(dout2, cache2)             # ReLU层反向
dX, dW1, db1 = affine_backward(dout1, cache1)     # 隐藏层Affine反向上
dW1 += self.reg * W1
dW2 += self.reg * W2
grads['W1'] = dW1
```

grads['b1'] = db1

grads['W2'] = dW2

grads['b2'] = db2

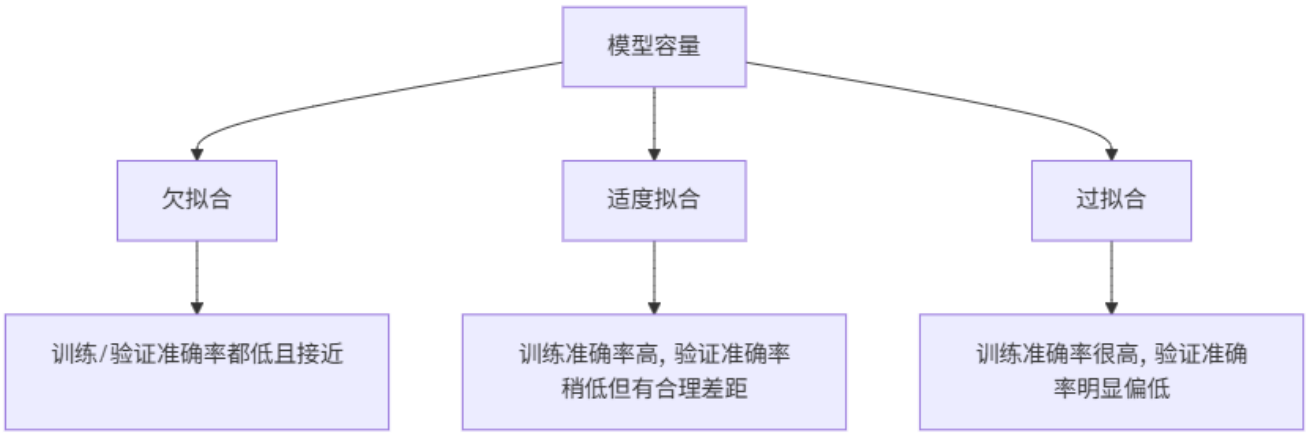
- **Softmax损失**：首先计算数据损失（交叉熵损失）以及损失相对于输出得分 `scores`的梯度 `dscores` [1]。
- **L2正则化**：在损失函数中增加一项 `0.5 * reg * sum(W^2)`，用于惩罚大的权重，鼓励模型更简单，提高泛化能力。
- **反向传播**：这是训练神经网络的核心算法，目的是计算损失函数关于每个参数（`W1`, `b1`, `W2`, `b2`）的梯度。其过程与前向传播相反，应用**链式法则**逐层回溯 [3] [4]。
  - `affine_backward(dout, cache)`：接收上游梯度 `dout`，利用前向传播缓存的 `(x, w, b)`，计算三个梯度：相对于输入 `dx`、权重 `dw`和偏置 `db`。计算公式为 `dx = dout.dot(w.T)`, `dw = x.T.dot(dout)`, `db = np.sum(dout, axis=0)` [2]。
  - `relu_backward(dout, cache)`：ReLU的导数在输入大于0时为1，否则为0。因此，其反向传播只是将上游梯度 `dout`原样传递给那些在前向传播中大于0的输入对应的位置，其他位置置零 [2]。
- **正则化梯度**：L2正则化项对权重 `W`的导数是 `reg * W`，需要将其加到反向传播计算出的数据梯度上，得到最终的总梯度。

超参数调优：

1.损失函数下降：学习率问题（损失值以直线缓慢下降，而不是指数快速下降）

- **理想情况**：良好的学习率会使损失快速下降然后逐渐平缓，形成"曲棍球棒"曲线
- **学习率过低**：参数更新步长太小，每次迭代只能向最优解移动微小距离
- **数学表达**：梯度下降公式 $W = W - \eta \nabla L$ 中，η（学习率）过小导致收敛缓慢

2.训练与验证准确率无差距：模型容量不足



超参数	作用	调优策略	对模型的影响
隐藏层大小	控制模型容量	从小开始逐步增加	容量↑：拟合能力↑但计算成本↑
学习率	控制参数更新步长	对数尺度搜索（如0.1, 0.01, 0.001）	过大：震荡；过小：收敛慢
训练轮数	控制训练时长	观察损失曲线提前停止	过少：欠拟合；过多：过拟合
