Advanced Coding Practice

HW5

MNIST dataset MLP classifier

2023 Fall, CSE4152

Sogang University

Purpose of the HW5

Implement a basic MLP for classifying **images of digits** into **digits**.

Become proficient in designing and training neural networks using PyTorch.

Gain an understanding of the overall MLP learning process.

MNIST Dataset

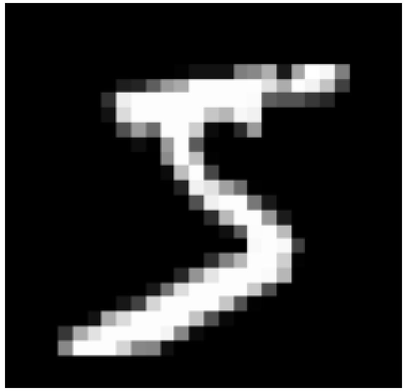
Most famous dataset for digit classification

Goal: Classify handwritten digits 0~9

A total of 70,000 images with labels (60,000 training, 10,000 test)

MNIST dataset

MNIST MLP Classifier



Input : MNIST image data
(28 x 28 vector)

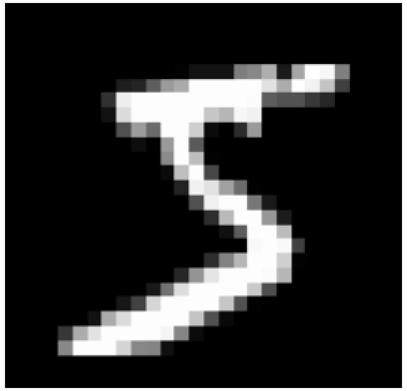
$f_{\mathbf{W}}$

MLP-based Classifier

5

Output : digit (0~9)

MNIST MLP Classifier



Input : MNIST image data
(28 x 28 vector)

$$f_{\mathbf{W}}$$

MLP-based Classifier

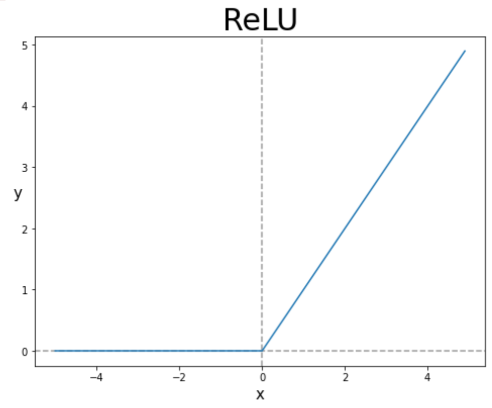
0	0.0
1	0.0
2	0.1
3	0.2
4	0.0
5	**0.6**
6	0.0
7	0.0
8	0.1
9	0.0

