人工神经网络第一次大作业

ConvNet

【学号】22336259　　【姓名】谢宇桐　　【专业】计算机科学与技术

【实验环境】

已创建python 3.7的虚拟环境，将requirements.txt文件中的库都安装完成，并安装了编译CPython文件需要的库。已将前置所有实验按要求完成。

【实验内容】

根据 `ConvolutionalNetwork.ipynb` 中 `Train your best model` 的要求，利用 `annp` 文件夹中的模块实现用于分类 CIFAR-10 数据集的卷积神经网络。需要注意的是，只能用 `annp` 文件夹中的模块实现你的模型，不允许使用额外的深度学习框架。请在 `annp/classifiers/cnn.py` 中实现模型，在 Jupyter Notebook 对应位置实现你的训练过程、实验结果以及可视化分析。请各位同学仔细阅读 `annp` 文件夹中每个模块的用法。

【模型架构】

采用了两层卷积神经网络结构：

1. 第一卷积层：

   卷积：32个7×7滤波器；激活函数：ReLU；池化：2×2最大池化

2. 第二卷积层：

   卷积：64个7×7滤波器；激活函数：ReLU；池化：2×2最大池化

3. 全连接层：

   隐藏层：500个神经元 + ReLU

   输出层：10个神经元（对应CIFAR-10的10个类别）

【实验过程与调参】

一、初始实验

刚开始我是用batchnorm+relu的卷积层，以及在全连接层加入dropout层，但无论怎么调整参数，准确率都不到60%。这也导致我浪费了很多时间。后面不用batchnorm和dropout，改为只用relu后准确率成功到达60%。

初始实验结果：

```
# 在测试集上评估
test_acc = solver.check_accuracy(data['X_test'], data['y_test'])
```

```
print('Test accuracy: %f' % test_acc)
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
(Iteration 1 / 9800) loss: 2.405771
(Epoch 0 / 20) train acc: 0.159000; val_acc: 0.151000
# ...
(Epoch 19 / 20) train acc: 0.548000; val_acc: 0.525000
(Iteration 9351 / 9800) loss: 0.875325
(Iteration 9401 / 9800) loss: 0.567016
(Iteration 9451 / 9800) loss: 0.578852
(Iteration 9501 / 9800) loss: 0.562205
(Iteration 9551 / 9800) loss: 0.674606
(Iteration 9601 / 9800) loss: 0.622219
(Iteration 9651 / 9800) loss: 0.519482
(Iteration 9701 / 9800) loss: 0.657519
(Iteration 9751 / 9800) loss: 0.563937
(Epoch 20 / 20) train acc: 0.589000; val_acc: 0.560000
```

