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

E16 Deep Learning (C++/Python)

17341015 Hongzheng Chen

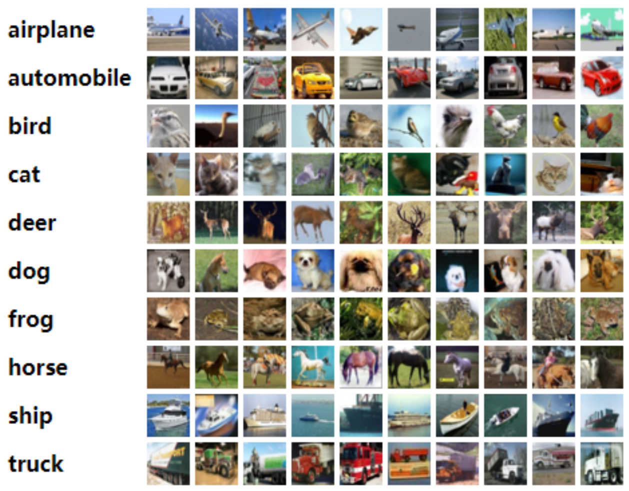
December 26, 2019

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.

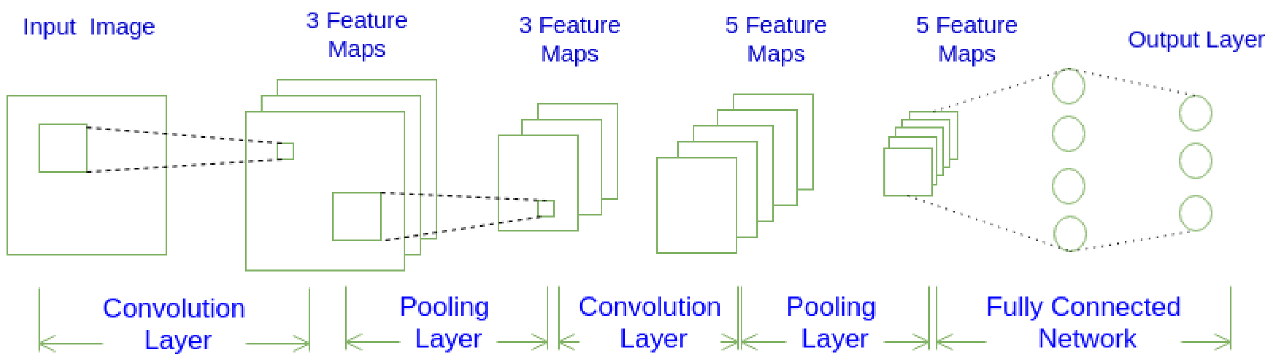
2 Convolutional Neural Networks (CNNs / ConvNets)

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 dont 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:

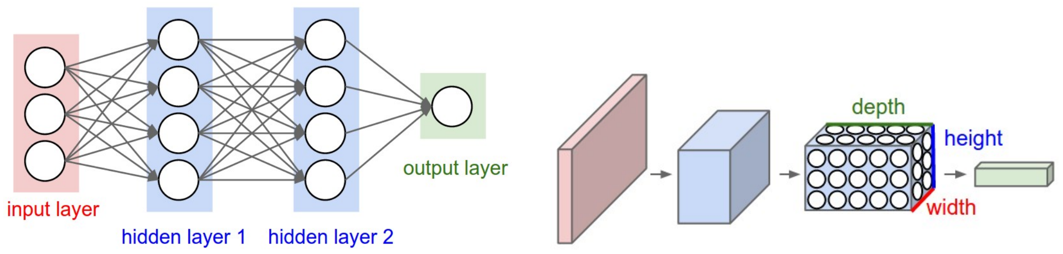


Figure 1: *
Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input 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.

