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

E15 Deep Learning (C++/Python)

18364066 Lu Yanzuo

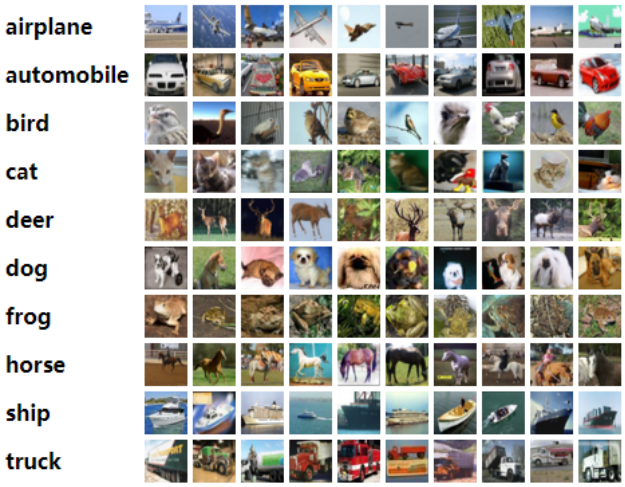
December 22, 2020

Contents

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)

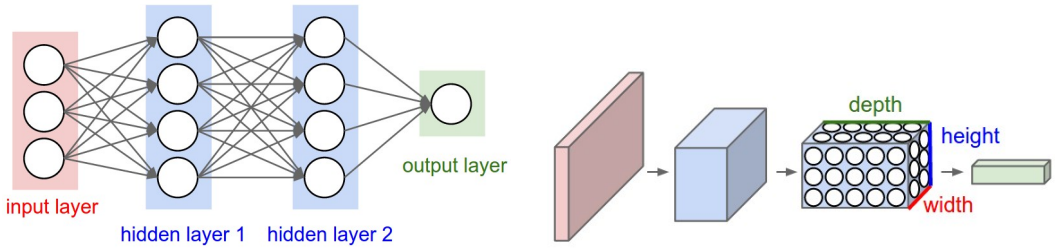
Chinese version: https://www.zybuluo.com/hanbingtao/note/485480
English version: http://cs231n.github.io/convolutional-networks/#layers

2.1 Architecture Overview

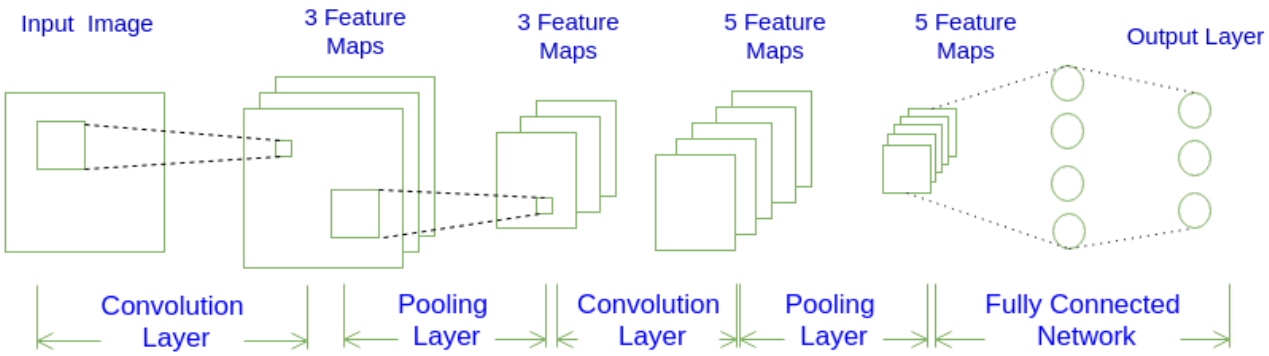
Regular Neural Nets don't scale well to full images. In CIFAR-10, images are only of size $32 \times 32 \times 3$ (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have $32 * 32 * 3 = 3072$ weights. This amount still seems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