二、参数调整与分析

我对参数调整作了以下实验：

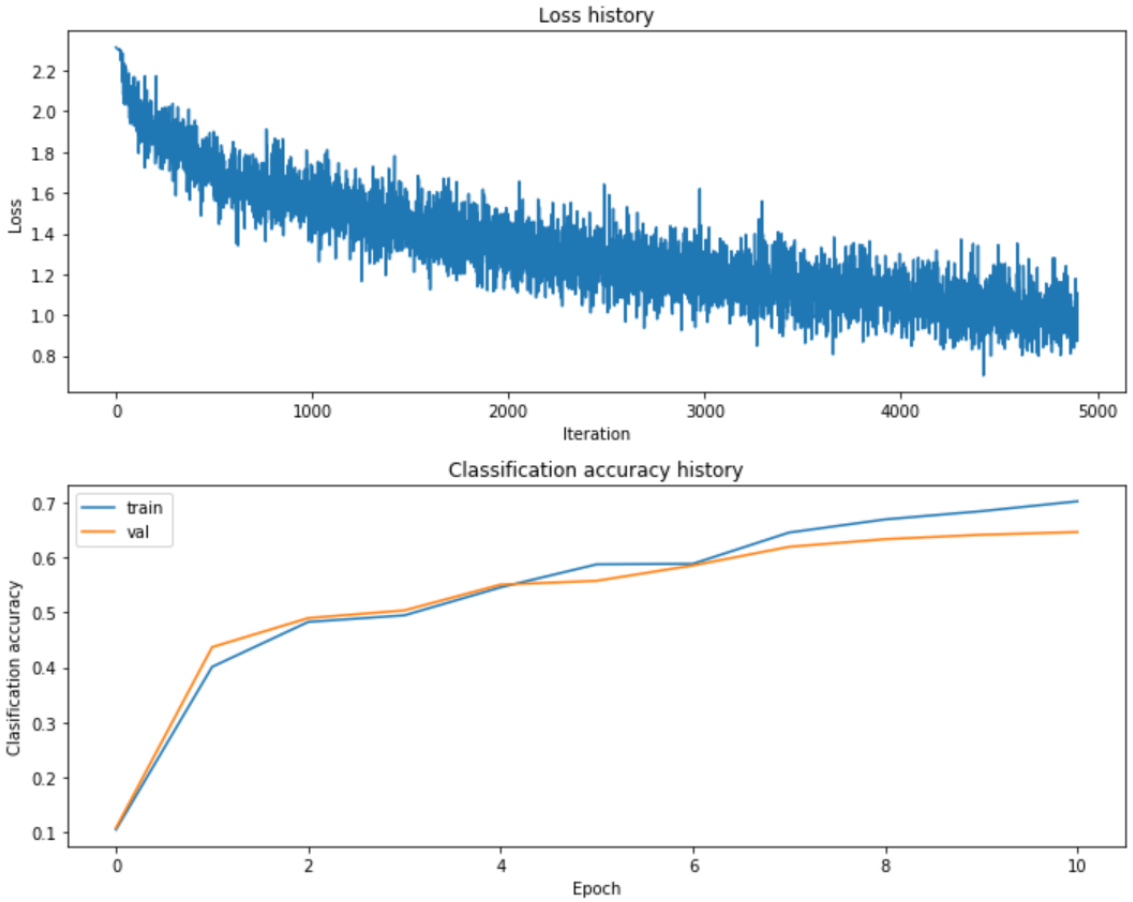
滤波器尺寸（Filter Size）

尺寸	训练准确率	验证准确率	分析
3*3	70.3%	64.7%	基础模型，准确率较低
5*5	83.3%	71.6%	性能提高，符合要求
7*7	87.8%	69.4%	

【实验结果与分析】

3*3：

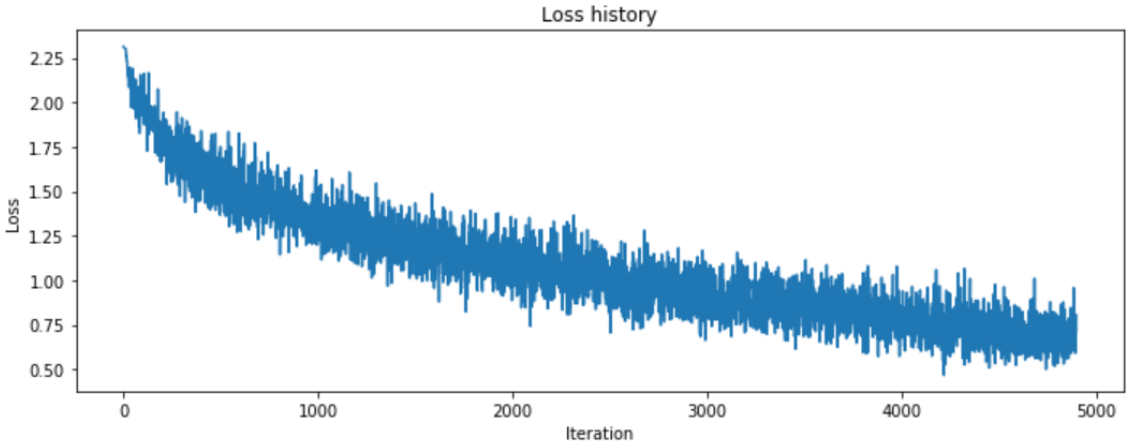
```
(Iteration 1 / 4900) loss: 2.312939
(Epoch 0 / 10) train acc: 0.104000; val_acc: 0.107000
# ...
(Iteration 4801 / 4900) loss: 1.104757
(Iteration 4851 / 4900) loss: 0.993034
(Epoch 10 / 10) train acc: 0.703000; val_acc: 0.647000
```