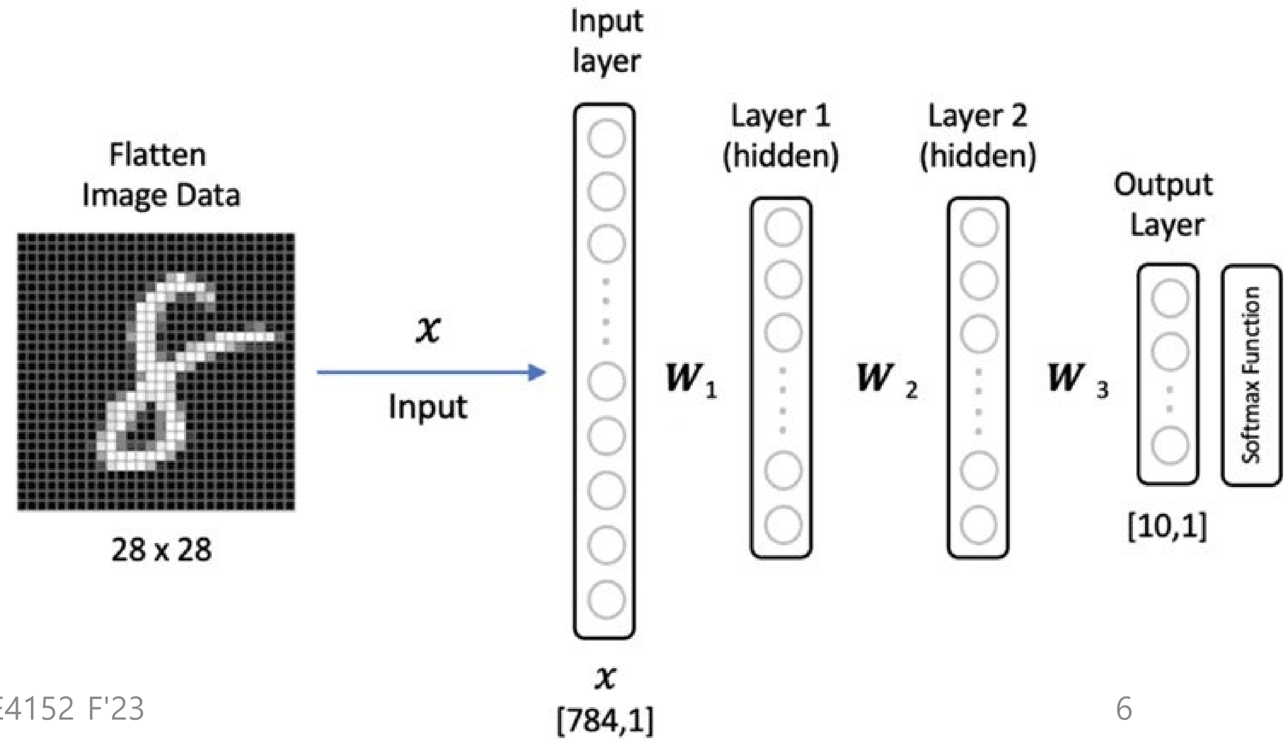
5

Output: Probability of being each digit
(10x1 real-number vector)

MLP-based classifier

- Flatten input 28 x 28 2D image to 1D tensor with 784 elements
- **Hidden layer 1**: input : 784 (28 x 28) output : 50, Act. func.: RELU
- **Hidden layer 2**: input : 50 output : 50, Act. func.: RELU
- **Output Layer**: input : 50 output 10
- Apply a **softmax layer** to the result of the output layer **in order to obtain class probabilities for classification**





