模式识别与机器学习
第二次作业报告

神经网络实现

前馈过程

在神经网络的前馈过程中，网络首先计算第一层的加权输入和激活，使用ReLU激活函数引入非线性。之后，网络计算第二层的加权输入，稳定化后经过SoftMax激活函数得到概率分布。稳定化即所有分数分量都减去分数的最大值，可以防止指数项在计算时溢出。数学表示如下：

1. 隐藏层：$\tilde{H} = \max(0, \tilde{X}\tilde{W}_1) \cup \vec{1}$
2. 输出层：$p_i = \frac{e^{s_i - \max S}}{\sum_j e^{s_j - \max S}}, S = \tilde{H}\tilde{W}_2$

在 `TwoLayerNet` 的 `loss` 方法中添加代码：

```
W1 = np.concatenate((W1, b1.reshape(1, -1)), axis=0)
W2 = np.concatenate((W2, b2.reshape(1, -1)), axis=0)
X = np.concatenate((X, np.ones((N, 1))), axis=1)
h = np.concatenate((np.maximum(0, X @ W1),
                                        np.ones((N, 1))), axis=1)
scores = h @ W2
scores -= np.max(scores, axis=1, keepdims=True)
scores = np.exp(scores)
scores = scores / np.sum(scores, axis=1, keepdims=True)
```

利用真实标签，从概率分布中提取对应的概率，计算其交叉熵损失；为了减轻过拟合，加入权重的L2正则化项：

$$J = -\frac{1}{N}\sum_i \lg p_i + \lambda(W_1^T W_1 + W_2^T W_2)$$

代码为：

```
logprobs = -np.log(scores[np.arange(N), y])
data_loss = np.sum(logprobs) / N
reg_loss = reg * (np.sum(np.delete(W1, -1, axis=0) ** 2) \
           + np.sum(np.delete(W2, -1, axis=0) ** 2))
loss = data_loss + reg_loss
```

训练过程

每轮训练中从加载的数据集中随机选取一部分作为该batch的训练样本：

```
mask = np.random.choice(num_train, batch_size)
X_batch = X[mask]
y_batch = y[mask]
```

前向传播并计算各层梯度后，对每个神经元按学习率更新参数：

$$W^{(t+1)} = W^{(t)} - \eta\nabla J$$

```
for param in self.params:
        self.params[param] -= learning_rate * grads[param]
```

分类过程

预测过程中不用关心概率分布，分数最高的类别即为预测类别：

```
z1 = np.dot(X, self.params['W1']) + self.params['b1']
a1 = np.maximum(0, z1)
scores = np.dot(a1, self.params['W2']) + self.params['b2']
y_pred = np.argmax(scores, axis=1)
```