正则化强度	控制过拟合	与模型容量配合调整	强正则化抑制过拟合但可能欠拟合

Q2：

现在你已经训练了一个神经网络分类器，你可能会发现你的测试精度远低于训练精度。我们可以用什么方法来缩小这个差距呢？选择所有适用的。

在更大的数据集上训练。（增强模型一般性）

添加更多隐藏单位。（增加模型容量可能会得模型更复杂导致过拟合）

增加正则化强度。（惩罚模型参数大小来限制模型复杂度，防止过拟合）

以上都不是。

4.features

在这个练习中，我们将展示我们可以通过训练线性分类器来提高我们的分类性能，而不是在原始像素上，而是在从原始像素计算的特征上。

特征1：HOG-方向梯度直方图

目标：描述图像的局部形状和纹理，同时对光照和颜色不敏感

（忽略颜色信息）

步骤：计算梯度：首先将图像变成灰度图，计算每个像素点的梯度（方向和大小），代表像素点亮度变换最快的方向

创建细胞单元cell与投票

将图像划分成小的相连的矩阵区域（细胞）

在同一个cell中，统计所有像素梯度方向，归入设计好的方向区间内，票数按照像素梯度振幅加权投票（边缘更明显）

块归一化：

相邻多个细胞合成一个更大得快，对块内的细胞单元直方图进行归一化处理（使得特征对光照变换具有鲁棒性，图像整体变亮或变暗，梯度方向分布可能不变，但幅值会变，归一化后可以消除这种影响）

形成最终特征向量：

将所有块的归一化直方图连接起来，形成一个长的一维向量，这就是图像的HOG特征描述符

特征2：颜色直方图（在HSV颜色空间色调通道）

转换颜色从RGB到HSV

计算直方图：仅使用H通道，将色调值的范围（如0-180）划分为若干个“桶”，这个“桶”就叫bin（例如，每20个值一个bin，总共9个bins）。遍历图像中的每个像素，根据其色调值，将其投到对应的bin中。最终，统计每个bin中有多少个像素，就得到了一个颜色分布直方图。这个直方图就是颜色特征向量。

代码部分：

num_color_bins = 25 # Number of bins in the color histogram

颜色直方图的区间为25(默认为10），使得颜色特征更加精细但增加过拟合与计算复杂度

feature_fns = [hog_feature, lambda img: color_histogram_hsv(img, nbin=num_color_bins)]

创建特征函数列表

X_train_feats = extract_features(X_train, feature_fns, verbose=True)

X_val_feats = extract_features(X_val, feature_fns)

X_test_feats = extract_features(X_test, feature_fns)