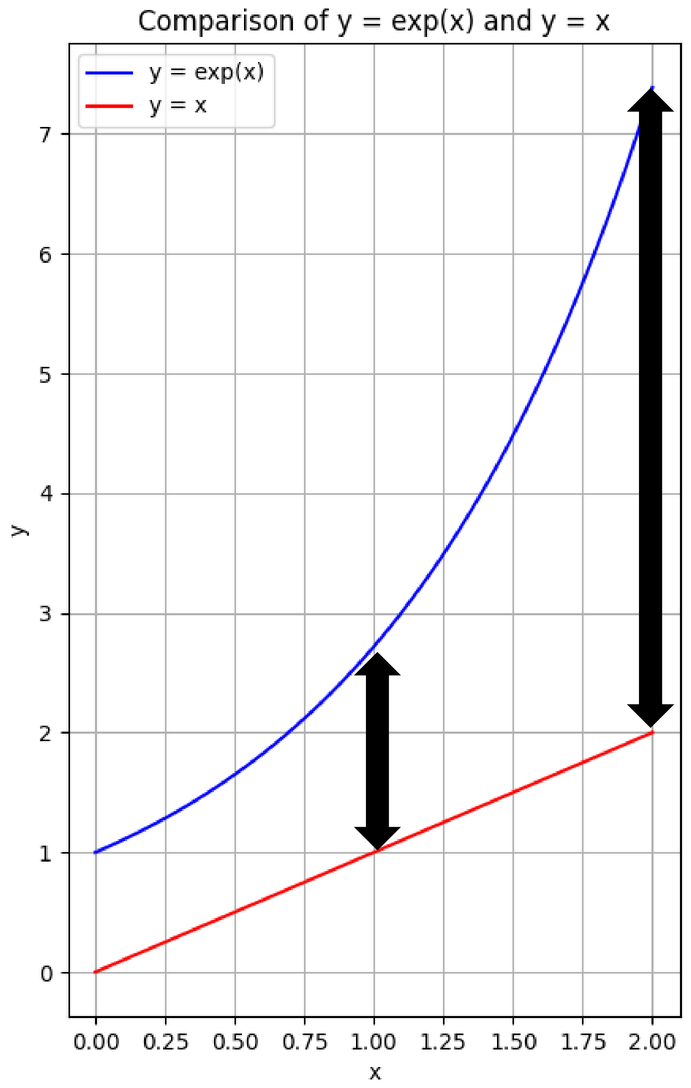
Softmax function

Softmax function

$$S(f_{y_i}) = \frac{e^{f_i}}{\Sigma_j e^{f_j}}$$

The ratio between two values becomes severe after applying softmax.

$$(x, 2x) \underset{\text{softmax}}{\rightarrow} \left(e^x, e^{2x} = \left(e^x\right)^2\right)$$



Softmax function

Softmax 함수로 **두 확률 비율의 차이가 더 커지며, 0과 1사이로 정규화**

Input pixels, $\mathbf{x}$

Shape: (3, 32, 32)

Forward propagation →

Feedforward output, $\mathbf{y}_i$

cat	dog	horse
5	4	2
4	2	8
4	4	1

Shape: (3,)

Softmax function →

Softmax output, $\mathbf{S}(\mathbf{y}_i)$

cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

Shape: (3,)

$$S(f_{y_i}) = \frac{e^{f_i}}{\Sigma_j e^{f_j}} \quad \text{VS} \quad S(f_{y_i}) = \frac{f_i}{\Sigma_j f_i}$$

