MNIST Digit Classification with CNN

Background

In this assignment, we will continue working on MNIST digit classification by utilizing convolutional neural networks (CNNs). The final loss layer is changed to cross-entropy loss layer where output probability is generated by softmax function.

The main challenge is to implement the forward and back-propagation functions of convolutional layer and pooling layer from scratch! If you succeed, you will get a profound understanding of deep learning mechanism. So roll up your sleeves and get started!

Requirements

- python 2.7 or python 3.6
- numpy >= 1.12.0
- scipy >= 0.17.0

Dataset Description

To load data, you may use

```python
from load_data import load_mnist_4d
train_data, test_data, train_label, test_label = load_mnist_4d('data')
```

Then `train_data`, `train_label`, `test_data` and `test_label` will be loaded in `numpy.array` form. Digits range from 0 to 9, and the corresponding labels are from 0 to 9.

NOTE: Image data passing through the network should be a 4-dimensional tensor with $N \times C \times H \times W$ dimensions, where H and W denote the height and width of the given image. C denotes the number of image channels (1 for gray scale and 3 for RGB scale), and N denotes the batch size. Among these dimensions, the channel number C would be a confusing concept for hidden layer outputs. In this context, we interpret the i-th channel of the n-th sample in one mini-batch (`output[n, i, :, :]` written in python syntax) as **one** feature map in the output by using **one** filter W_i.

Python Files Description

`layers.py`, `network.py` and `solve_net.py` are identical to those included in Homework 1. `run_cnn.py` is the main script for running the whole program. It demonstrates how to define a neural network by sequentially adding layers and train the net.

NOTE: Any modifications of these files or adding extra python files should be explained and documented in README.

The new loss layer in `loss.py` is `SoftmaxCrossEntropyLoss`. `Softmax` function can map the input to a probability distribution in the following form:

$$P(t_k = 1|\boldsymbol{x}) = \frac{exp(x_k)}{\sum_{j=1}^{K} exp(x_j)}$$

where x_k is the k-th component in the input vector $\boldsymbol{x}$ and $P(t_k = 1|\boldsymbol{x})$ indicates the probability of being classified to class k. Given the ground-truth labels $\boldsymbol{t}^{(1)}, \cdots, \boldsymbol{t}^{(N)}$ (one-hot encoding form) and the corresponding predicted vectors $\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(N)}$, `SoftmaxCrossEntropyLoss` can be computed in the form $E = \frac{1}{N}\sum_{n=1}^{N} E^{(n)}$, where

$$E^{(n)} = -\sum_{k=1}^{K} t_k^{(n)} ln h_k^{(n)}$$

$$h_k^{(n)} = P(t_k^{(n)} = 1|\boldsymbol{x}^{(n)}) = \frac{exp(x_k^{(n)})}{\sum_{j=1}^{K} exp(x_j^{(n)})}$$

In `SoftmaxCrossEntropyLoss`, you need to implement its forward and backward methods.

NOTE: The definition of `SoftmaxCrossEntropyLoss` layer is a little different from slides, since we don't include trainable parameters θ in the layer. However, the parameters can be explicitly splited out and functioned exactly as one `Linear` layer.

There are two new layers `Conv2D` and `AvgPool2D` in `layers.py`. But the implementations of forward and backward processes are included in `functions.py`. Here are some important descriptions about these classes and functions:

- `Conv2D` describes the convolutional layer which performs convolution on the input using a set of learnable filters. It consists of weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$. $\boldsymbol{W}$ are stored in a 4 dimensional tensor with $n_{out} \times n_{in} \times k \times k$ dimensions, where k specifies the height and width of each filter (also called `kernel_size`), n_{in} denotes the channel numbers of the input which each filter will convolve with, and n_{out} denotes the number of filters.
- `conv2d_forward` implements the convolution operation given the layer's weights, biases and input. For simplicity, we **only** need to implement the standard convolution with `stride` equal to 1. There is another important parameter `pad` which specifies the number of zeros added to each side of **input** (not output). Therefore the expected height of the output should be equal to $H + 2\times$ `pad` $-$ `kernel_size` $+1$ and width likewise.
- `AvgPool2D` describes the pooling layer. For simplicity, we **only** need to implement average pooling operation in non-overlapping style (which means `kernel_size` = `stride`). Therefore the expected height of the output should be equal to $(H + 2\times$ `pad` $)/$ `kernel_size` and width likewise.

Hint: To accelerate convolution and pooling, you should avoid too many nested for loops and instead, use matrix multiplication and numpy, scipy functions as much as possible. To implement convolution with multiple input channels, one possible way is to utilize `conv2` function to perform convolution channel-wise and then sum across channels. To implement pooling operation, one can employ `im2col` function to lay out each pooling area and rearrange them in a matrix. Then perform pooling operation on each column. Theoretically, by using `im2col` properly, one can implement both convolution and pooling operation in the most general form (like convolution with `stride` bigger than 1 and max pooling). However, the back propagation would be very tricky and involve much endeavor to make it work! There are also faster ways to implement the above required convolution and pooling operations, so try to explore and discover them!

Report

Everyone needs to perform the following experiments in this homework:

1. Plot the loss value against to every iteration during training.
2. Construct a CNN with two `Conv2D` + `Relu` + `AvgPool2D` modules and a final `Linear` layer to predict labels using `SoftmaxCrossEntropyLoss`. Compare the difference of results you obtained when working with CNN/MLP (you can discuss the difference from the aspects of training time, convergence, numbers of parameters and accuracy).
3. Try to visualize the first convolution layer's output after rectified linear unit for 4 different digit images. Refer to caffe visualization tutorial [1] for more details.

NOTE: Source codes should not be included in report. Only some essential lines of codes are permitted to be included for explaining complicated thoughts.

NOTE: Any deep learning framework or any other open source codes are **NOT** permitted in this homework. Once discovered, it shall be regarded as plagiarism.

Submission Guideline

You need to submit a report document and codes, as required as follows:

- **Report:** well formatted and readable summary to describe the results, discussions and your analysis. Source codes should *not* be included in the report. Only some essential lines of codes are permitted. The format of a good report can be referred to a top-conference paper.
- **Codes:** organized source code files with README for extra modifications or specific usage. Ensure that TAs can easily reproduce your results following your instructions. **DO NOT include model weights/raw data/compiled objects/unrelated stuff over 50MB (due to the limit of XueTang)**.

You should submit a `.zip` file named after your student number, organized as below:

- `Report.pdf`
- `codes/`
  - `*.py`
  - `README.md`

Deadline

October 16th

TA contact: 王诗瑶, sy-wang14@mails.tsinghua.edu.cn

[1] http://nbviewer.ipython.org/github/BVLC/caffe/blob/master/examples/00-classification.ipynb