对三个集合分别提取特征extract_features函数会遍历所有图像，对每张图像依次调用feature_fns中的函数，并将结果拼接成特征向量

归一化处理：

mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)

X_train_feats -= mean_feat

X_val_feats -= mean_feat

X_test_feats -= mean_feat

所有数据集都减去训练集的均值（重要：不能使用测试集的统计信息）

std_feat = np.std(X_train_feats, axis=0, keepdims=True)

X_train_feats /= std_feat

X_val_feats /= std_feat

X_test_feats /= std_feat

所有数据集都除以训练集的标准差

X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])

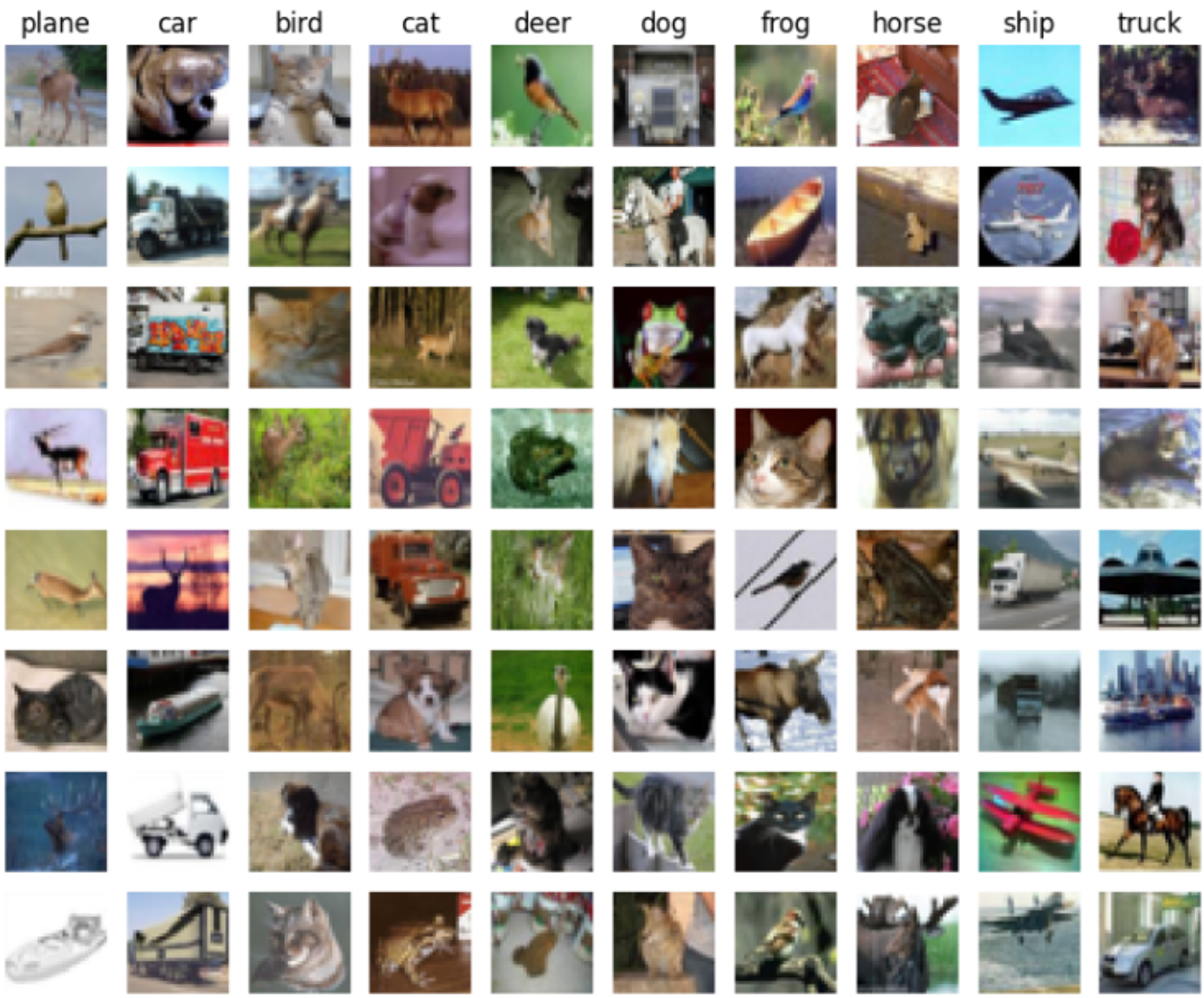
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])

X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])

在最后一列加入1（在w矩阵），保证由bais但更复杂

介绍提取特征函数：

函数名称	功能	输入	输出	特点
hog_feature	提取图像的纹理特征	单张图像	HOG特征向量	对颜色不敏感，捕捉形状和纹理
color_histogram_hsv	提取图像的颜色特征	单张图像	颜色直方图向量	对纹理不敏感，捕捉颜色分布
extract_features	批量特征提取和组合	图像集 + 特征函数列表	特征矩阵	自动化批量处理，特征拼接



Q1：描述错误分类，以及意义

尺寸小：每张图片的尺寸是 32x32 像素，非常低分辨率。这使得分类任务具有挑战性，因为细节信息很少。

主体清晰：尽管图片很小，但每个类别中的物体通常处于图片中心，且姿态、角度、颜色各异。

背景多样：图片背景复杂多变，例如飞机可能在天空或机场，青蛙可能在池塘或草地上。这要求分类器必须学会聚焦于物体本身，而不是依赖固定的背景。

意义：

分析错误：当模型分类错误时，研究者可以制作类似的图表（称为“混淆矩阵”的可视化），来查看模型最常将“猫”误认为“狗”，还是将“船”误认为“卡车”，从而诊断模型的弱点。

5.FullyConnectedNets

定义全连接神经网络函数

{线性变换 - [批归一化] - ReLU激活 - [Dropout]} × (L-1层) - 线性变换 - Softmax

init：