Cat : $\frac{e^5}{e^5+e^4+e^2}$ =0.71

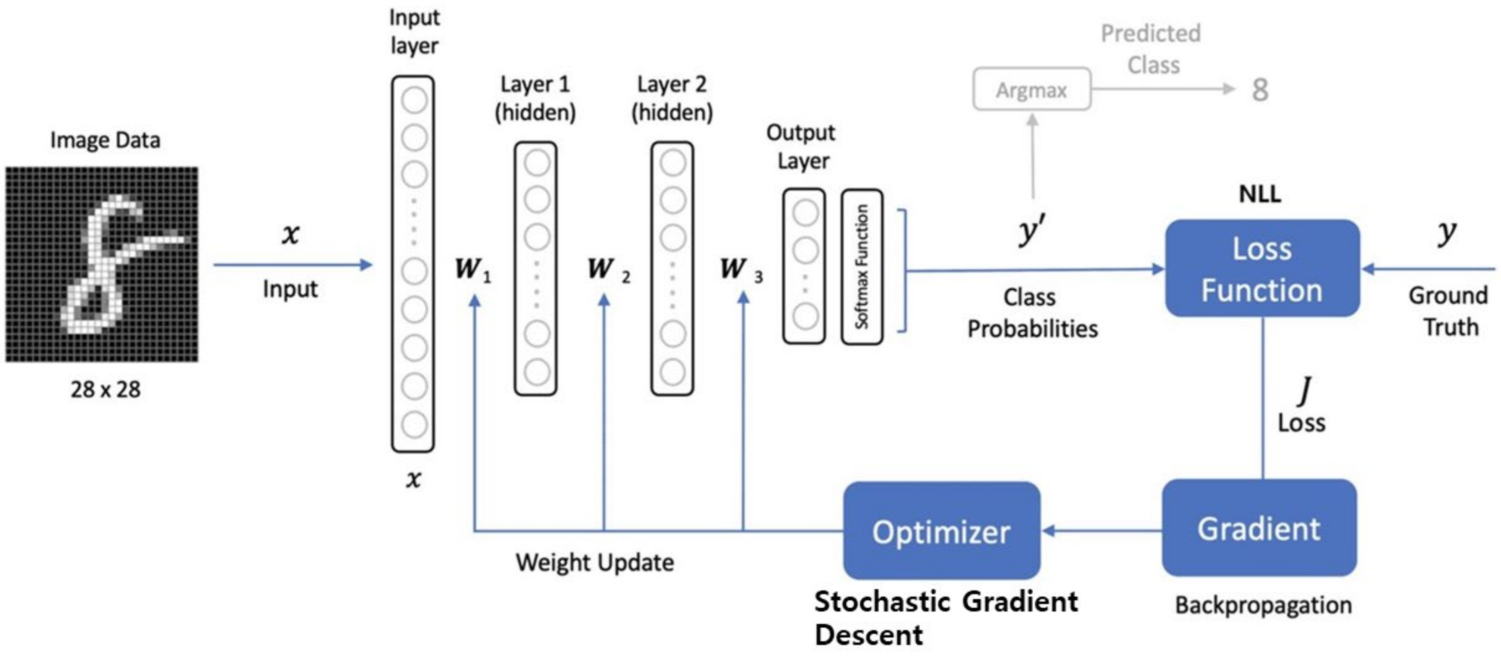
Cat : $\frac{5}{5+4+2}$ =0.45

Cat과 dog의 forward output 차이 : 1 < dog과 horse의 forward output 차이 : 2
Cat과 dog의 softmax output 차이 : 0.45 > dog과 horse의 softmax output 차이 : 0.22
=> 확률이 높은 쪽에 가중치를 주려는 의도

Optimization Framework

Loss function – the error between predicted probabilities of each digit and the ground truth