Toy Example 上训练

检查代码修改正确后，在Toy Example上训练得到loss曲线和准确率如下：

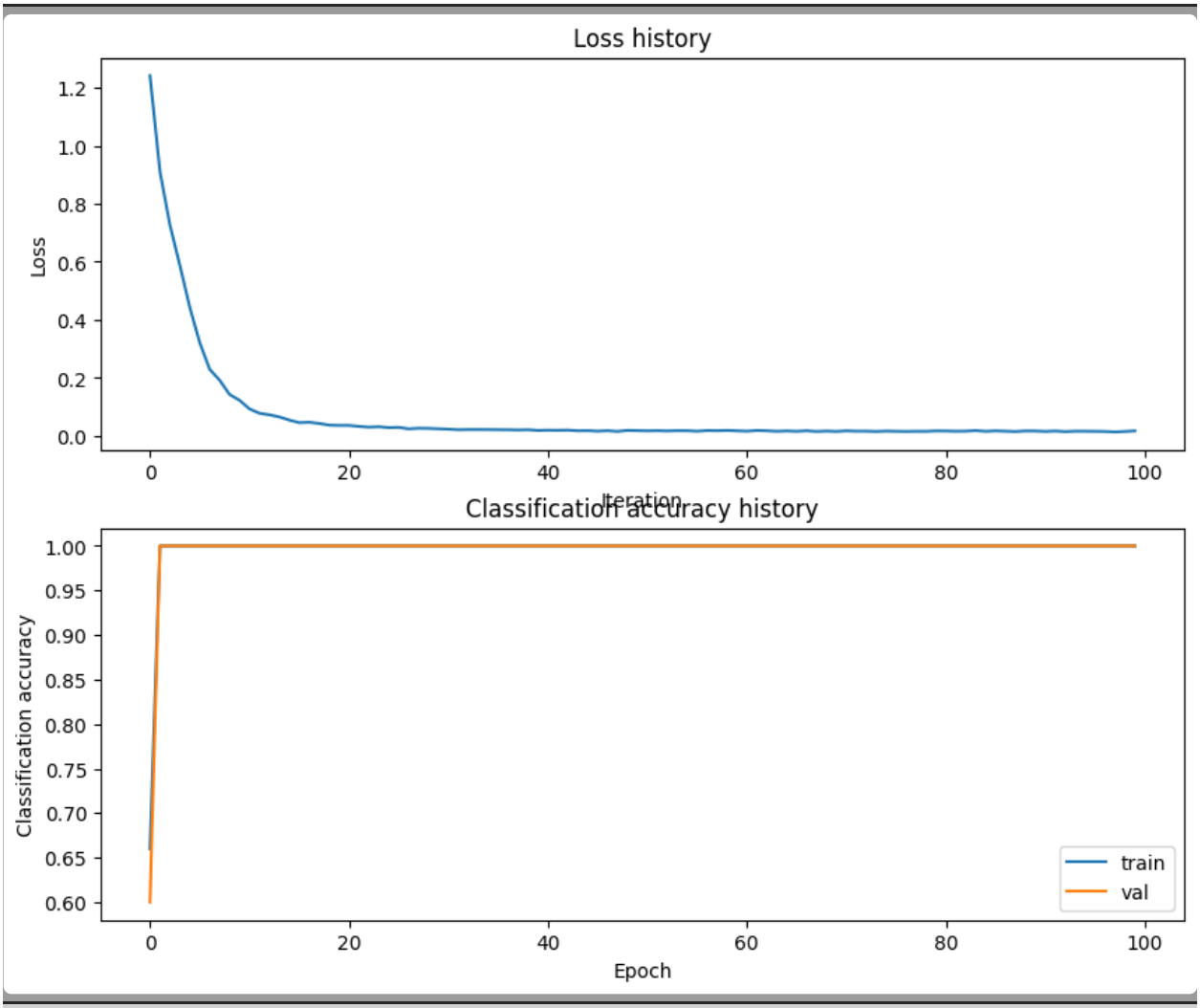

图1 在Toy Example上的loss曲线、训练和验证准确率

loss正确下降，说明神经网络设计可行；训练全程都是下降趋势，说明训练时长没有过长；在100次迭代后几乎收敛到平稳，说明训练时长没有过短。 训练集和验证集准确率都高到了100%，说明既没有欠拟合也没有过拟合。

CIFAR-10 数据集上训练

默认超参

按照脚本默认参数训练：

```
hidden_size = 50
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=1.0,
            reg=0.25, verbose=True)
```