```
def init（self, hidden_dims，# 隐藏层维度列表，如 [100, 100] 表示2个隐藏层，每层100个神经元
  input_dim=3 * 32 * 32，  # 输入维度，默认3072 (CIFAR-10的图像大小)
  num_classes=10,        # 输出类别数，默认10 (CIFAR-10有10个类别)
  dropout_keep_ratio=1，  # Dropout保留率，1表示不使用dropout
  normalization=None,     # 归一化类型：None, "batchnorm", 或 "layernorm"
  reg=0.0,               # L2正则化强度
  weight_scale=1e-2,      # 权重初始化的标准差
  dtype=np.float32,       # 数据类型
  seed=None,             # 随机种子（用于dropout）
):
```

初始化实例子变量：

```
self.normalization = normalization        # 存储归一化类型
self.use_dropout = dropout_keep_ratio != 1  # 如果keep_ratio不等于1，则使用dropout
self.reg = reg                          # 存储正则化强度
self.num_layers = 1 + len(hidden_dims)    # 计算总层数（隐藏层 + 输出层）
self.dtype = dtype                        # 存储数据类型
self. params = {}                         # 初始化参数字典（存储W, b, gamma, beta等）
```

创建层数维度：

```
layer_sizes = [input_dim] + hidden_dims + [num_classes]
```

初始化权重与偏置：

```
for i in range(self.num_layers): # 遍历每一层
  # Weights for layer i+1 (1-based indexing)
  W_key = 'W{}'.format(i+1)   # 权重键名，如 'W1', 'W2', 'W3'
  b_key = 'b{}'. format(i+1)   # 偏置键名，如 'b1', 'b2', 'b3'

  # 初始化权重：从均值为0、标准差为weight_scale的正态分布采样
  # 形状：(当前层大小, 下一层大小)
```

```
    self.params[W_key] = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * weight_scale

    # 初始化偏置：全0向量
    # 形状：(下一层大小,)
    self.params[b_key] = np.zeros(layer_sizes[i+1])
```

初始化批归一化参数：

```
    # If we're using batch normalization and this is not the output layer
    if self.normalization == 'batchnorm' and i < self.num_layers - 1:
        gamma_key = 'gamma{}'. format(i+1)  # 缩放参数键名
        beta_key = 'beta{}'.format(i+1)    # 平移参数键名
        # gamma初始化为1（缩放因子）
        self.params[gamma_key] = np.ones(layer_sizes[i+1])
        # beta初始化为0（平移因子）
        self.params[beta_key] = np.zeros(layer_sizes[i+1])
```

- 只在隐藏层后使用批归一化（不在输出层）
- `i < self.num_layers - 1` 确保输出层不添加批归一化

初始化Dropout参数配置：

```
self.dropout_param = {}
if self.use_dropout:
    self.dropout_param = {
        "mode": "train",              # 模式：训练或测试
        "p": dropout_keep_ratio         # 保留概率（不是丢弃概率！）
    }
    if seed is not None:
        self. dropout_param["seed"] = seed   # 设置随机种子（用于调试）
```

初始化：批归一化处理

```
self.bn_params = []
if self.normalization == "batchnorm":
    # 为每个批归一化层创建参数字典
    # 注意：只有 num_layers - 1 个（输出层不使用BN）
    self.bn_params = [{"mode": "train"} for i in range(self.num_layers - 1)]
```

if self.normalization == "layernorm":

  # Layer normalization 不需要tracking running statistics

  self.bn_params = [{} for i in range(self. num_layers - 1)]

- `bn_params[0]` 对应第1个隐藏层的BN参数
- `bn_params[1]` 对应第2个隐藏层的BN参数
- 测试时会自动切换到 `"mode": "test"`

参数转化：

for k, v in self. params.items():

  self.params[k] = v.astype(dtype) # 将所有参数转换为指定的数据类型

dtype=np.float32（速度快）而64表示精度高

loss方法前向传播：

def loss(self, X, y=None):

  X = X.astype(self. dtype) # 将输入转换为正确的数据类型

  mode = "test" if y is None else "train" # 如果没有标签，则为测试模式

  # Set train/test mode for batchnorm params and dropout param

  if self.use_dropout:

    self.dropout_param["mode"] = mode

  if self.normalization == "batchnorm":

    for bn_param in self.bn_params:

      bn_param["mode"] = mode

前向传播循环：

caches = [] # 存储每层的中间结果，用于反向传播

current_input = X # 当前层的输入（初始为原始输入）

for i in range(self.num_layers):

  W_key = 'W{}'. format(i+1) # 获取权重键名

  b_key = 'b{}'.format(i+1) # 获取偏置键名

  W = self.params[W_key] # 获取权重矩阵

  b = self.params[b_key] # 获取偏置向量

隐藏层前向：

  if i < self.num_layers - 1: # 如果不是最后一层

    out_affine, cache_affine = affine_forward(current_input, W, b)

```
    # Step 2: Optional batch normalization
    if self.normalization == 'batchnorm':
        gamma_key = 'gamma{}'. format(i+1)
        beta_key = 'beta{}'.format(i+1)
        gamma = self.params[gamma_key]  # 缩放参数
        beta = self.params[beta_key]    # 平移参数
        bn_param = self.bn_params[i]    # BN参数字典
        # 批归一化前向传播
        out_bn, cache_bn = batchnorm_forward(out_affine, gamma, beta, bn_param)
        out_to_relu = out_bn
        cache_combined = (cache_affine, cache_bn)  # 组合缓存
    else:
        out_to_relu = out_affine
        cache_combined = cache_affine
```

批次归一化公式如下：

mean = 1/N * Σ(x)

var = 1/N * Σ(x - mean)²

x_normalized = (x - mean) / sqrt(var + eps)

out = gamma * x_normalized + beta

ReLU:

```
代码块
1   out_relu, cache_relu = relu_forward(out_to_relu)
```

```
    # Step 4: Optional dropout
    if self.use_dropout:
        out_dropout, cache_dropout = dropout_forward(out_relu, self.dropout_param)
        current_input = out_dropout  # 下一层的输入
        caches.append((cache_combined, cache_relu, cache_dropout))
    else:
        current_input = out_relu
```