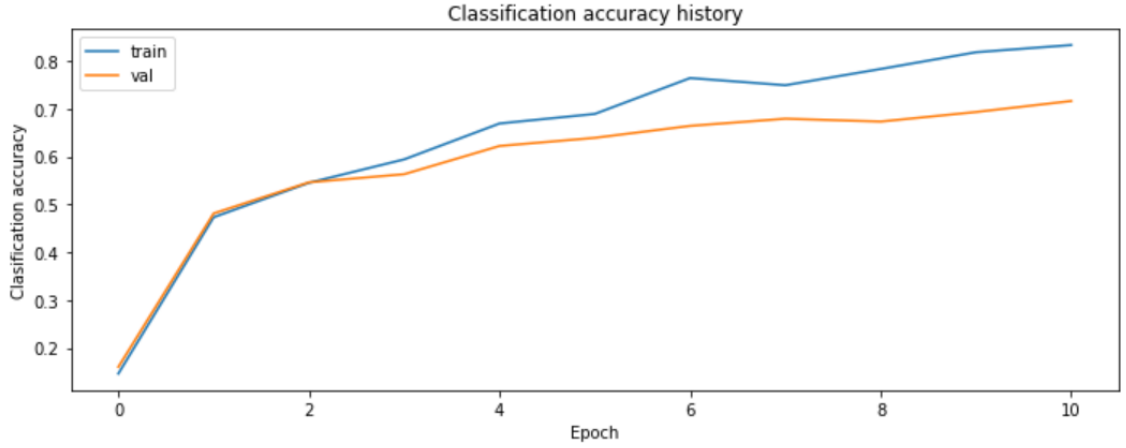


由上图我们可以看到其准确率较低，可能是因为较小的滤波器捕捉到的特征不够丰富，导致模型的表达能力有限。

5*5：

```
(Iteration 1 / 4900) loss: 2.313112
(Epoch 0 / 10) train acc: 0.146000; val_acc: 0.160000
(Iteration 51 / 4900) loss: 1.967453
# ...
(Iteration 4801 / 4900) loss: 0.683393
(Iteration 4851 / 4900) loss: 0.694077
(Epoch 10 / 10) train acc: 0.833000; val_acc: 0.716000
```