训练结果为：

```
Validation accuracy: 0.3
Test accuracy: 0.306
```

结果不是很好，查看训练loss曲线、训练和验证准确率、神经网络权重如图2, 3：

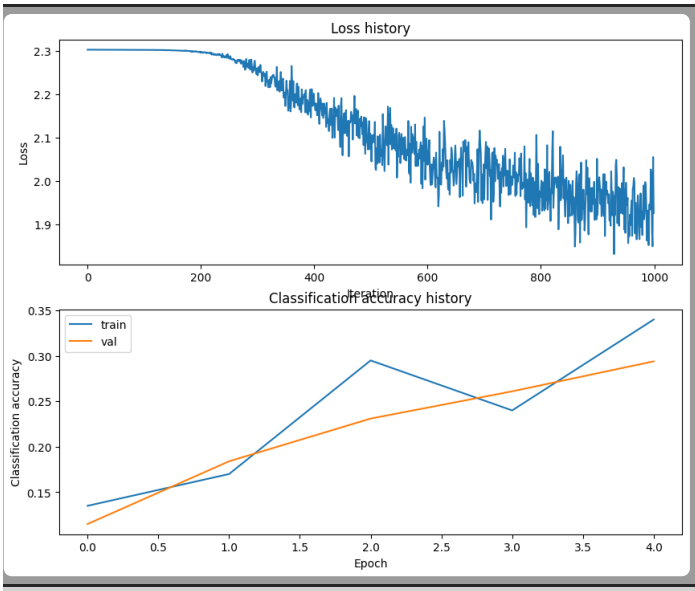

图2 默认超参训练loss曲线、训练和验证准确率

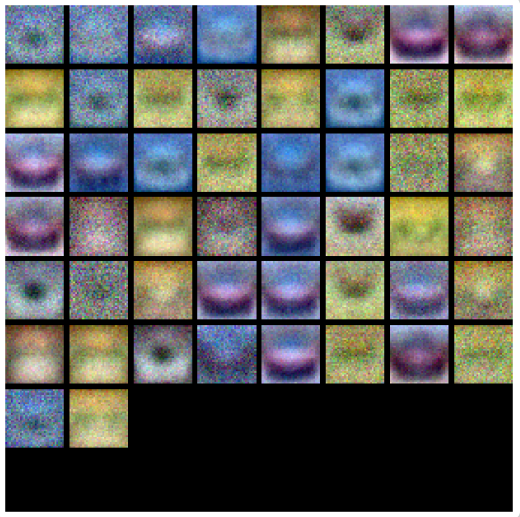

图3 默认超参训练后神经网络权重

loss曲线下降缓慢且没有收敛，说明学习率过低或者训练时间太短。训练和验证准确率一致但都低，说明欠拟合。神经网络权重可视化后没有规律清晰的形状，说明神经网络没有学习到样本中良好的模式，训练不足。

调整超参

选取一些可能的超参配置：

```
learning_rates = (3e-3, 1e-3, 1e-4)
hidden_sizes = (50, 75, 100)
iter_nums= (1000, 2000, 3000, 4000)
lr_decays = (0.5, 0.7, 0.8, 1.0)
regs = (0.003, 0.3, 3, 30.0)
```

进行枚举尝试，代码见 `main.ipynb`，全部结果、最佳配置、最佳网络分别保存在 `results.json`，`best_config.json`，`best_net.npy` 中，得到的调整后配置为：


```
{"iter_num": 3000, "hidden_size": 75, "learning_rate": 0.003, "lr_decay":
0.8, "reg": 0.003, "val_acc": 0.547}
```


结果达到了：

```
Validation accuracy: 0.529
Test accuracy: 0.528
```

查看训练loss曲线、训练和验证准确率、神经网络权重如图4, 5：

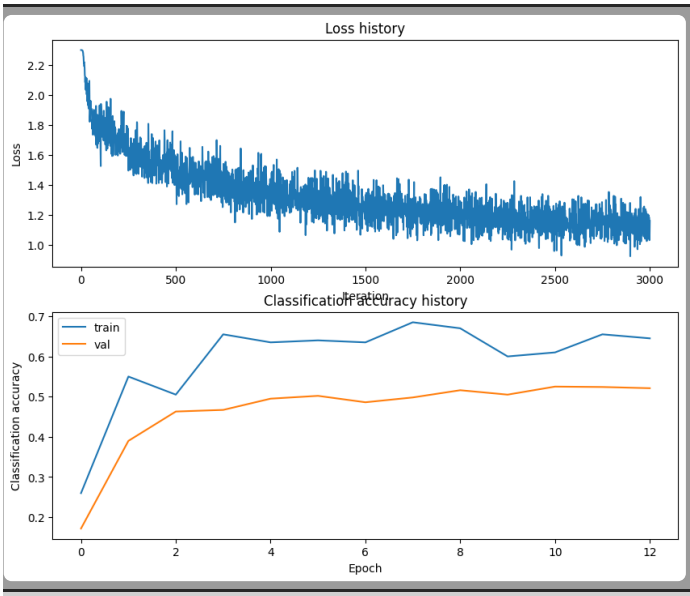

图4 调整超参训练loss曲线、训练和验证准确率

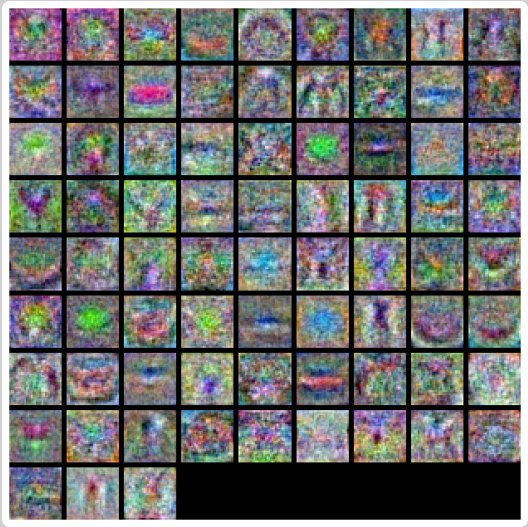

图5 调整超参训练后神经网络权重

loss曲线一直呈下降趋势，且到3000轮基本收敛，说明训练时间合适。训练和验证准确率达到较高水平，训练准确率比验证准确率稍高，说明网络没有欠拟合，存在一定的过拟合现象。神经网络权重可视化后有明显多样的规律形状，说明神经网络学习到很多样本中的典型模式，对任务学习效果很好。