```
        caches.append((cache_combined, cache_relu))
```

输出层前向传播：

```
    else:
        # Last layer: only affine (no relu, bn, or dropout)
        scores, cache_affine = affine_forward(current_input, W, b)
        caches.append(cache_affine)
```

返回测试：

```
if mode == "test":
    return scores  # 测试时只返回分数，不计算loss和梯度
```

损失函数计算：

```
loss, grads = 0.0, {}
loss, dscores = softmax_loss(scores, y)
```

正则化：

```
for i in range(self.num_layers):
    W_key = 'W{}'.format(i+1)
    W = self.params[W_key]
    loss += 0.5 * self.reg * np.sum(W * W)  # 添加 0.5 * λ * ||W||²
```

反向传播：

```
dout = dscores  # 从损失函数的梯度开始反向传播
for i in reversed(range(self.num_layers)):  # 从最后一层往回传播
    W_key = 'W{}'. format(i+1)
    b_key = 'b{}'.format(i+1)
    if i == self.num_layers - 1:
        # Last layer: only affine backward
        cache_affine = caches[i]
        dout, dW, db = affine_backward(dout, cache_affine)
        # dout:  损失对输入的梯度
        # dW: 损失对权重的梯度
        # db: 损失对偏置的梯度
```

隐藏层：

```
    else:
        # Step 1: Backward through dropout (if used)
```

```
        if self.use_dropout:
            cache_combined, cache_relu, cache_dropout = caches[i]
            dout = dropout_backward(dout, cache_dropout)
        else:
            cache_combined, cache_relu = caches[i]
```

ReLU：

```
        # Step 2: Backward through ReLU
        dout = relu_backward(dout, cache_relu)
        # ReLU梯度：dout * (x > 0)
        # Step 3: Backward through batch normalization (if used)
        if self.normalization == 'batchnorm':
            cache_affine, cache_bn = cache_combined
            dout, dgamma, dbeta = batchnorm_backward(dout, cache_bn)
```

BN

```
            # Store bn gradients
            gamma_key = 'gamma{}'.format(i+1)
            beta_key = 'beta{}'.format(i+1)
            grads[gamma_key] = dgamma
            grads[beta_key] = dbeta
        else:
            cache_affine = cache_combined
        # Step 4: Backward through affine
        dout, dW, db = affine_backward(dout, cache_affine)
```

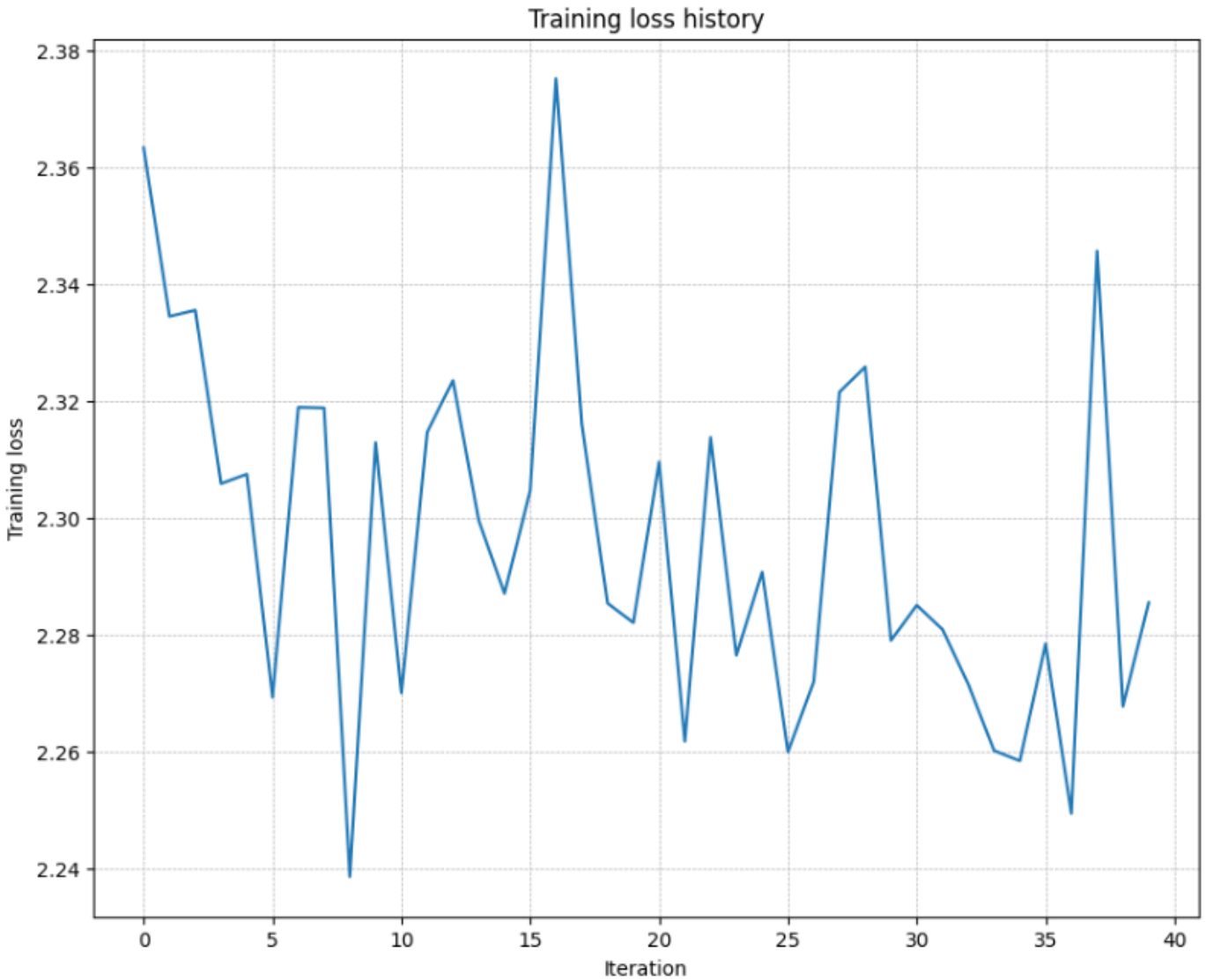
加入正则化梯度：

