实验报告

18340013 陈琮昊

1.模型结构：

对于卷积神经网络，其结构往往有如下*正则表达式*:

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC

其中**INPUT**为输入层，**CONV**为卷积层，**RELU**为激活函数，**POOL**为池化层，**FC**为全连接层。

比如对于`cnn.py`中所给`ThreeLayerConvNet`，其结构为：

INPUT -> [CONV -> RELU -> POOL] -> [FC -> RELU] -> FC

而在我自己实现的卷积神经网络中，我采用的结构如下：

INPUT -> [CONV -> RELU -> POOL]*2 -> [FC -> RELU] -> FC

在本次实验中我只改变了网络的结构，即使用两个卷积层，其余保持不变。参数的选定则与所给初始值一样，只不过加了一个学习率的衰减（*学习率以0.95的比率衰减*）。主要是因为几个小滤波器卷积层的叠加要比只使用一个大滤波器的卷积层要好：其能够提取到更多的特征、参数数量上也要少很多。当然，这种结构也会降低训练速度，增加了训练时间。

2.实验结果：

我实现的卷积神经网络如下（见`cnn.py->class myCNN()`）：

```
class myCNN(object):
    """
    the architecture of my ConvNets:

    conv - relu - 2x2 max pool - conv - relu - 2x2 max pool - affine - relu -
affine - softmax

    The network operates on minibatches of data that have shape (N, C, H, W)
    consisting of N images, each with height H and width W and with C input
    channels.
    """

    def __init__(
        self,
        input_dim=(3, 32, 32),
        num_filters=32,
        filter_size=7,
        hidden_dim=100,
        num_classes=10,
        weight_scale=1e-3,
        reg=0.0,
        dtype=np.float32
    ):
        """
        Initialize a new network.
```

```
        Inputs:
        - input_dim: Tuple (C, H, W) giving size of input data
        - num_filters: Number of filters to use in the convolutional layer
        - filter_size: Width/height of filters to use in the convolutional layer
        - hidden_dim: Number of units to use in the fully-connected hidden layer
        - num_classes: Number of scores to produce from the final affine layer.
        - weight_scale: Scalar giving standard deviation for random
initialization
          of weights.
        - reg: Scalar giving L2 regularization strength
        - dtype: numpy datatype to use for computation.
        """
        self.params = {}
        self.reg = reg
        self.dtype = dtype


 ############################################################################
        # TODO: Initialize weights and biases for the three-layer convolutional
   #
        # network. Weights should be initialized from a Gaussian centered at 0.0
  #
        # with standard deviation equal to weight_scale; biases should be
   #
        # initialized to zero. All weights and biases should be stored in the
   #
        #  dictionary self.params. Store weights and biases for the
convolutional  #
        # layer using the keys 'W1' and 'b1'; use keys 'W2' and 'b2' for the
  #
        # weights and biases of the hidden affine layer, and keys "W3" and 'b3'
   #
        # for the weights and biases of the output affine layer.
  #
        #
   #
        # IMPORTANT: For this assignment, you can assume that the padding
   #
        # and stride of the first convolutional layer are chosen so that
  #
        # **the width and height of the input are preserved**. Take a look at
   #
        # the start of the loss() function to see how that happens.
   #

 ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        C, H, W = input_dim

        # Convolutional layer
        self.params['W1'] = np.random.randn(num_filters, C, filter_size,
filter_size) * weight_scale
        self.params['b1'] = np.zeros(num_filters)
        self.params['W4'] = np.random.randn(num_filters, C, filter_size,
filter_size) * weight_scale
        self.params['b4'] = np.zeros(num_filters)
        # Hidden affine layer
```

```
        self.params['W2'] = np.random.randn(num_filters * (H // 2) * (W // 2),
hidden_dim) * weight_scale
        self.params['b2'] = np.zeros(hidden_dim)
        # Output affine layer
        self.params['W3'] = np.random.randn(hidden_dim, num_classes) *
weight_scale
        self.params['b3'] = np.zeros(num_classes)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

 ############################################################################
        #                             END OF YOUR CODE
  #
 ############################################################################

        for k, v in self.params.items():
            self.params[k] = v.astype(dtype)

    def loss(self, X, y=None):
        """
        Evaluate loss and gradient for the three-layer convolutional network.