超参分析

根据得到的 `results.json` 中的数据，选取特征进行绘图：

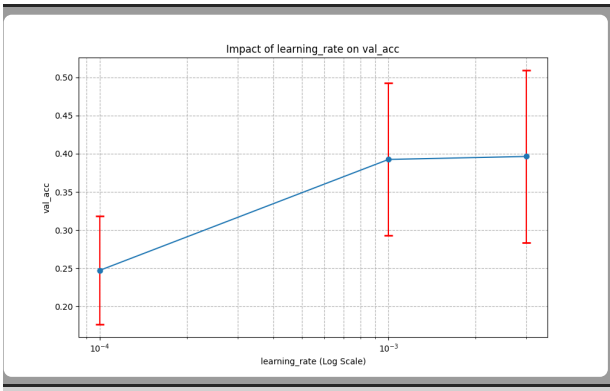

图6 学习率与验证准确率的关系

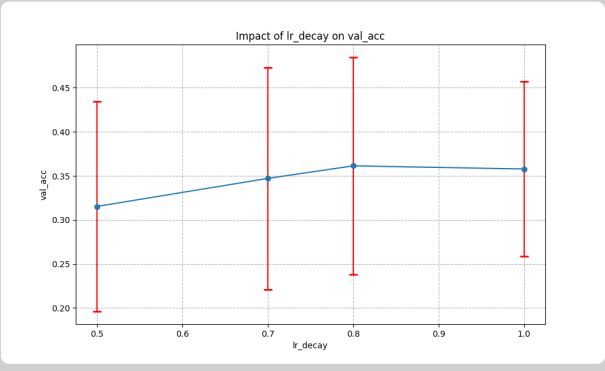

图7 学习率衰减率与验证准确率的关系

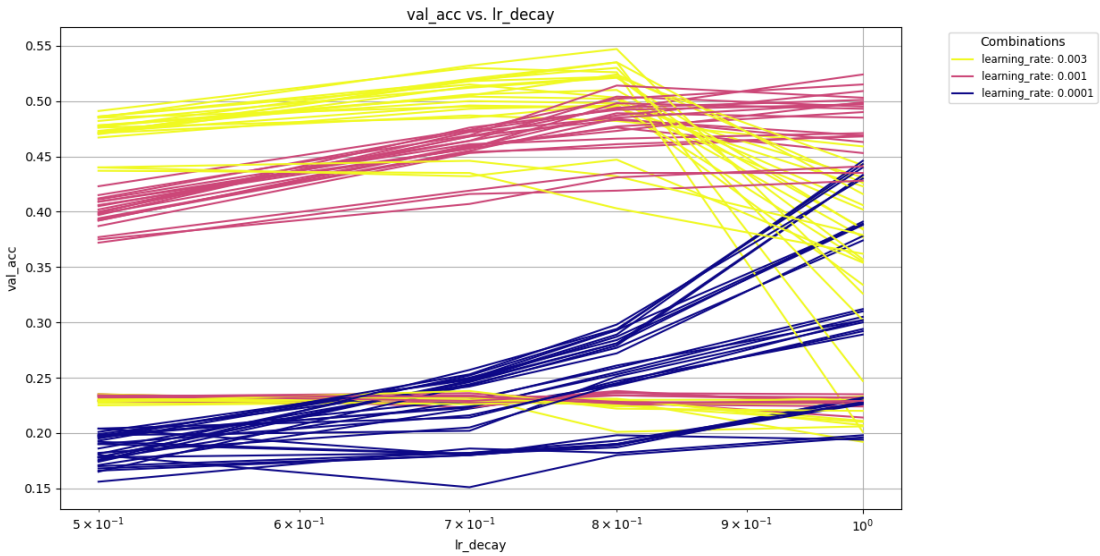

图8 学习率、学习率衰减率与验证准确率的关系

decay较小时，学习率从0.0001上升到0.003，收敛到极小值速度加快，过拟合风险减小，验证准确率也上升。 当学习率在0.0001到0.001之间，随decay增大，收敛到极小值速度加快，准确率率上升；当学习率在0.003，decay在1.0时准确率出现下降，说明此时训练中后期decay过大会导致学习率过大，在极小值振荡而不收敛。