```
    # Add regularization to weight gradients
    dW += self.reg * self.params[W_key]  # 添加 λ * W

    # Store gradients
    grads[W_key] = dW
    grads[b_key] = db
```

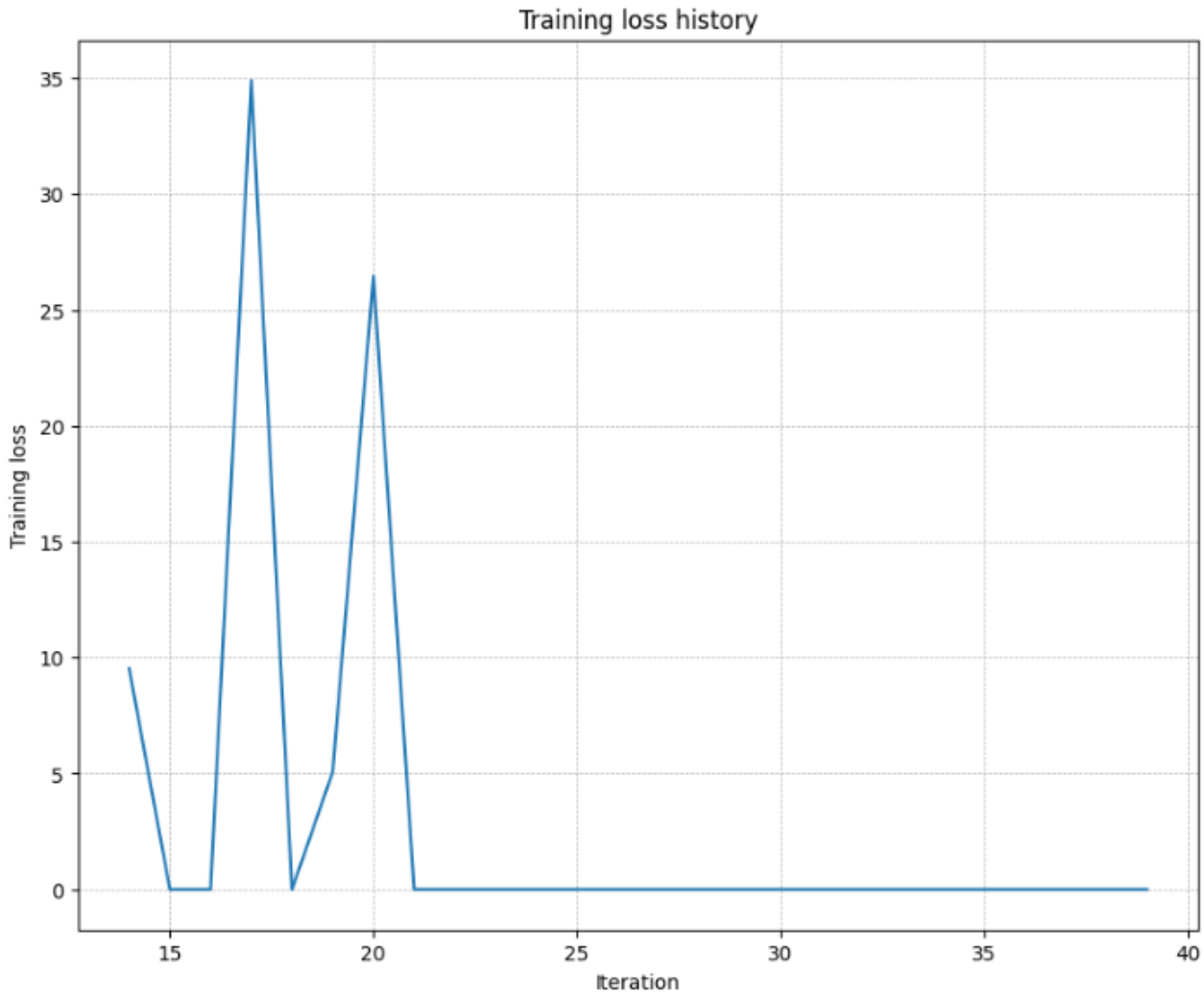
Q1：

你是否注意到训练三层网络和训练五层网络的难度差异？特别是，根据你的经验，哪个网络对初始化尺度更敏感？你认为为什么会这样？

三层



五层



三层网络：

- 训练损失相对稳定，在 2. 24-2.38 之间波动
- 损失曲线较为平滑，逐渐下降
- 没有出现剧烈的震荡或爆炸

五层网络：

- 训练损失出现剧烈波动，从接近0突然飙升到35
- 在第15-20次迭代出现梯度爆炸现象
- 之后损失突然降到0并保持稳定
- 训练过程极不稳定

五层网络对权重初始化尺寸更加敏感：

1.梯度传播更长

三层网络：输入 → 隐藏层1 → 隐藏层2 → 输出（2次梯度传播）

五层网络：输入 → 隐藏层1 → ... → 隐藏层4 → 输出（4次梯度传播）

2.每层初始化权重为W，前向后向传播会缩放W的L幂次，导致梯度爆炸或者消失

四种优化器：

SGD（随机梯度下降）：

$$W_{t+1} = W_t - \eta \times \nabla L(W_t)$$

- W_t：当前权重
- η：学习率
- $\nabla L(W_t)$：损失函数对权重的梯度