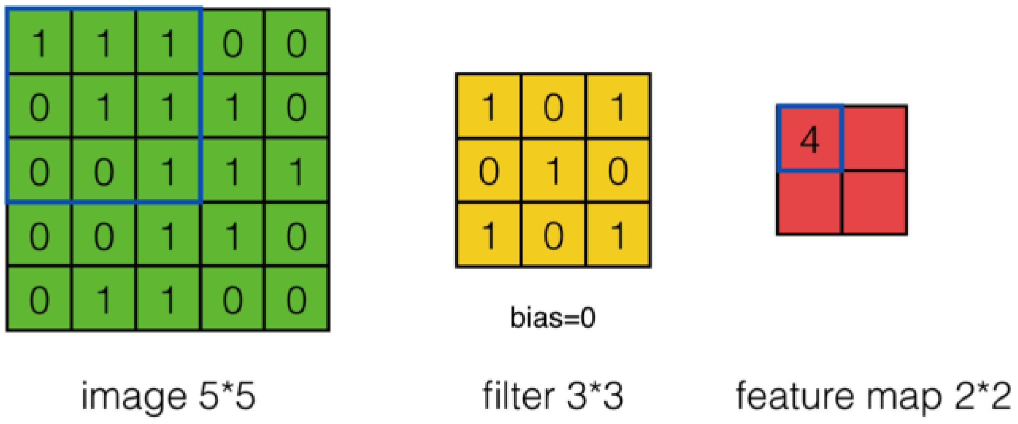
- RELU layer will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ($[32 \times 32 \times 12]$).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such 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.

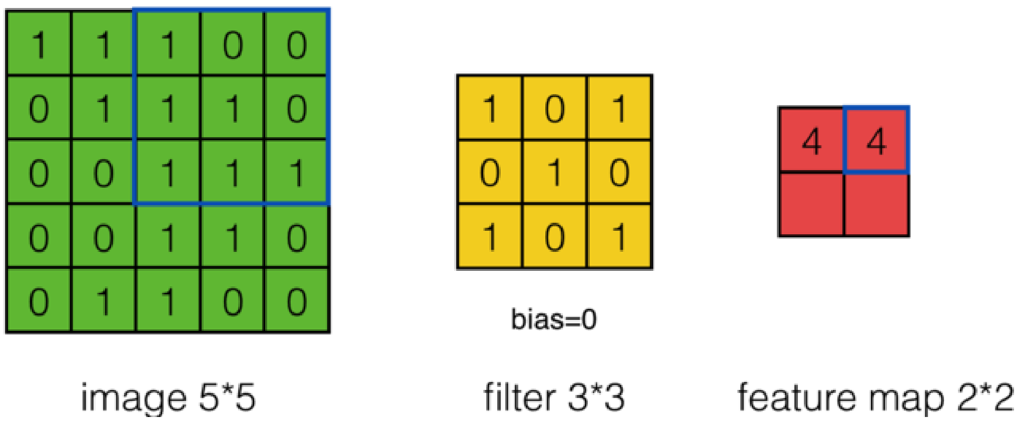
2.2.1 Convolutional Layer

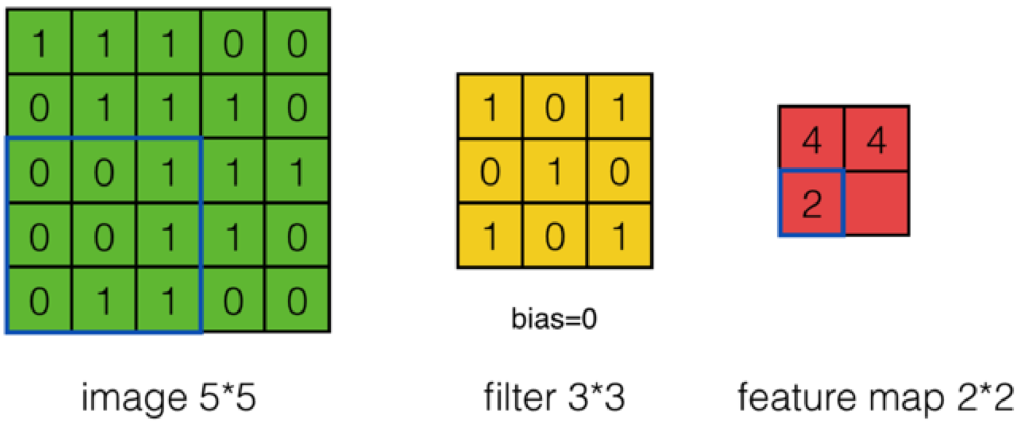
To summarize, the Conv Layer:

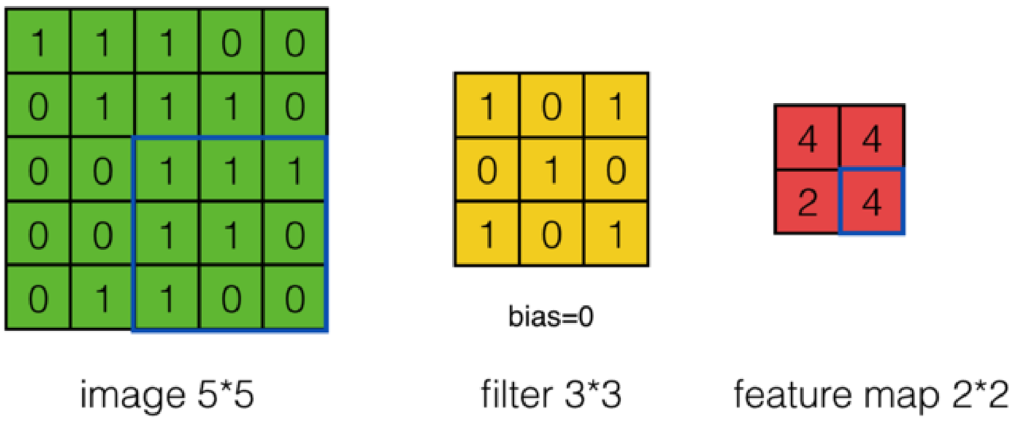
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (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.







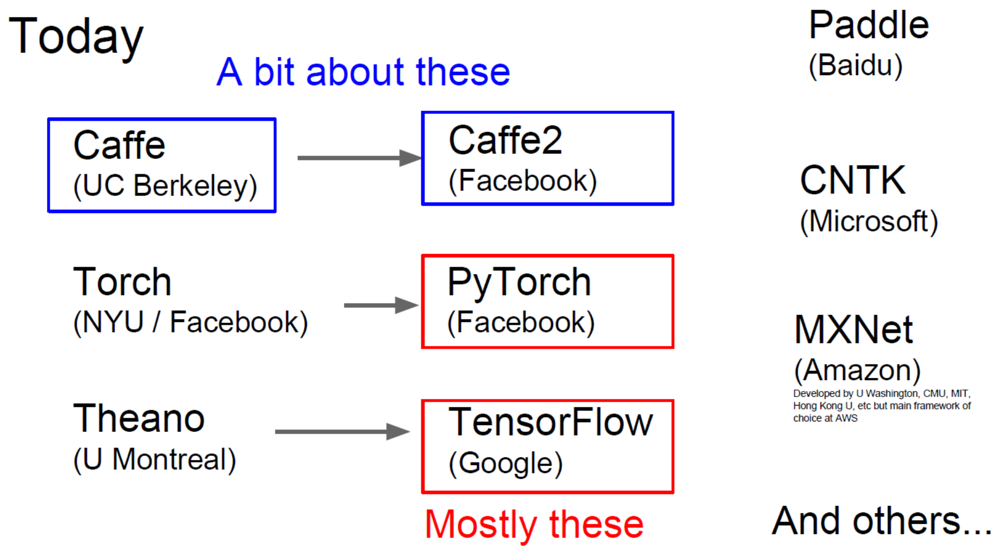


2.2.2 Pooling Layer

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

Following the steps described above, I glue different modules in `cs231n` together.

(a) To read the input images, I leverage the `get_CIFAR10_data` function in `data_utils.py`, where it separate the data into training set, validation set, and test set, all of which are stored in a dictionary. Moreover, it normalizes the data for easy training later. The return data has the shape as follows,

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,)

Notice the channel dimension has been swap to axis 1 in numpy representation.

