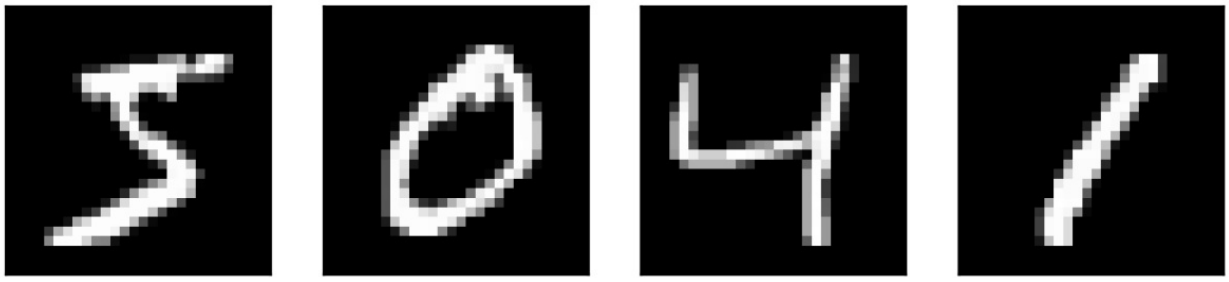
MNIST Digit Classification with MLP

Before Homework

If you are not familiar with Python or NumPy, we provide a tutorial in this folder.

Background

MNIST is a widely used dataset for image classification in machine learning. It contains 60,000 training samples and 10,000 testing samples. Each sample is a 784 × 1 column vector, which originates from a grayscale image with 28 × 28 pixels. Some typical digit images are shown as below.



In this homework, you need to implement multilayer perceptron (MLP) to perform digit classification on MNIST.

Requirements

- python >= 3.6
- numpy >= 1.12.0

Hints

We highly recommend you to use anaconda (or miniconda) to manage your packages (especially if you want to use GPUs in the next homeworks). You can download anaconda here (https://mirrors.tuna.tsinghua.edu.cn/help/anaconda/) .