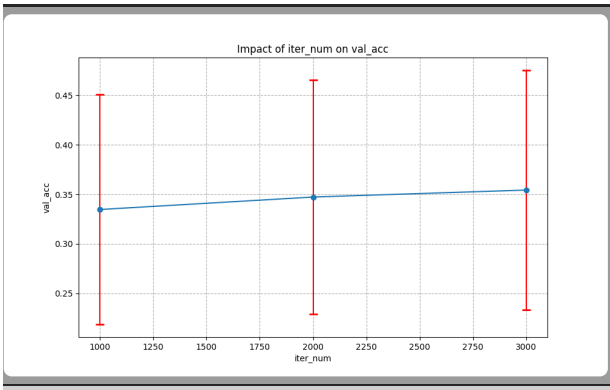

图9 训练迭代次数与验证准确率的关系

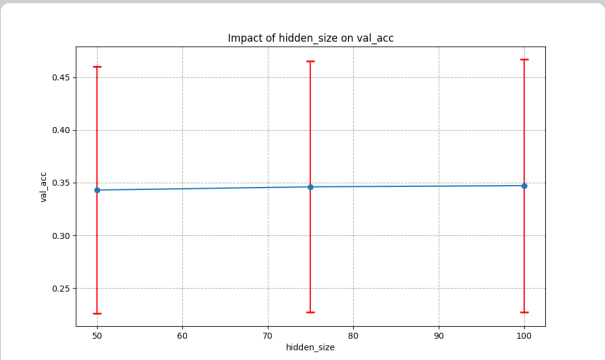

图10 隐藏层尺寸与验证准确率的关系

总体上看，迭代次数越多，验证准确率越高，这是在正则化系数束缚下没有严重过拟合导致的。隐藏层尺寸越大，验证准确率也有提升，但十分小，并且随着尺寸越大，提升效果下降，说明当前尺寸已经足够学习这个复杂度的样本，更高容易过拟合。