```
def sgd(w, dw, config=None):
    """
    最简单的随机梯度下降优化器
    参数:
        w: 当前权重，numpy数组
        dw: 损失对w的梯度，与w形状相同
        config: 配置字典，包含超参数
    """
    # 如果没有传入配置，创建空字典
    if config is None:
        config = {}
    # 设置默认学习率为0.01（如果config中没有指定）
    config.setdefault("learning_rate", 1e-2)
    # 核心更新规则：w = w - learning_rate × dw
    w -= config["learning_rate"] * dw
    # 返回更新后的权重和配置
    return w, config
```

优点：

- 简单易懂
- 计算效率高
- 内存占用小

缺点：

- 收敛速度慢
- 容易陷入局部最优
- 对学习率敏感

SGD+Momentum（动量优化器）：

$$v_t = \mu \times v_{t-1} - \eta \times \nabla L(W_t)$$
$$W_{t+1} = W_t + v_t$$

- v_t：速度（动量）
- μ：动量系数（通常0.9）
- η：学习率

```
def sgd_momentum(w, dw, config=None):
    """
    带动量的随机梯度下降
    config格式:
    - learning_rate: 学习率（标量）
    - momentum: 动量系数，0到1之间（0表示退化为普通SGD）
    - velocity: 速度向量，与w和dw形状相同，存储梯度的移动平均
    """
    # 初始化配置字典
    if config is None:
        config = {}
    # 设置默认学习率为0.01
    config.setdefault("learning_rate", 1e-2)
    # 设置默认动量系数为0.9
    config.setdefault("momentum", 0.9)
    # 获取速度向量（如果第一次调用，返回None）
    v = config.get("velocity", None)
    # 如果速度向量不存在，初始化为与w形状相同的零向量
    if v is None:
        v = np.zeros_like(w) # 形状与w相同，全为0
```

```
    # 核心更新公式：
    # v = momentum × v - learning_rate × dw
    # 旧速度的加权 + 当前梯度的贡献
    v = config["momentum"] * v - config["learning_rate"] * dw
    # 更新权重：w = w + v
    next_w = w + v
    # 保存速度向量到config中，供下次迭代使用
    config["velocity"] = v
    # 返回新权重和更新后的配置
    return next_w, config
```

优点：

- 加速收敛（特别是在一致方向上）
- 减少震荡（在峡谷地形）
- 能逃离浅层局部最优

缺点：

- 仍需手动调整学习率
- 在最优点附近可能"冲过头"

3.RMSprop（自适应学习率方法）

$$\text{cache\_t} = \beta \times \text{cache\_\{t-1\}} + (1-\beta) \times (\nabla L)^2$$
$$W\_\{t+1\} = W\_t - \eta \times \nabla L \,/\, (\sqrt{\text{cache\_t}} + \varepsilon)$$

- `cache`：梯度平方的指数移动平均
- `β`：衰减率（通常0.99）
- `ε`：防止除零的小常数