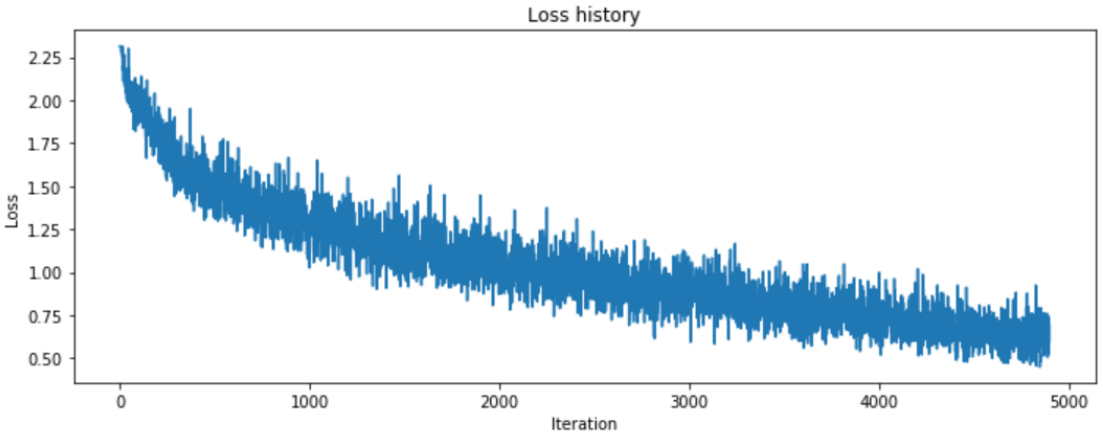


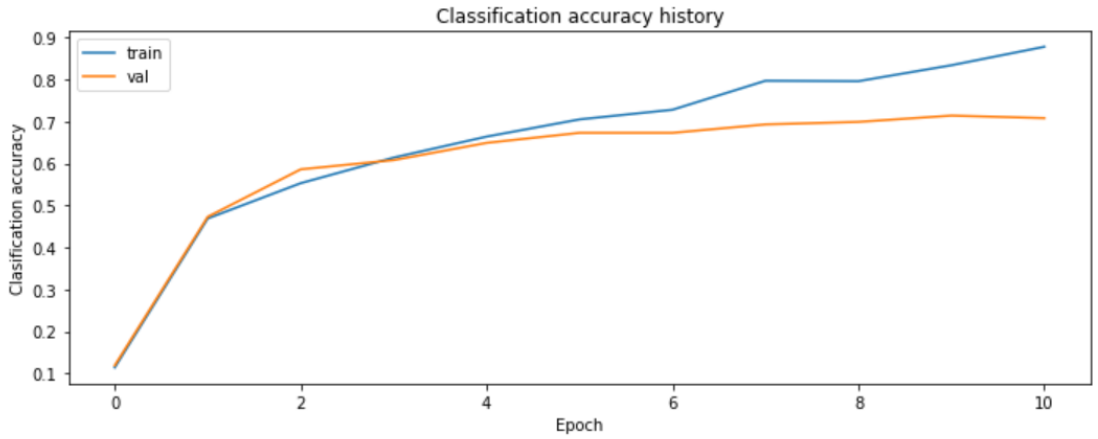


由上图我们可以看到：性能显著提高，符合要求。这表明5*5滤波器能够更好地捕捉图像中的特征，从而提高分类的准确率。

7*7：

```
(Iteration 1 / 4900) loss: 2.313376
(Epoch 0 / 10) train acc: 0.114000; val_acc: 0.119000
(Iteration 51 / 4900) loss: 2.006845
(Iteration 101 / 4900) loss: 1.893948
# ...
(Iteration 4801 / 4900) loss: 0.589659
(Iteration 4851 / 4900) loss: 0.511854
(Epoch 10 / 10) train acc: 0.878000; val_acc: 0.708000
```





尽管训练准确率最高，但验证准确率并没有比*5*5滤波器高，甚至略低。这可能表明使用*7*7滤波器的模型可能开始出现过拟合的迹象，即模型在训练集上表现很好，但在验证集上的表现并没有相应提高。