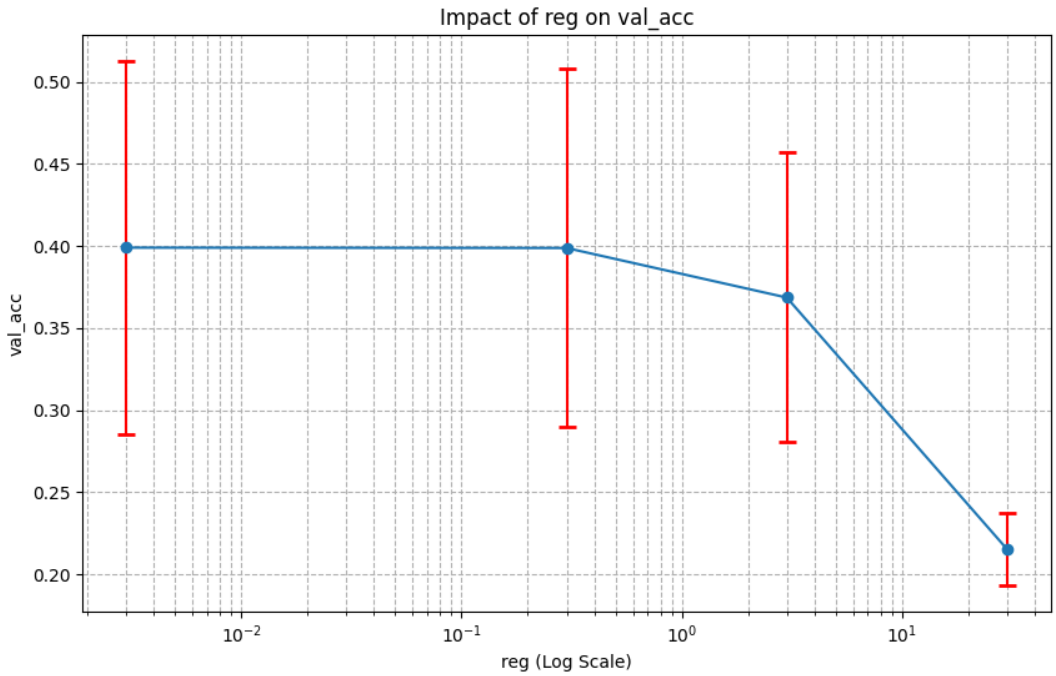

图11 正则化系数与验证准确率的关系

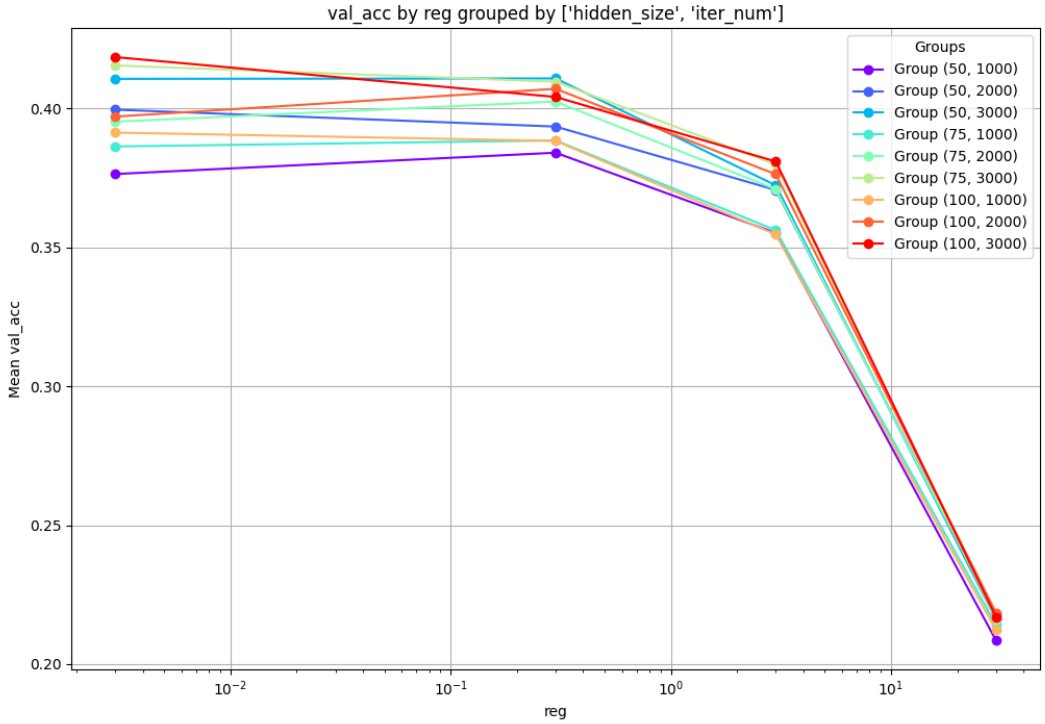

图12 正则化系数和隐藏层尺寸与验证准确率的关系

在0.003到30的范围内，正则化系数增大，准确率总总体有下降的趋势，因为过大的正则化系数会限制网络的复杂程度；而迭代次数多、宽度大的网络更早开始下降，可能是因为正则化项并没有对隐藏层大小归一化；至于正则化系数小的时候准确率还未下降，可能是测试网络的复杂程度还不足以有严重的过拟合现象。

其他激活函数

Sigmoid函数

为了和ReLU版本隔离， `git checkout -b sigmoid` 新建分支。定义sigmoid函数：

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

将 `loss` 前向传播和 `predict` 中的ReLU替换为sigmoid；然后由于:

$$\sigma'(a) = \sigma(a)(1 - \sigma(a))$$

只需修改 `loss` 反向传播方法中：

```
# dh = dh*(np.delete(h, -1, axis=1) != 0)
dh *= h[:, :-1] * (1 - h[:, :-1])
```

测试后发现sigmoid作为激活函数的效果不及ReLu.