Optimizer – Stochastic Gradient Descent

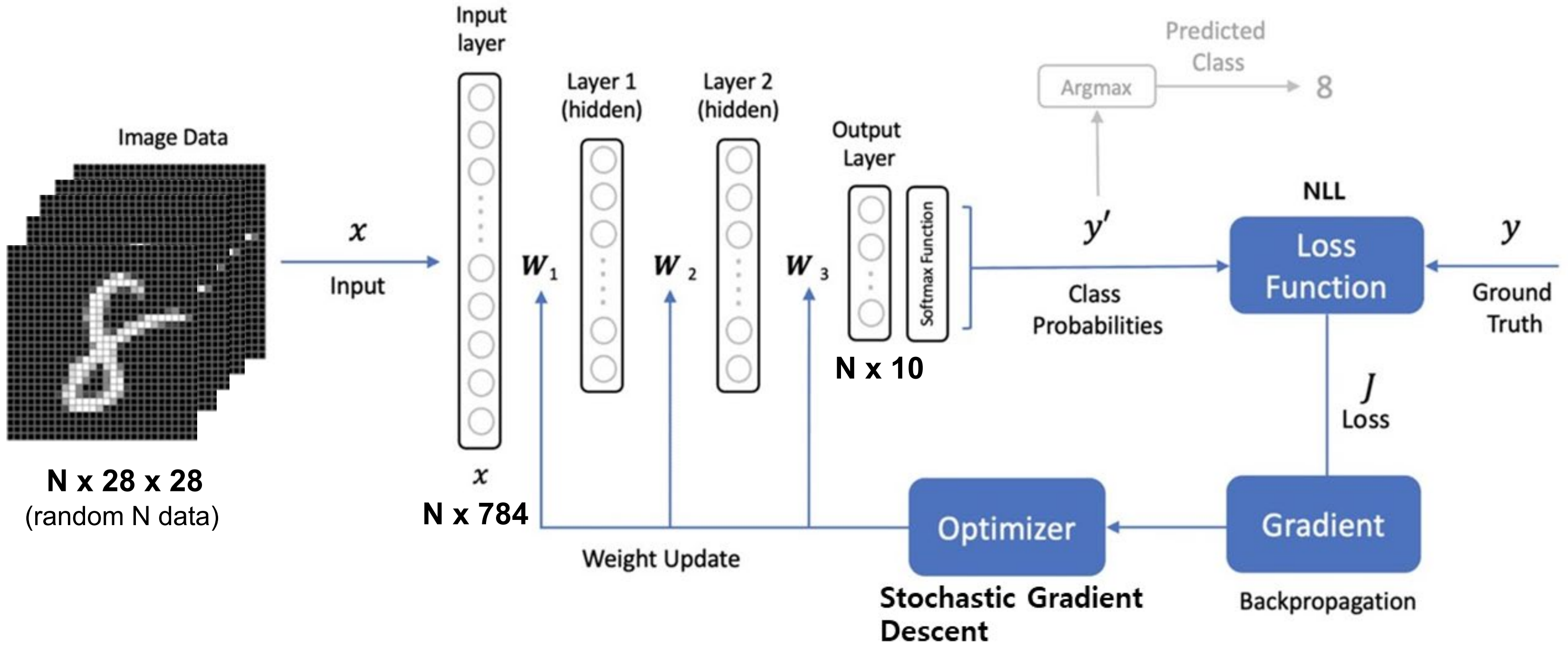


output	GT
0.2	0
0.0	0
0.0	0
0.1	0
0.0	0
0.0	0
0.1	0
0.0	0
0.3	**1**
0.1	0

Loss

Optimization Framework

Batch size : N



0.2	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.3	0.0
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

Output(N x 10)

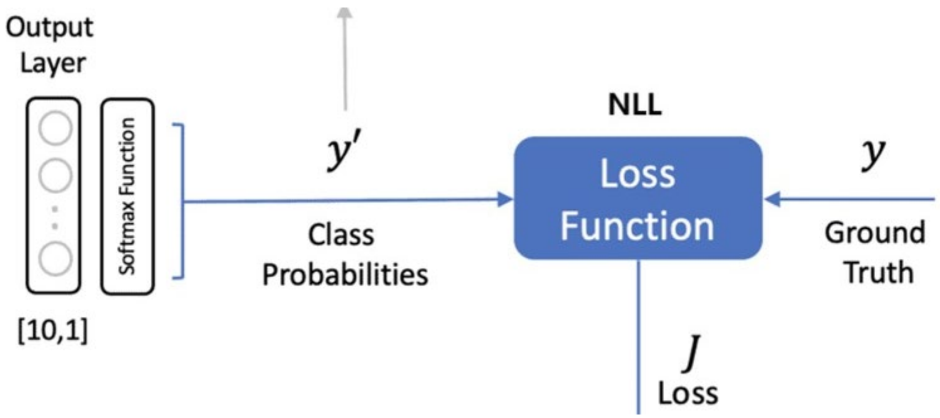
0	0	0	0	0	0	0	0	1	0
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.

GT(N x 10)