结论：

最佳滤波器尺寸：5*5滤波器在这三个选项中提供了最好的平衡，既有较高的训练准确率，也有较高的验证准确率，表明模型在未见数据上的泛化能力较好。

过拟合风险：7*7滤波器虽然训练准确率最高，但验证准确率并没有相应提高，这可能是过拟合的信号。

【模型代码】

```
class MyConvNet:
    def __init__(self, input_dim=(3, 32, 32), num_classes=10, weight_scale=1e-3,
reg=0.0,
                 dtype=np.float32):
        self.params = {}
        self.reg = reg
        self.dtype = dtype

        C, H, W = input_dim
        F1, F2 = 32, 64  # 卷积层滤波器数量
        HH, WW = 7, 7    # 使用3x3卷积核
        stride = 1
        pad = (HH - 1) // 2

        # 初始化权重 - 使用更小的weight_scale(1e-3)
        self.params['W1'] = np.random.normal(0, weight_scale, (F1, C, HH, WW))
```

```
        self.params['b1'] = np.zeros(F1)
        self.params['W2'] = np.random.normal(0, weight_scale, (F2, F1, HH, WW))
        self.params['b2'] = np.zeros(F2)

        # 计算卷积层输出尺寸
        conv1_out_h = 1 + (H + 2 * pad - HH) // stride
        conv1_out_w = 1 + (W + 2 * pad - WW) // stride
        pool1_out_h = conv1_out_h // 2
        pool1_out_w = conv1_out_w // 2

        conv2_out_h = 1 + (pool1_out_h + 2 * pad - HH) // stride
        conv2_out_w = 1 + (pool1_out_w + 2 * pad - WW) // stride
        pool2_out_h = conv2_out_h // 2
        pool2_out_w = conv2_out_w // 2

        # 打印卷积层和池化层的输出维度，确保它们是正确的
        print("Conv1 output shape:", (F1, conv1_out_h, conv1_out_w))
        print("Pool1 output shape:", (F1, pool1_out_h, pool1_out_w))
        print("Conv2 output shape:", (F2, conv2_out_h, conv2_out_w))
        print("Pool2 output shape:", (F2, pool2_out_h, pool2_out_w))

        # 全连接层
        hidden_dim = 500
        self.params['W3'] = np.random.normal(0, weight_scale,
                                             (F2 * pool2_out_h * pool2_out_w,
hidden_dim))
        self.params['b3'] = np.zeros(hidden_dim)
        self.params['W4'] = np.random.normal(0, weight_scale, (hidden_dim,
num_classes))
        self.params['b4'] = np.zeros(num_classes)

        # 转换为正确的数据类型
        for k, v in self.params.items():
            self.params[k] = v.astype(dtype)

    def loss(self, X, y=None):
        mode = 'test' if y is None else 'train'

        conv_param = {'stride': 1, 'pad': (7 - 1) // 2}  # 3x3卷积的padding
        pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        W3, b3 = self.params['W3'], self.params['b3']
        W4, b4 = self.params['W4'], self.params['b4']

        # 第一层: Conv -> ReLU -> Pool
        conv1, cache1 = conv_forward_fast(X, W1, b1, conv_param)
        relu1, cache_relu1 = relu_forward(conv1)
        pool1, cache_pool1 = max_pool_forward_fast(relu1, pool_param)

        # 第二层: Conv -> ReLU -> Pool
        conv2, cache2 = conv_forward_fast(pool1, W2, b2, conv_param)
        relu2, cache_relu2 = relu_forward(conv2)
        pool2, cache_pool2 = max_pool_forward_fast(relu2, pool_param)
```

```
        # 全连接层1: Affine -> ReLU
        fc1_input = pool2.reshape(pool2.shape[0], -1)
        fc1, cache_fc1 = affine_forward(fc1_input, W3, b3)
        relu3, cache_relu3 = relu_forward(fc1)

        # 全连接层2: Affine
        scores, cache_fc2 = affine_forward(relu3, W4, b4)

        if y is None:
            return scores

        loss, grads = 0, {}

        # 计算损失和梯度
        loss, dscores = softmax_loss(scores, y)
        # 添加L2正则化
        loss += 0.5 * self.reg * (np.sum(W1**2) + np.sum(W2**2) + np.sum(W3**2) +
np.sum(W4**2))

        # 反向传播
        dfc2, grads['W4'], grads['b4'] = affine_backward(dscores, cache_fc2)
        drelu3 = relu_backward(dfc2, cache_relu3)
        dpool2, grads['W3'], grads['b3'] = affine_backward(drelu3, cache_fc1)

        dpool2 = dpool2.reshape(pool2.shape)
        drelu2 = max_pool_backward_fast(dpool2, cache_pool2)
        dconv2 = relu_backward(drelu2, cache_relu2)
        dpool1, grads['W2'], grads['b2'] = conv_backward_fast(dconv2, cache2)

        drelu1 = max_pool_backward_fast(dpool1, cache_pool1)
        dconv1 = relu_backward(drelu1, cache_relu1)
        _, grads['W1'], grads['b1'] = conv_backward_fast(dconv1, cache1)

        # 添加正则化梯度
        grads['W4'] += self.reg * W4
        grads['W3'] += self.reg * W3
        grads['W2'] += self.reg * W2
        grads['W1'] += self.reg * W1

        return loss, grads
```

测试代码：

```
# 加载数据
data = get_CIFAR10_data()

# 初始化模型 - 使用更强的正则化(0.01)和更小的weight_scale(1e-3)
model = MyConvNet(weight_scale=1e-3, reg=0.01)

# 创建Solver实例 - 使用更小的学习率(1e-4)
solver = Solver(model, data,
                num_epochs=10,
                batch_size=100,
                update_rule='adam',
                optim_config={
```

```
                    'learning_rate': 1e-4,  # 更小的学习率
                },
                lr_decay=0.95,
                verbose=True,
                print_every=50)

# 训练模型
solver.train()
# 绘制训练过程
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history)
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, label='train')
plt.plot(solver.val_acc_history, label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
plt.legend()
plt.tight_layout()
plt.show()
```