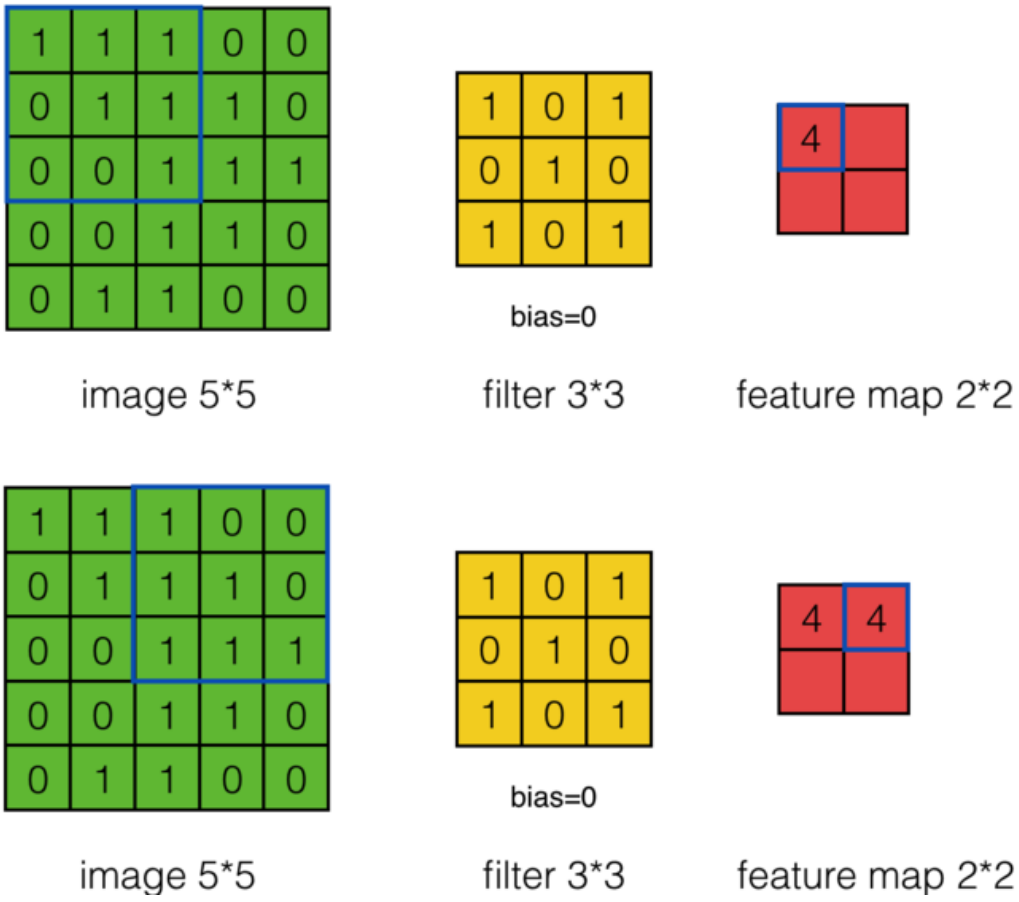
To summarize, the Conv Layer:

- 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)
- $D_2 = K$

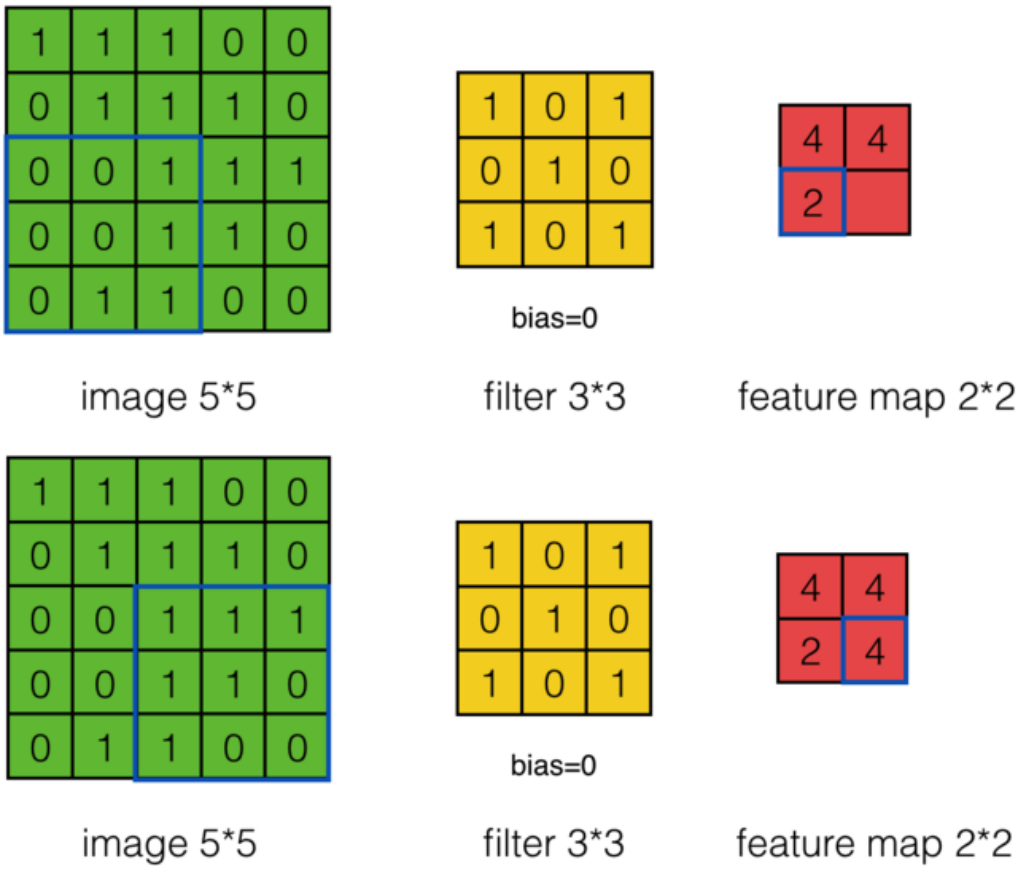
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input 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

