School of Data and Computer Science, Sun Yat-sen University

E16 Deep Learning (C++/Python)

19214808 Yikun Liang

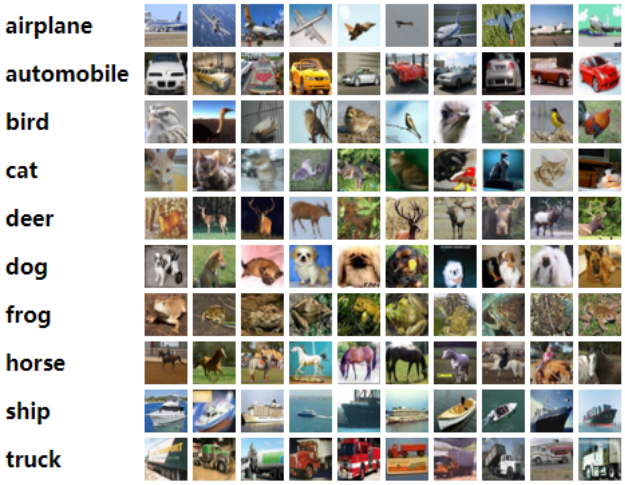
2019 年 12 月 28 日

目录

1 The CIFAR-10 dataset

The CIFAR-10 dataset (`http://www.cs.toronto.edu/~kriz/cifar.html`) consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. Here are the classes in the dataset, as well as 10 random images from each:



The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

2 Convolutional Neural Networks (CNNs / ConvNets)

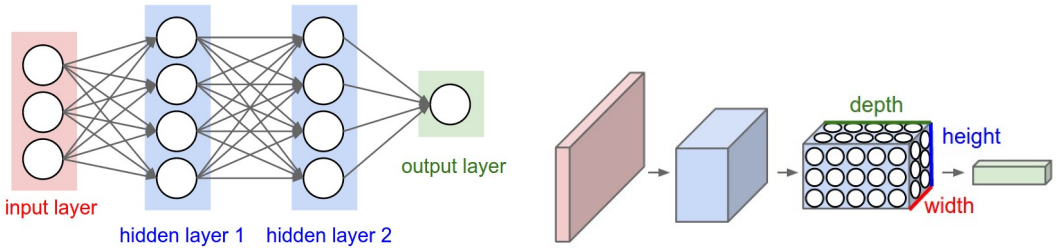
Chinese version: `https://www.zybuluo.com/hanbingtao/note/485480`
English version: `http://cs231n.github.io/convolutional-networks/#layers`

2.1 Architecture Overview

Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only of size $32 \times 32 \times 3$ (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have $32 * 32 * 3 = 3072$ weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example,

an image of more respectable size, e.g. $200 \times 200 \times 3$, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

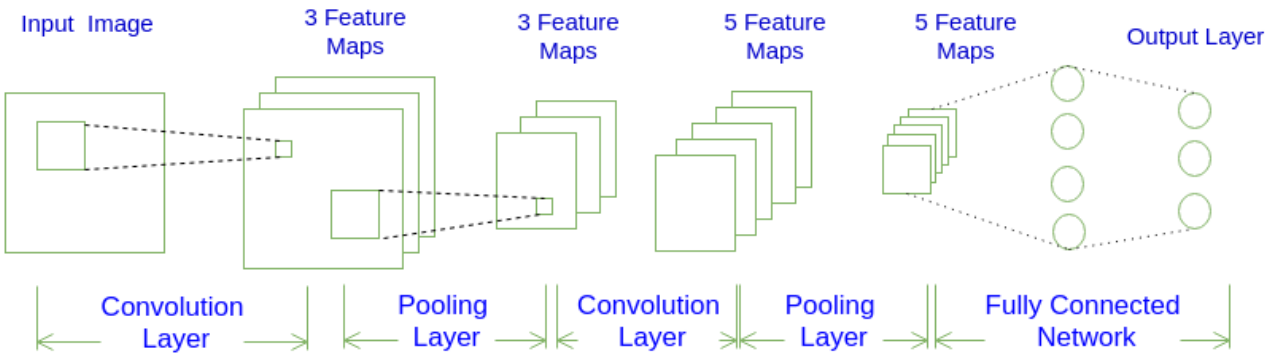
Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions $32 \times 32 \times 3$ (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions $1 \times 1 \times 10$, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

2.2 Layers used to build ConvNets

a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.



Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [**INPUT - CONV - RELU - POOL - FC**]. In more detail:

- INPUT $[32 \times 32 \times 3]$ will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as $[32 \times 32 \times 12]$ if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ($[32 \times 32 \times 12]$).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as $[16 \times 16 \times 12]$.
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size $[1 \times 1 \times 10]$, where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

2.2.1 Convolutional Layer

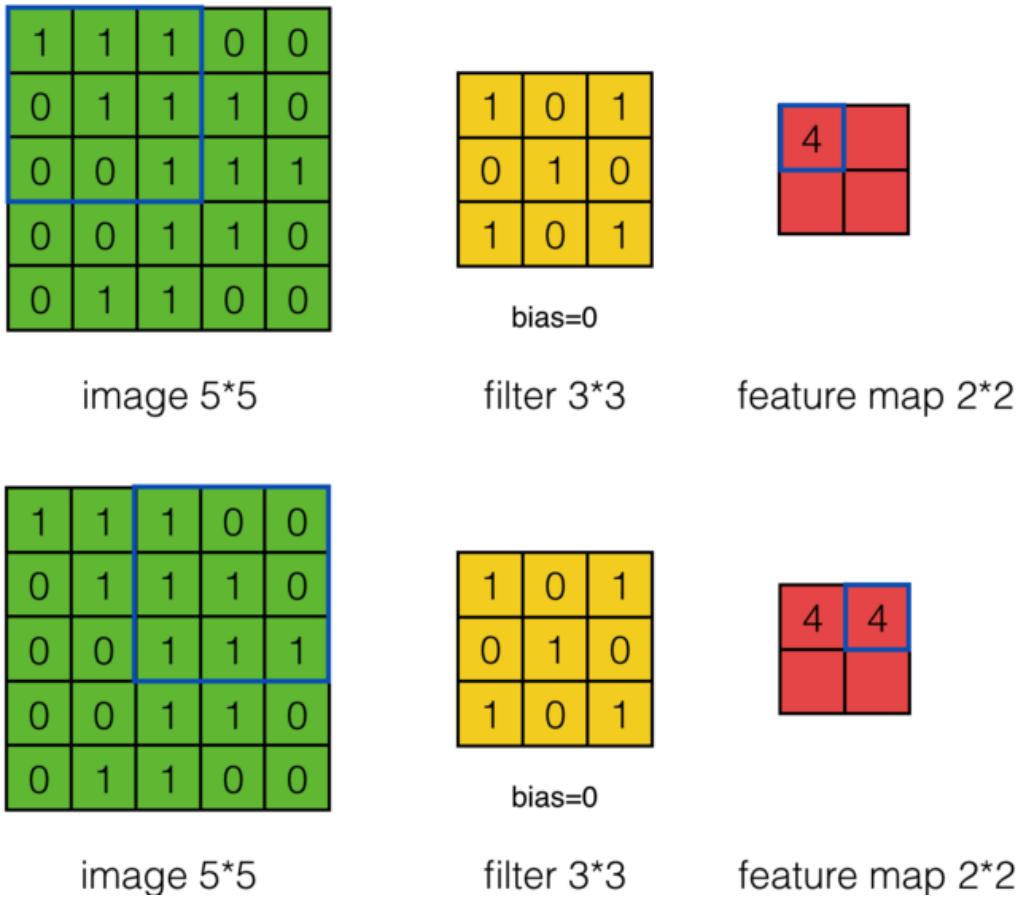
To summarize, the Conv Layer:

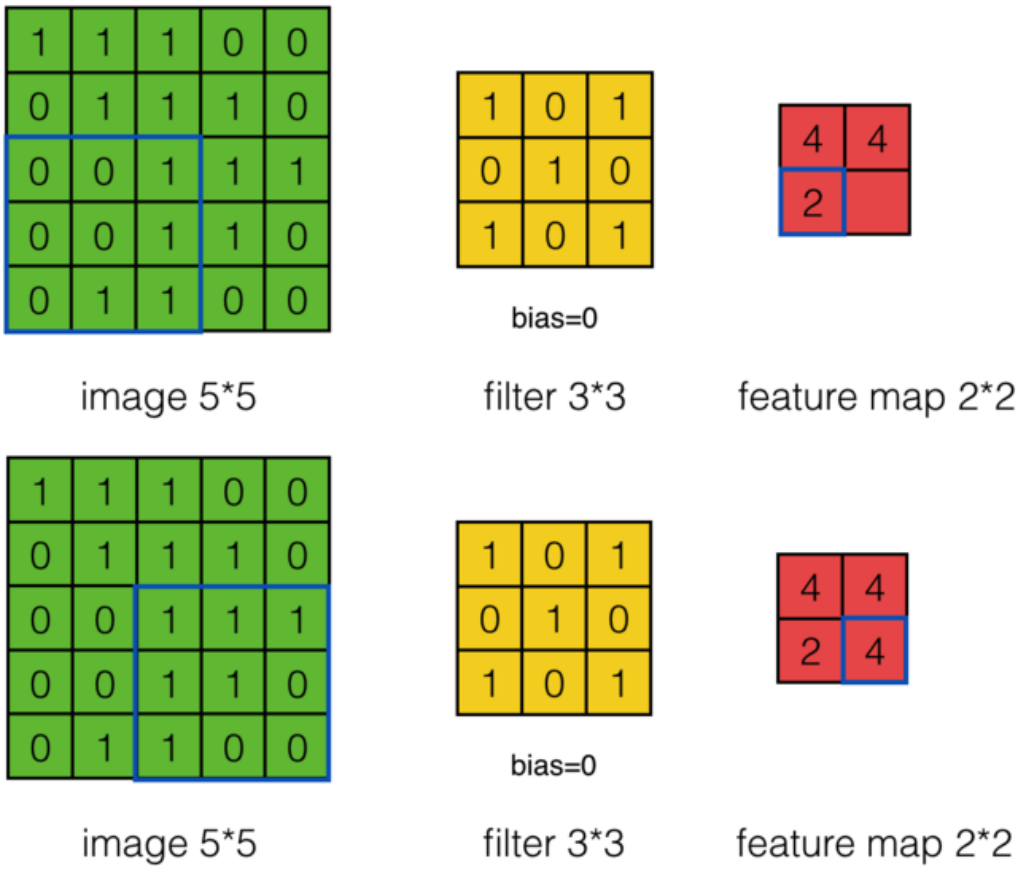
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,

  - the amount of zero padding P.

- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

A common setting of the hyperparameters is $F = 3$,$S = 1$,$P = 1$. However, there are common conventions and rules of thumb that motivate these hyperparameters.





2.2.2 Pooling Layer

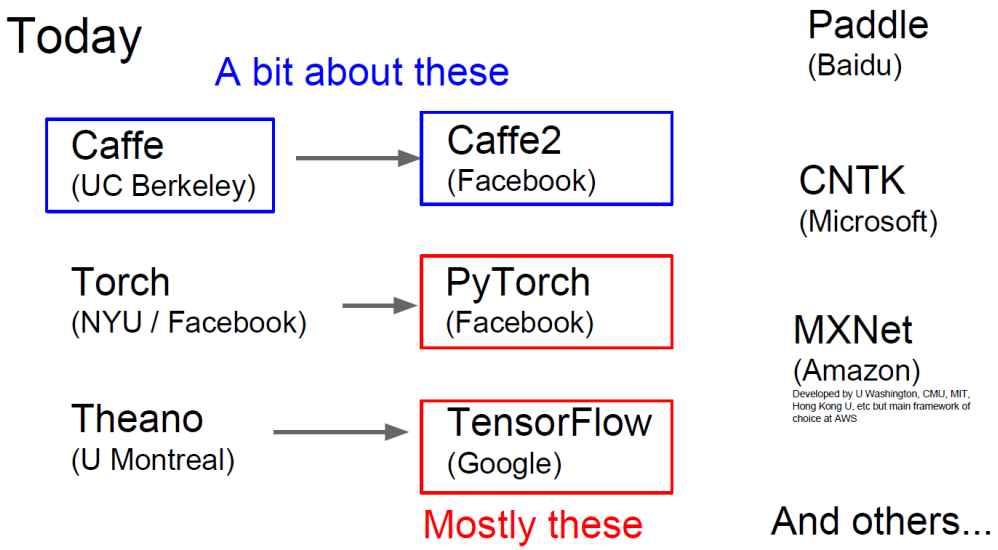
It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the **MAX** operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2×2 region in some depth slice). The depth dimension remains unchanged. More generally, the pooling layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - their spatial extent F,
  - the stride S,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$
- $D2 = D1$

- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

3 Deep Learning Softwares



4 Tasks

1. Given the data set in the first section, please implement a convolutional neural network to calculate the accuracy rate. The major steps involved are as follows:
   (a) Reading the input image.
   (b) Preparing filters.
   (c) Conv layer: Convolving each filter with the input image.
   (d) ReLU layer: Applying ReLU activation function on the feature maps (output of conv layer).
   (e) Max Pooling layer: Applying the pooling operation on the output of ReLU layer.
   (f) Stacking conv, ReLU, and max pooling layers

2. You can refer to the codes in `cs231n`. Don't use Keras, TensorFlow, PyTorch, Theano, Caffe, and other deep learning softwares.
3. Please submit a file named `E16_YourNumber.rar`, which should includes the code files and the result pictures, and send it to `ai_201901@foxmail.com`

5 Codes and Results

5.1 main 函数关键代码

```
# 导入训练集
data = get_CIFAR10_data()

# 初始化网络和优化器，进行训练
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=50, batch_size=100,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                  #'learning_rate': 0.002,
                },
                lr_decay = 0.97,
                print_every = 20,
                verbose = True)
solver.train()
```

5.2 实验心得

下面讲一下自己对于代码的理解，Solver 优化器主要是一些凸优化的知识点，不在课堂讲授的内容中，所以略过。主要讲一下主函数中调用的 ThreeLaterConvNet 部分

首先是定义部分

输入维度为 32*32*3，也就是 CIFAR-10 数据集的图片格式，输出维度为 10，代表十个分类。

W1,W2,W3 为三个权重矩阵，三个权重矩阵的维度设置需要根据一些公式进行计算。W1 为卷积层的权重矩阵，卷积层中 filternum=32,filter=(7, 7)，根据 filter 的设置以及图片信息进行维度设置。

W2 为计算池化层输出的权重矩阵，根据代码可以看出 pool size 为 (2, 2), 与 pool_param 中的设置相符

最后 W3 为输出层，因为有十个类型的分类，所以输出的维度为 10(num_class)

```
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):
self.params = {}
self.reg = reg
self.dtype = dtype

C, H, W=input_dim
F=num_filters

self.params['W1'] = weight_scale * np.random.randn(F,C,filter_size,filter_size)
self.params['b1'] = np.zeros(F)

#2*2 pooling,H and W all have been divided by 2
self.params['W2'] = weight_scale * np.random.randn(F*int(H/2)*int(W/2), hidden_dim)
self.params['b2'] = np.zeros(hidden_dim)

self.params['W3'] = weight_scale * np.random.randn(hidden_dim, num_classes)
self.params['b3'] = np.zeros(num_classes)

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

接下来为 loss 函数

这个函数的主要作用是，先进行前向传播，计算 loss，然后进行后向传播，计算梯度，在 solver 类中每一个 step 都会调用这个类，得到误差和梯度，调整网络的参数。

前向传播，先计算卷积层，再计算池化层，最后输出层

```
out, con_cachei = conv_relu_pool_forward(X, W1, b1, conv_param, pool_param)
out, aff1_relu_cache=affine_relu_forward(out, W2, b2)
out, aff2_cache=affine_forward(out,W3, b3)
```

计算 loss