(b) The filters are defined in `layers.py`. The naive implementation of convolutional layer needs four nested loops. (Actually if counting the numpy summation implementation, there will be more loops.) In the innerest loop, the original image makes a dot product with the convolutional kernel.

```
for ni in range(x.shape[0]):
    for fi in range(w.shape[0]):
        for xi in range(H_):
            for yi in range(W_):
                out[ni, fi, xi, yi] = np.sum(x_pad[ni, :, xi * stride:xi * stride + HH,
                    ↪  yi * stride:yi * stride + WW] * w[fi, :, :, :])

        out[ni,fi,:,:]+=b[fi]
```

The implementation of Max Pooling layer is similar, except for the max function. The key computation is listed below. The size of max pooling output is much easier to be calculated than the convolutional layer, since the stride is commonly set equal to the size of the max pooling kernel.

```
for ni in range(x.shape[0]):
    for ci in range(x.shape[1]):
        for xi in range(H_):
            for yi in range(W_):
                out[ni, ci, xi, yi] = np.max(x[ni,ci,xi * stride:xi * stride +
                    ↪ pool_height,yi * stride:yi * stride + pool_width])
```

The backward pass of the convolutional layer and the max pooling layer are much harder than the forward pass, and the naive implementation also has great performance degradation. For example, backpropagation through max pooling layer needs to find the index that generates the current maximum, which needs a tedious traversal in each convolution. This is a large overhead if the traversal does not deal well.

(c,d,e,f) Once the key components are built, what we need to do is to glue them. The CNN has been encapsulated as `ThreeLayersConvNet` in `cnn.py`. The architecture is defined below,

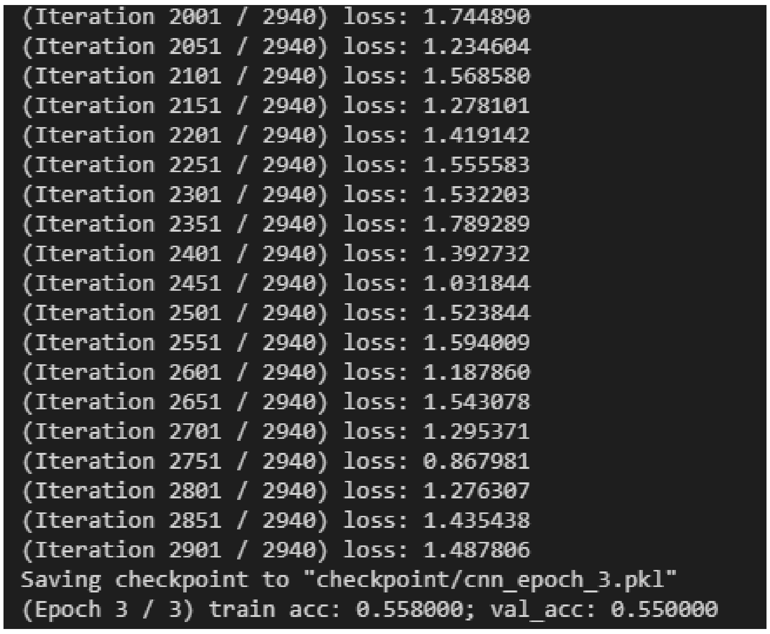
conv - relu - 2x2 max pool - affine - relu - affine - softmax

Then we directly call the forward pass and the backward pass of these layers, and the CNN will work properly. Notice the weights and biases should be stored when propagation in order to calculate the loss faster.

After the three-layer CNN is built, I take the most of the `Solver`. Just instantiate the data and model, and pass them into the solver. Press `solver.train()`, and the CNN will train automatically. The codes are listed below.

```
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.solver import Solver

data = get_CIFAR10_data()
for k, v in data.items():
    print('{}: '.format(k), v.shape)


# set up model
model = ThreeLayerConvNet(hidden_dim=512, reg=0.001)

# encapsulate it into solver
solver = Solver(model, data,
               num_epochs=1,
               batch_size=50,
               update_rule='adam',
               optim_config={
                 'learning_rate': 1e-3,
               },
               verbose=True,
               print_every=50,
               checkpoint_name="checkpoint/cnn")
solver.train()


solver.check_accuracy(data["X_test"],data["y_test"])


import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_xlabel("Iteration")
ax.set_ylabel("Loss")
ax.plot(solver.loss_history)
fig.savefig("fig/loss.pdf",format="pdf")
```