Dataset Description

To load data, you may use

```
from load_data import load_mnist_2d
train_data, test_data, train_label, test_label = load_mnist_2d('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 corresponding labels are from 0 to 9.

NOTE: during the training process, any information about testing samples should never be introduced.

Python Files Description

In neural networks, each basic operation can be viewed as a functional layer. And a neural network can be constructed by stacking multiple layers to define a certain data processing pipeline. So the neural network implementation is a modular design. Each layer class has four main methods: constructor, forward, backward and update. For some trainable layers with weights and biases, constructor functions as parameter initialization and update method will update parameters via stochastic gradient descent. Forward method represents the data processing performed by the layer and backward performs backpropagation operations. In `layers.py` and `loss.py`, each layer's definition is listed as below.

- `Layer`: base class for all layers
- `Linear`: treat each input as a row vector x and produce an output vector by doing matrix multiplication with weight W and then adding bias b : $u = xW + b$
- `Relu`: linear rectifier activation unit, compute the output as $f(u) = max(0, u)$
- `Sigmoid`: sigmoid activation unit, compute the output as $f(u) = \frac{1}{1+exp(-u)}$
- `Gelu`: gaussian error linear units, compute the output as $f(u) = 0.5u(1 + tanh(\sqrt{\frac{2}{\pi}}(u + 0.044715u^3)))$. You can refer to the original paper for more information. Hint: (1) $tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$; (2) You can double check your gradient result by computing the slope. Specifically, $f'(u)$ should be approximate to $\frac{f(u+\delta) - f(u)}{\delta}$, where δ is a small number (e.g., 1e-5).
- `EuclideanLoss`: compute the squares of the Euclidean norm of differences between inputs $y(n)$ and labels $t(n)$: $\frac{1}{2N}\sum_{n=1}^{N}||t(n) - y(n)||_2^2$, where N denotes the batch size (the number of samples in one mini-batch). The labels in this task are represented in one-hot form.
- `SoftmaxCrossEntropyLoss` : `Softmax` function can map the input to a probability distribution in the following form:

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

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

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

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

- `HingeLoss`: For a classification task, we would like the predicted score corresponding to the correct label to be larger than that corresponding to the wrong label by at least a margin Δ. `HingeLoss` is computed as follows:

$$E = \frac{1}{N}\sum_{n=1}^{N} E^{(n)}$$

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

$$h_k^{(n)} = \begin{cases} 0, & \text{if } k = t_n \\ max(0, \Delta - x_{t_n}^{(n)} + x_k^{(n)}), & \text{otherwise} \end{cases}$$

Where t_n is the correct label (an integer ranging from 1 to K) for the n-th sample. The default value of Δ is 5. You can try tuning Δ and see the performance (e.g., add a softmax layer before the HingeLoss function and set Δ as 0.5).

When running backpropagation algorithm, `grad_output` is an essential variable to compute gradient in each layer. We define `grad_output` as the derivative of loss with respect to layer's output.

NOTE: Since layer definition here is a little different from lecture slides because we explicitly split out activation layers, you should implement backward method in activation layer separately. Hope you realize this.

All files included in the codes:

- `layers.py`: you should implement `Layer`, `Linear`, `Relu`, `Sigmoid`, `Gelu`
- `loss.py`: you should implement `EuclideanLoss`, `SoftmaxCrossEntropyLoss`, `HingeLoss`
- `load_data.py`: load mnist dataset
- `utils.py`: some utility functions
- `network.py`: network class which can be utilized when defining network architecture and performing training
- `solve_net.py`: train_net and test_net functions to help training and testing (running forward, backward, weights update and logging information)
- `run_mlp.py`: the main script for running the whole program. It demonstrates how to simply define a neural network by sequentially adding layers

If you implement layers correctly, just by running `run_mlp.py`, you can obtain lines of logging information and reach a relatively good test accuracy. All the above files are encouraged to be modified to meet personal needs.

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

Report

In the experimental report, you need to answer the following basic questions:

1. Plot the **loss** and **accuracy** curves on the **training set** and the **test set** for **all experiments**. (You can report average loss across several batches for training loss & accuracy because loss for each batch may be too noisy). You should also report the **final value** of **loss** and **accuracy** over the **training set** and the **test set**.

2. Construct a neural network with one hidden layer, and compare the difference of results when using different activation functions (`Relu`/`Sigmoid`/`Gelu`, at least 3 experiments needed and remember controlling variables.), as well as using different loss functions (`EuclideanLoss`, `SoftmaxCrossEntropyLoss`, `HingeLoss`, at least 2 more experiments needed and remember controlling variables.). You can discuss the difference from the perspectives of training time, convergence and accuracy. **NOTE**: "One hidden layer" means that besides the input neurons and output neurons, there are also one layer of hidden neurons.
3. Answer the following question according to the above experiments.

   Comparing `Relu`/`Sigmoid`/`Gelu`, which one is the best choice for activation function (consider the accuracy, convergence, etc.)? Explain why this function has the best performance and others do not. **NOTE**: The answer may vary under different hyper-parameter setups. You only need to ensure that your explanations make sense in your setup.

Bonus (< 2 points):

1. Conduct experiments on a neural network with two hidden layers. Compare the difference of results between one-layer structure and two-layer structure.
2. Tune the hyper-parameters such as the learning rate, the batch size, etc. Analyze how hyper-parameters influence the performance of the MLP. **NOTE**: The analysis is important. You will not get any bonus points if you only conduct extensive experiments but ignore the analysis.
3. Consider the stability of computation when implementing the activation functions and the loss functions.

NOTE: The current hyperparameter settings may not be optimal for good classification performance. Try to adjust them to make test accuracy as high as possible.

NOTE: Any deep learning framework or any other open source codes are **NOT** permitted in this homework. If you use them, you won't get any score from the homework.

NOTE: The accuracy of your best model is required to exceed 95%. If not, TAs believe there must be something wrong with your code. Nevertheless, TAs will still go through your code for any possible bugs even if you reach the requirement.

Code Checking

We introduce a code checking tool this year to avoid plagiarism. You **MUST** submit a file named `summary.txt` along with your code, which contains what you modified and referred to. You should follow the instructions below to generate the file:

1. Fill the codes. Notice you should only modify the codes between `# TODO START` and `# TODO END`, the other changes should be explained in `README.txt`. **DO NOT** change or remove the lines start with `# TODO`.
2. Add references if you use or refer to a online code, or discuss with your classmates. You should add a comment line just after `# TODO START` in the following formats:
   1. If you use a code online: `# Reference: https://github.com/xxxxx`
   2. If you discuss with your classmates: `# Reference: Name: Xiao Ming Student ID:`

`2018xxxxxx`

You can add multiple references if needed.

Warning: You should not copy codes from your classmates, or copy codes directly from the Internet, especially for some codes provided by students who did this homework last year. In all circumstances, you should at least write more than 70% codes. (You should not provide your codes to others or upload them to Github before the course ended.)

警告：作业中不允许复制同学或者网上的代码，特别是往年学生上传的答案。我们每年会略微的修改作业要求，往年的答案极有可能存在错误。一经发现，按照学术不端处理（根据情况报告辅导员或学校）。在任何情况下，你至少应该自己编写70%的代码。在课程结束前，不要将你的代码发送给其他人或者上传到github上。

3. Here is an example of your submitted code:

```
    def forward(self, input):
        # TODO START
        # Reference: https://github.com/xxxxx
        # Reference: Name: Xiao Ming Student ID: 2018xxxxxx
        your codes...
        # TODO END
```

4. At last, run `python ./code_analyze/analyze.py`, the result will be generated at `./code_analyze/summary.txt`. Open it and check if it is reasonable. A possible code checking result can be:

```
########################
# Filled Code
########################
# ..\codes\layers.py:1
    # Reference: https://github.com/xxxxx
    # Reference: Name: Xiao Ming Student ID: 2018xxxxxx
    your codes...

########################
# References
########################
# https://github.com/xxxxx
# Name: Xiao Ming Student ID: 2018xxxxxx

########################
# Other Modifications
########################
# _codes\layers.py -> ..\codes\layers.py
# 8 -         self._saved_tensor = None
# 8 +         self._saved_tensor = None # add some thing
```

Submission Guideline

You need to submit a report document, the codes, and the code checking result, as required as follows:

- **Report:** well formatted and readable summary to describe the network structure and details, experimental settings, results 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. (**Both Chinese and English are permitted.**)
- **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.
- **Code Checking Result**: You should only submit the generated `summary.txt`. **DO NOT** upload any codes under `code_analysis`. However, TAs will regenerate the code checking result to ensure the correctness of the file.

You should submit a `.zip` file named after your student number to Xuetang(网络学堂), organized as below:

- `Report.pdf/md/doc/docx`
- `summary.txt`
- `codes/`
  - `*.py`
  - `README.md/txt` (optional)

Deadline

October 4

TA contact: 顾煜贤，guyx21@mails.tsinghua.edu.cn