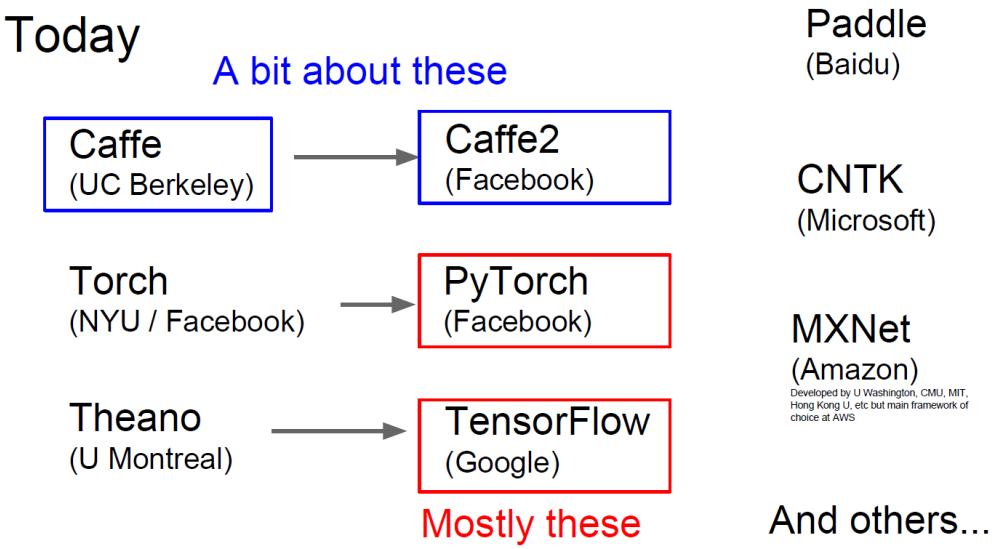
It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The 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 `E15_YourNumber.rar`, which should includes the code files and the result pictures, and send it to `ai_2020@foxmail.com`

5 Codes and Results

根据 TA 提供的 CS231N 的课程代码，我们只需要配置好相应的环境并且修改一些 import 的 bug 即可，例如 scipy 中的 imread 函数在最新版本已经被废弃了，可以使用 imageio 中的 imread，注意这

里不能直接使用 cv2 的 imread，因为 cv2 默认的颜色顺序是 BGR 而原来的 scipy 中的 imread 颜色顺序是 RGB，除此之外还需要安装好 Cython 用于加速，下面是我用来训练的代码。

```
from classifiers.cnn import *
from solver import *
from data_utils import *
import matplotlib.pyplot as plt

if __name__ == '__main__':
        dataset = get_CIFAR10_data()
        model = ThreeLayerConvNet(hidden_dim=500,reg=1e-3)
        solver = Solver(model, dataset, update_rule='adam', optim_config
           ={'learning_rate':1e-3,},
                lr_decay=0.95, num_epochs=50, print_every=100)
        solver.train()

        plt.plot(solver.loss_history)
        plt.xlabel("Iteration")
        plt.ylabel("Loss")
        plt.show()
```

这里我采用的隐藏层维度是 500，使用的梯度下降方法是 Adam，这是结合了 gradient descent 和 monmentum 的一种做法，相较于原有的传统梯度下降具有速度更快等有限，学习率用了比较高的 1e-3，总共跑了 50 个 epoch，但事实上 10 个 epoch 以后再往下训练就已经开始过拟合了，后续的 loss 下降也没有什么意义，最后的准确度只有百分之 60 左右，考虑到模型结构比较简单，且学习率不好做动态调整，这样的精度也是相对合理的，下面是 loss 下降图以及每 10 个 epoch 后的测试精度。

```
(Iteration 4501 / 24500) loss: 0.858987
(Iteration 4601 / 24500) loss: 0.951916
(Iteration 4701 / 24500) loss: 0.938186
(Iteration 4801 / 24500) loss: 1.095315
(Epoch 10 / 50) train acc: 0.686000; val_acc: 0.646000
```

```
(Iteration 9401 / 24500) loss: 0.649831
(Iteration 9501 / 24500) loss: 0.875437
(Iteration 9601 / 24500) loss: 0.689534
(Iteration 9701 / 24500) loss: 0.737394
(Epoch 20 / 50) train acc: 0.753000; val_acc: 0.645000
```

```
(Iteration 14301 / 24500) loss: 0.467293
(Iteration 14401 / 24500) loss: 0.478162
(Iteration 14501 / 24500) loss: 0.691181
(Iteration 14601 / 24500) loss: 0.480326
(Epoch 30 / 50) train acc: 0.847000; val_acc: 0.651000
```

```
(Iteration 19201 / 24500) loss: 0.290030
(Iteration 19301 / 24500) loss: 0.368500
(Iteration 19401 / 24500) loss: 0.346332
(Iteration 19501 / 24500) loss: 0.412505
(Epoch 40 / 50) train acc: 0.884000; val_acc: 0.637000
```

```
(Iteration 24101 / 24500) loss: 0.369909
(Iteration 24201 / 24500) loss: 0.433693
(Iteration 24301 / 24500) loss: 0.244366
(Iteration 24401 / 24500) loss: 0.297380
(Epoch 50 / 50) train acc: 0.910000; val_acc: 0.634000
```