```
def rmsprop(w, dw, config=None):
    """
    RMSprop优化器：自适应学习率方法
    """
    # 初始化配置
    if config is None:
```

```
    config = {}
  # 设置默认超参数
  config.setdefault("learning_rate", 1e-2)     # 学习率
  config.setdefault("decay_rate", 0.99)        # 衰减率β
  config.setdefault("epsilon", 1e-8)           # 防止除零
  config.setdefault("cache", np.zeros_like(w))  # 梯度平方的累积
  # 更新cache：梯度平方的指数移动平均
  # cache = decay_rate × cache_old + (1 - decay_rate) × dw²
  config["cache"] = (config["decay_rate"] * config["cache"] +
                     (1 - config["decay_rate"]) * (dw ** 2))
  # 自适应学习率更新权重
  # next_w = w - lr × dw / (√cache + ε)
  # 分母√cache使得梯度大的方向步长变小，梯度小的方向步长变大
  next_w = w - (config["learning_rate"] * dw /
                (np.sqrt(config["cache"]) + config["epsilon"]))
  return next_w, config
```

优点：

- 自适应学习率
- 对稀疏梯度表现好
- 减少手动调参

缺点：

- 学习率可能过早衰减
- 在某些情况下不收敛

4.Adam

$$
\begin{aligned}
m_t &= \beta_1 \times m_{t-1} + (1-\beta_1) \times \nabla L && \text{\# 一阶矩估计（动量）} \\
v_t &= \beta_2 \times v_{t-1} + (1-\beta_2) \times (\nabla L)^2 && \text{\# 二阶矩估计（RMSprop）} \\
\\
\hat{m}_t &= m_t / (1 - \beta_1^t) && \text{\# 偏差修正} \\
\hat{v}_t &= v_t / (1 - \beta_2^t) \\
\\
W_{t+1} &= W_t - \eta \times \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)
\end{aligned}
$$

- m_t : 梯度的指数移动平均（一阶矩）
- v_t : 梯度平方的指数移动平均（二阶矩）
- β_1 : 一阶矩衰减率（通常0.9）
- β_2 : 二阶矩衰减率（通常0.999）
- 偏差修正防止初期m和v偏向0

```
def adam(w, dw, config=None):
    """
    Adam优化器：自适应矩估计

    结合了Momentum和RMSprop的优点
    """
    # 初始化配置
    if config is None:
        config = {}

    # 设置默认超参数
    config.setdefault("learning_rate", 1e-3)     # 学习率（Adam默认更小）
    config.setdefault("beta1", 0.9)              # 一阶矩衰减率
    config.setdefault("beta2", 0.999)            # 二阶矩衰减率
    config.setdefault("epsilon", 1e-8)           # 数值稳定性
    config.setdefault("m", np.zeros_like(w))     # 一阶矩（梯度均值）
    config.setdefault("v", np.zeros_like(w))     # 二阶矩（梯度方差）
    config.setdefault("t", 0)                    # 迭代计数器
    # 增加迭代计数
```

```
    config["t"] += 1
    # 更新一阶矩（类似Momentum）
    # m = β₁ × m + (1-β₁) × dw
    config["m"] = (config["beta1"] * config["m"] +
                   (1 - config["beta1"]) * dw)
    # 更新二阶矩（类似RMSprop）
    # v = β₂ × v + (1-β₂) × dw²
    config["v"] = (config["beta2"] * config["v"] +
                   (1 - config["beta2"]) * (dw ** 2))
    # 偏差修正（Bias Correction）
    # 因为m和v初始化为0，在训练初期会偏向0
    # 修正公式：m̂ = m / (1 - β₁ᵗ)
    m_hat = config["m"] / (1 - (config["beta1"] ** config["t"]))
    v_hat = config["v"] / (1 - (config["beta2"] ** config["t"]))
    # 更新权
    # next_w = w - lr × m̂/(√v̂+ε)
    next_w = w - (config["learning_rate"] * m_hat /
                  (np. sqrt(v_hat) + config["epsilon"]))
    return next_w, config
```

优点：

- 结合了Momentum和RMSprop的优点
- 自适应学习率
- 偏差修正确保训练初期稳定
- 超参数鲁棒性好（默认值通常有效）
- 目前最流行的优化器

❌缺点：

- 计算开销稍大（需维护两个动量）
- 内存占用是SGD的3倍
- 在某些情况下可能不如SGD+Momentum泛化好

优化器	学习率	内存	收敛速度	适用场景
SGD	固定	1×	慢	简单问题、需要好泛化性
Momentum	固定	2×	中等	有明显方向性的问题
RMSprop	自适应	2×	快	RNN、稀疏梯度
Adam	自适应	3×	最快	**大多数深度学习任务（推荐）**