        Input / output: Same API as TwoLayerNet in fc_net.py.
        """
        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"]
        # pass conv_param to the forward pass for the convolutional layer
        # Padding and stride chosen to preserve the input spatial size
        filter_size = W1.shape[2]
        conv_param = {"stride": 1, "pad": (filter_size - 1) // 2}

        # pass pool_param to the forward pass for the max-pooling layer
        pool_param = {"pool_height": 2, "pool_width": 2, "stride": 2}

        scores = None

 ############################################################################
        # TODO: Implement the forward pass for the three-layer convolutional
net,  #
        # computing the class scores for X and storing them in the scores
   #
        # variable.
   #
        #
   #
        # Remember you can use the functions defined in annp/fast_layers.py and
#
        # annp/layer_utils.py in your implementation (already imported).
#

 ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        conv_relu_pool_out, conv_relu_pool_cache = conv_relu_pool_forward(X, W1,
b1, conv_param, pool_param)
```

```
        conv_relu_pool_out2, conv_relu_pool_cache2 =
conv_relu_pool_forward(conv_relu_pool_out, W4, b4, conv_param, pool_param)
        affine_relu_out, affine_relu_cache =
affine_relu_forward(conv_relu_pool_out2, W2, b2)
        scores, scores_cache = affine_forward(affine_relu_out, W3, b3)

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

 ############################################################################
        #                                END OF YOUR CODE
  #

 ############################################################################

        if y is None:
            return scores

        loss, grads = 0, {}

 ############################################################################
        # TODO: Implement the backward pass for the three-layer convolutional
net, #
        # storing the loss and gradients in the loss and grads variables.
Compute  #
        # data loss using softmax, and make sure that grads[k] holds the
gradients #
        # for self.params[k]. Don't forget to add L2 regularization!
  #
        #
   #
        # NOTE: To ensure that your implementation matches ours and you pass the
  #
        # automated tests, make sure that your L2 regularization includes a
factor #
        # of 0.5 to simplify the expression for the gradient.
   #

 ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        loss, dscores = softmax_loss(scores, y)
        loss += 0.5 * self.reg * (np.sum(W1 * W1) + np.sum(W2 * W2) + np.sum(W3
* W3) + np.sum(W4 * W4))

        daffine_relu_out, dW3, db3 = affine_backward(dscores, scores_cache)
        dconv_relu_pool_out2, dW2, db2 = affine_relu_backward(daffine_relu_out,
affine_relu_cache)
        dconv_relu_pool_out, dW4, db4 =
conv_relu_pool_backward(dconv_relu_pool_out2, conv_relu_pool_cache)
        dx, dW1, db1 = conv_relu_pool_backward(dconv_relu_pool_out,
conv_relu_pool_cache)

        grads['W1'] = dW1 + self.reg * W1
        grads['W2'] = dW2 + self.reg * W2
        grads['W3'] = dW3 + self.reg * W3
        grads['W4'] = dW4 + self.reg * W4
        grads['b1'] = db1
        grads['b2'] = db2
```

```
        grads['b3'] = db3
        grads['b4'] = db4
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

 ############################################################################
        #                             END OF YOUR CODE
  #

 ############################################################################

        return loss, grads
```

在写好卷积神经网络后，需要在`jupyter`上写一个训练该网络的用例，运行该kernel来得到准确率。

根据`solver.py`内所给信息，可知训练一个卷积神经网络所写的用例程序模板如下：

```
Example usage might look something like this:

    data = {
      'X_train': # training data
      'y_train': # training labels
      'X_val': # validation data
      'y_val': # validation labels
    }
    model = MyAwesomeModel(hidden_size=100, reg=10)
    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    lr_decay=0.95,
                    num_epochs=10,
                    batch_size=100,
                    print_every=100)
    solver.train()
```