```
data_loss, dout = softmax_loss(scores, y)
W_square_sum = 0
for layer in range(3):
        Wi = self.params['W%d' % (layer + 1)]
        W_square_sum += (np.sum(Wi ** 2))
reg_loss = 0.5 * self.reg * W_square_sum
loss = data_loss + reg_loss
```

后向传播计算梯度

```
dout, dW3, db3=affine_backward(dout, aff2_cache)
dout, dW2, db2=affine_relu_backward(dout, aff1_relu_cache)
dout,dW1,db1=conv_relu_pool_backward(dout, con_cachei)

dW1+=self.reg*W1
dW2 += self.reg * W2
dW3 += self.reg * W3
grads['W1'] = dW1
grads['b1'] = db1
grads['W2'] = dW2
grads['b2'] = db2
grads['W3'] = dW3
grads['b3'] = db3
```

接下来三个前向传播的函数

首先是 **conv_relu_pool_forward**, 作用是将数据输入卷积层，然后输出连接到一个 relu 层，最后再通过 pool 层。**affine_relu_forward** 就是一个常规的 relu 层，最后 **affine_forward** 经过一个全连接层，输出结果。

实验结果如下, 最终测试集精度在 65% 左右，而训练集精度可以高达 90%，应该是出现过拟合了。图2中看出，测试集精度在迭代大约 10 轮后就不再有明显的提升，而训练集精度则一直在上升。

```
(Iteration 24181 / 24500) loss: 0.355476
(Iteration 24201 / 24500) loss: 0.381417
(Iteration 24221 / 24500) loss: 0.407681
(Iteration 24241 / 24500) loss: 0.406724
(Iteration 24261 / 24500) loss: 0.315189
(Iteration 24281 / 24500) loss: 0.412932
(Iteration 24301 / 24500) loss: 0.298980
(Iteration 24321 / 24500) loss: 0.389248
(Iteration 24341 / 24500) loss: 0.351589
(Iteration 24361 / 24500) loss: 0.425324
(Iteration 24381 / 24500) loss: 0.273327
(Iteration 24401 / 24500) loss: 0.357316
(Iteration 24421 / 24500) loss: 0.441135
(Iteration 24441 / 24500) loss: 0.423455
(Iteration 24461 / 24500) loss: 0.388743
(Iteration 24481 / 24500) loss: 0.315396
(Epoch 50 / 50) train acc: 0.919000; val_acc: 0.638000
```

图 1: Accuracy

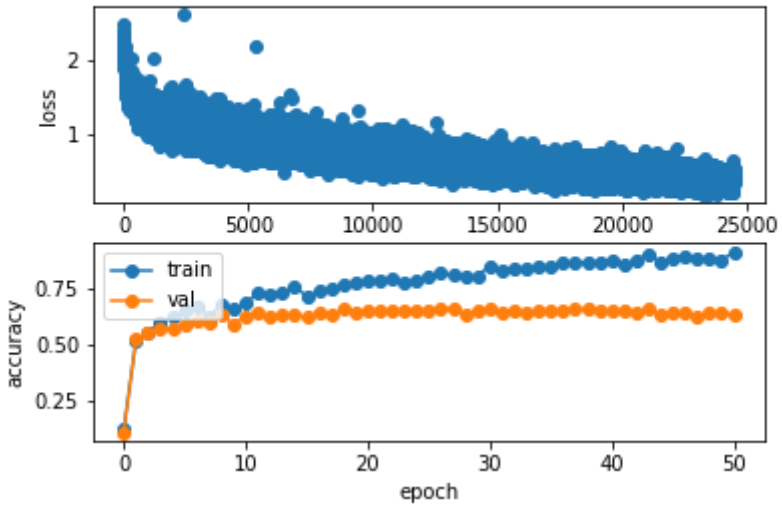

loss
2
1
0
5000
10000
15000
20000
25000
train
val
accuracy
0.75
0.50
0.25
0
10
20
30
40
50
epoch

图 2: loss