Tanh函数

为了和ReLU版本隔离，从main `git checkout -b tanh` 新建分支。将 `loss` 前向传播和 `predict` 中的ReLU替换为Tanh；然后由于：

$$\tanh'(a) = 1 - \tanh^2(a)$$

只需修改 `loss` 反向传播方法中：

```
# dh = dh*(np.delete(h, -1, axis=1) != 0)
dh *= (1 - h[:, :-1] ** 2)
```

测试后发现Tanh的效果也不及ReLU

Dropout与PCA

进行更多尝试， `git branch -b more` 创建新分支。

Dropout

以一定的概率使得隐藏神经元失活，添加网络属性 `dropout` ，在 `loss` 方法中应用：

```
if self.dropout > 0:
        mask = (np.random.rand(*h.shape) < (1 - self.dropout))
        h *= mask / (1 - self.dropout)
```

反向传播时，只更新活动的神经元：

```
if self.dropout > 0:
        dh *= mask / (1 - self.dropout)
```

PCA

利用 sklearn.decomposition 提取主成分并还原为 32*32*3，在 get_CIFAR10_data 函数中加入：

```
from sklearn.decomposition import PCA
pca = PCA(n_components=num_components)
pca.fit(X_train)  # Fit PCA only on training data
X_train_pca = pca.transform(X_train)
X_val_pca = pca.transform(X_val)
X_test_pca = pca.transform(X_test)
X_train = pca.inverse_transform(X_train_pca).reshape(-1, 3072)
X_val = pca.inverse_transform(X_val_pca).reshape(-1, 3072)
X_test = pca.inverse_transform(X_test_pca).reshape(-1, 3072)
```

结果分析

将正则化参数设置为0，增大迭代次数和隐藏层宽度：

```
num_components=300
hidden_size = 150
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=5000, batch_size=200,
            learning_rate=3e-3, learning_rate_decay=0.8,
            reg=0.0, tqdm_verbose=True)
```

得到的结果为：

```
Validation accuracy: 0.522
Test accuracy: 0.525
```

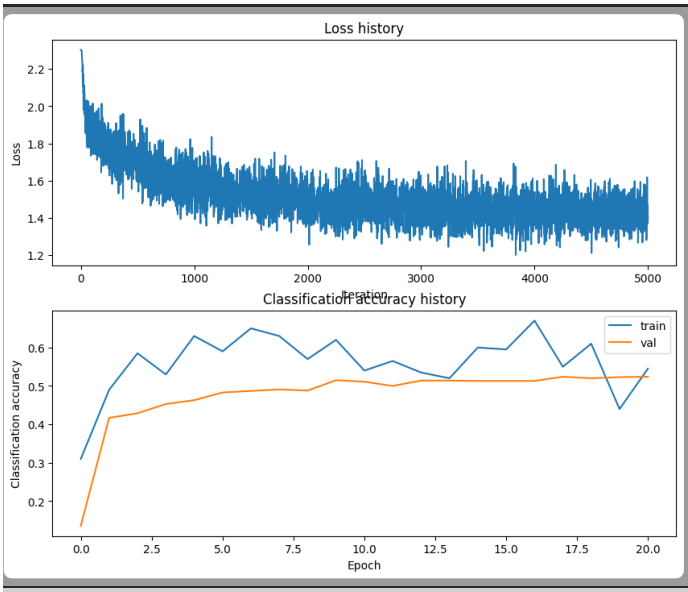

图13 带有PCA与dropout的训练loss曲线、训练和验证准确率

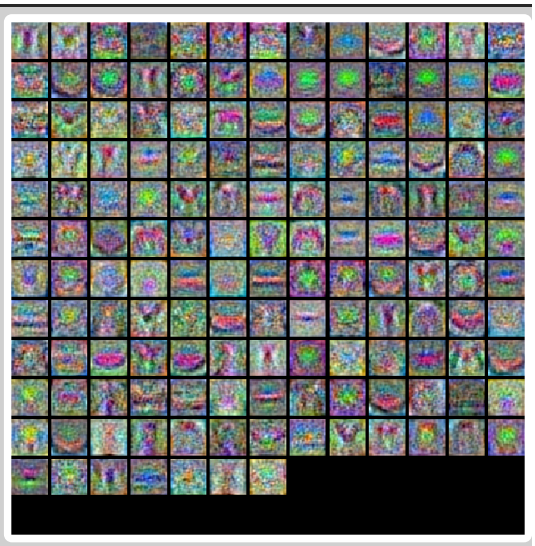

图14 带有PCA与dropout的训练后神经网络权重

可以发现即使是去除正则化项，训练时过拟合风险也显著下降，不过准确率和loss曲线因为dropout的存在会有明显的波动。