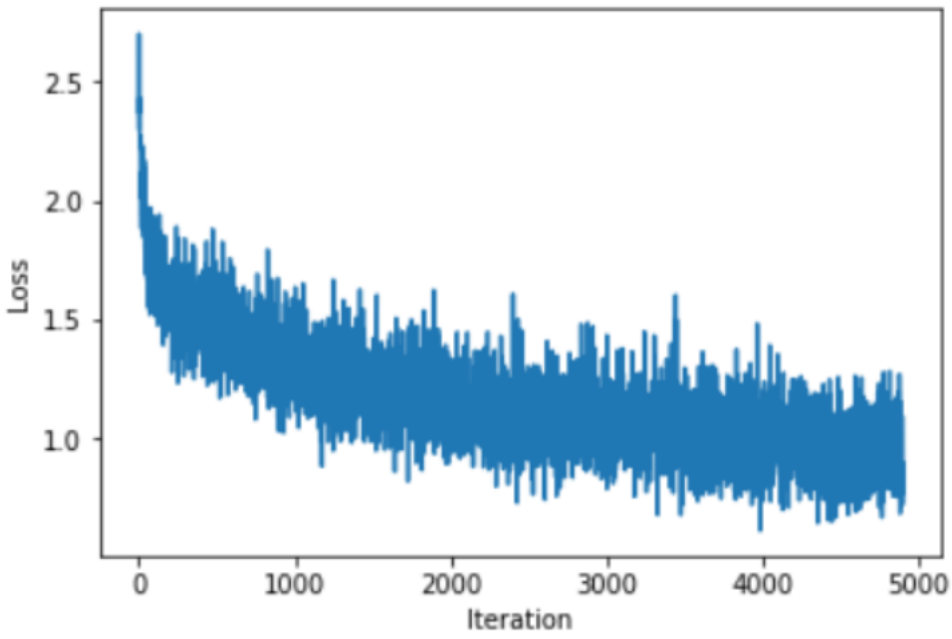
因此在`ConvolutionalNetworks.ipynb`上的运行代码如下：

```
## design and train your model
import numpy as np
import matplotlib.pyplot as plt
from annp.classifiers.cnn import *
from annp.data_utils import get_CIFAR10_data
from annp.gradient_check import eval_numerical_gradient_array,
eval_numerical_gradient
from annp.layers import *
from annp.fast_layers import *
from annp.solver import Solver
data = get_CIFAR10_data()
model = myCNN(weight_scale=0.001, hidden_dim=500, reg=0.001)
solver = Solver(model,data,
                num_epochs=10,
                lr_decay=0.95,
                batch_size=100,
                update_rule='adam',
                optim_config={
```

```
                  'learning_rate': 1e-3,
                },
                verbose=True,
                print_every=100)
solver.train()
print(solver.check_accuracy(data['X_train'],data['y_train']))
print(solver.check_accuracy(data['X_val'],data['y_val']))
fig=plt.figure()
ax=fig.add_subplot(111)
ax.set_xlabel("Iteration")
ax.set_ylabel("Loss")
ax.plot(solver.loss_history)
plt.show()
```

运行上述kernel结果如下：

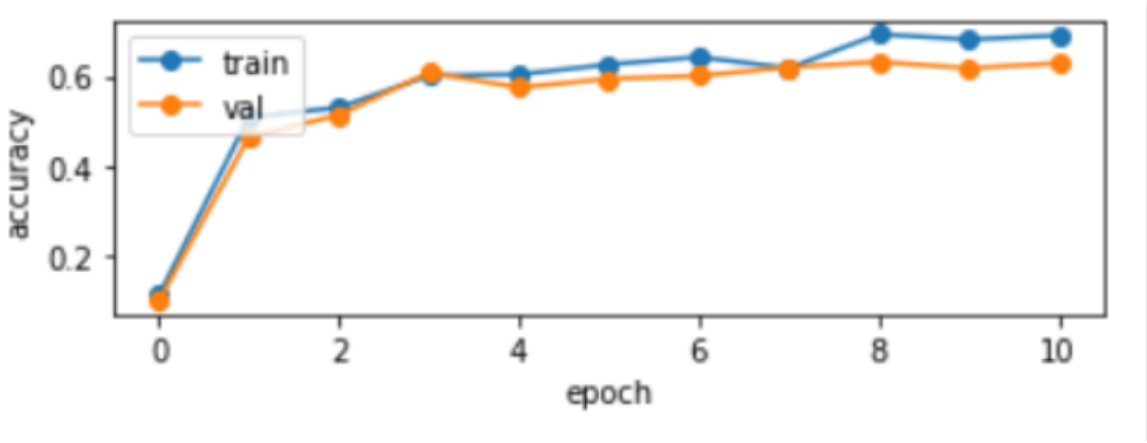
```
(Epoch 10 / 10) train acc: 0.691000; val_acc: 0.629000
0.6714489795918367
0.632
```