I only train the CNN for three epochs, obtaining 55.8% and 55% of accuracy on train set and validation set representatively, as shown in Fig. 2. The accuracy on test set is 54.5%.



```
(Iteration 2001 / 2940) loss: 1.744890
(Iteration 2051 / 2940) loss: 1.234604
(Iteration 2101 / 2940) loss: 1.568580
(Iteration 2151 / 2940) loss: 1.278101
(Iteration 2201 / 2940) loss: 1.419142
(Iteration 2251 / 2940) loss: 1.555583
(Iteration 2301 / 2940) loss: 1.532203
(Iteration 2351 / 2940) loss: 1.789289
(Iteration 2401 / 2940) loss: 1.392732
(Iteration 2451 / 2940) loss: 1.031844
(Iteration 2501 / 2940) loss: 1.523844
(Iteration 2551 / 2940) loss: 1.594009
(Iteration 2601 / 2940) loss: 1.187860
(Iteration 2651 / 2940) loss: 1.543078
(Iteration 2701 / 2940) loss: 1.295371
(Iteration 2751 / 2940) loss: 0.867981
(Iteration 2801 / 2940) loss: 1.276307
(Iteration 2851 / 2940) loss: 1.435438
(Iteration 2901 / 2940) loss: 1.487806
Saving checkpoint to "checkpoint/cnn_epoch_3.pkl"
(Epoch 3 / 3) train acc: 0.558000; val_acc: 0.550000
```

Figure 2: Accuracy of train set and validation set

The training loss is shown in Fig. 3.

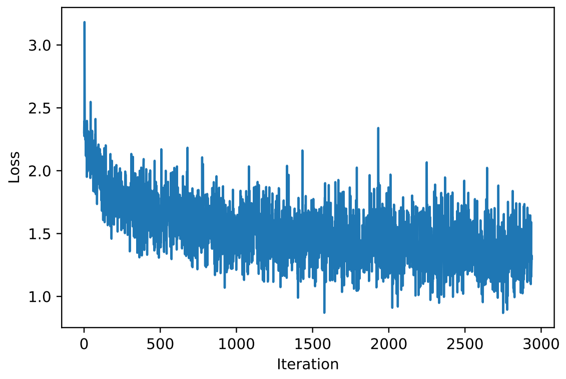


Figure 3: Training loss

From this experiment, I clearly realize that how slow the CPU is! Even though the CIFAR dataset is such a toy dataset in nowadays deep learning research, it takes almost an hour to train only 3 epochs using my Intel i7 CPU, which is extremely unacceptable. The code in `cs231n` has been optimized using the ahead of time (AOT) compiler Cython. However, the training process still takes a long time. And we may know how much optimization has been done in Pytorch and Tensorflow, where C/C++ and Python codes are deeply fused. Both of the frameworks greatly push the development of deep learning. Moreover, the deployment of GPU in deep learning truly leads the AI field to take a big step. That is why somebody said the Turing Award should be awarded to Jen-Hsun Huang, the CEO of Nvidia.

Moreover, we can observe a marginal effect decrement in Fig. 3. Actually after the first epoch, the CNN has reached 50% accuracy on the train set. The two more times of training only improve the performance for 5%. The investment and the return are not consistent. This also makes us to think if it necessary to take so many resources to improve only little prediction accuracy in nowadays deep learning research, or this kind of improvement is just anything but useful.

At last, to summarize the AI course this term, I have to say this is the best course I have ever toke in SYSU. Starting from simple searching algorithms, we got through logic, knowledge representation, planning, Bayes, and finally got the place of machine learning and deep learning. Though the number of experiments is double of those of CS major, we learned a lot and built lots of artificial idiots, which is also a kind of fun!

Last but not least, great thanks to TAs, especially to Yukun :), who provide us lots of guidance on the road to the peak of artificial intelligence, and give us lots of happiness and warmness.