可以看到在训练集上的准确率为67.14%，验证集上的准确率为63.2%。损失随迭代次数的变化规律如上图。

```
plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

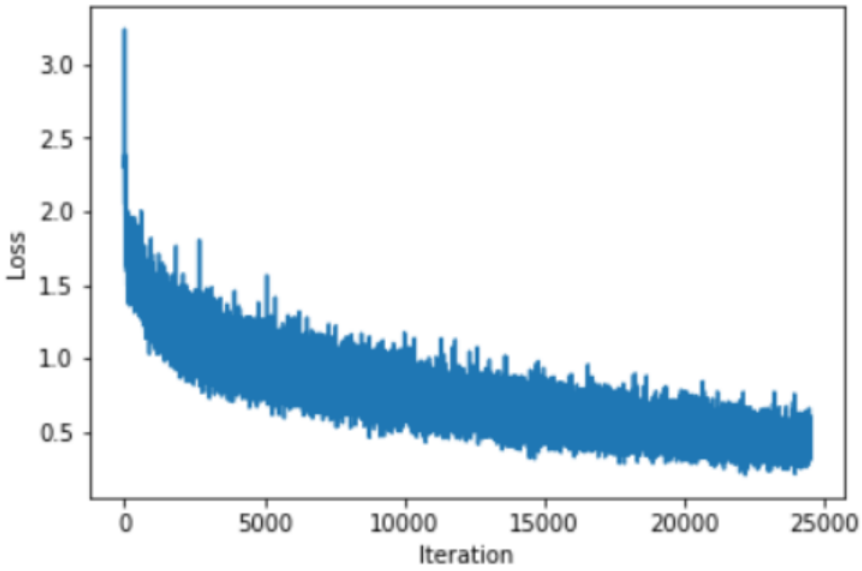
整个过程训练集和验证集的准确率变化过程如下：



3.调参对比：

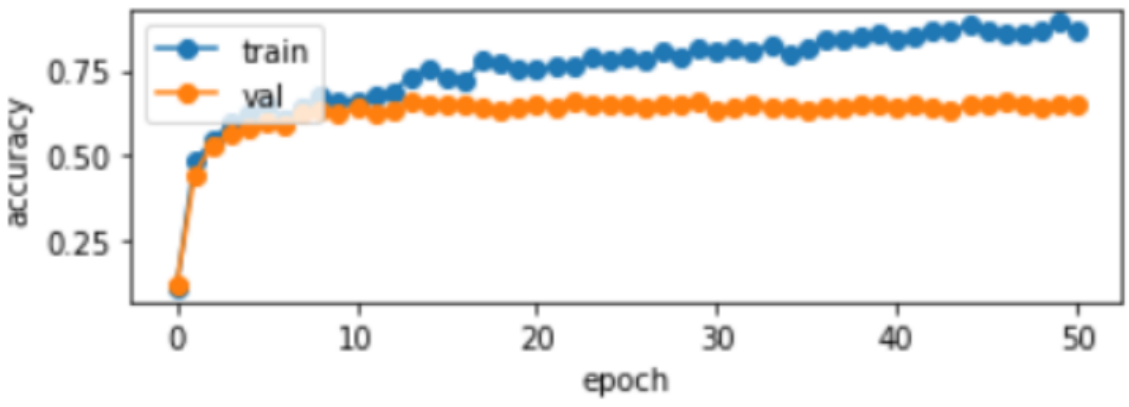
观察参数 `epoch` 对训练结果的影响，在上面的实验中 `num_epochs=10`；现调为 `50`，得到的结果如下：

```
0.7342448979591837
0.664
```



可以看到在训练集上的准确率为73.42%，验证集上的准确率为66.4%。损失随迭代次数的变化规律如上图。

整个过程训练集和验证集的准确率变化过程如下：



可以看到 `epoch=50` 时效果要好一些，但是会花费大量的训练时间。