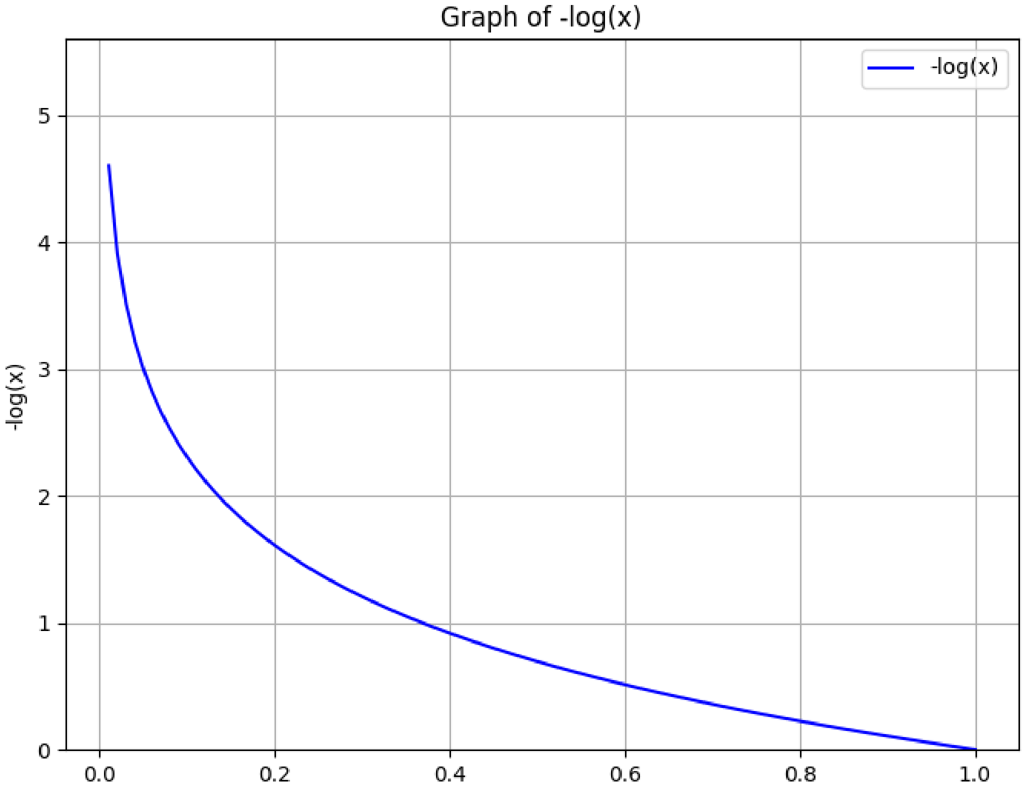
Loss

Negative Log-Likelihood

The gradient makes the probability converge to 1



$$Loss(Y) = -Log(Y)$$



Softmax 연산에서 나온 확률 값(0~1)이 높을 수록 NLL loss 값은 낮다.

Negative Log-Likelihood

Input pixels, $\mathbf{x}$

Softmax output, $\mathbf{S}(\mathbf{y}_i)$

cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

The correct class is highlighted in red

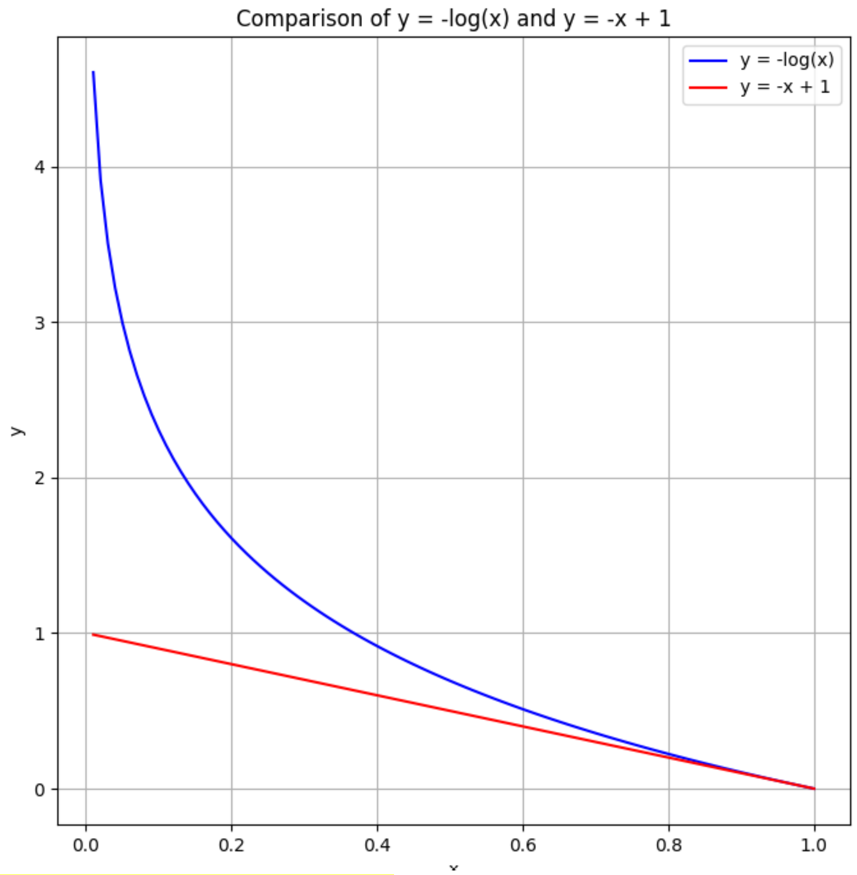
-log(a) at the correct classes

Loss, $\mathbf{L}(a)$

NLL
0.34
0.02
0.71

Total loss :

$$\frac{0.34 + 0.02 + 0.71}{3(batch\ size)} = 0.36$$



-Log(x)를 사용한 이유 :확률이 1과 가까워질 수록 Loss가 작아지는 쪽으로 가중치를 주었다.

Experiment

MNIST Classification

Two main tasks

1) Design a MLP classification model that classifies digit images.

2) Analysis
   - Activation function
   - Learning rate
   - Depth of layers
   - Hidden layer dimension

One file template:

- `mnist.py`

Design a network model

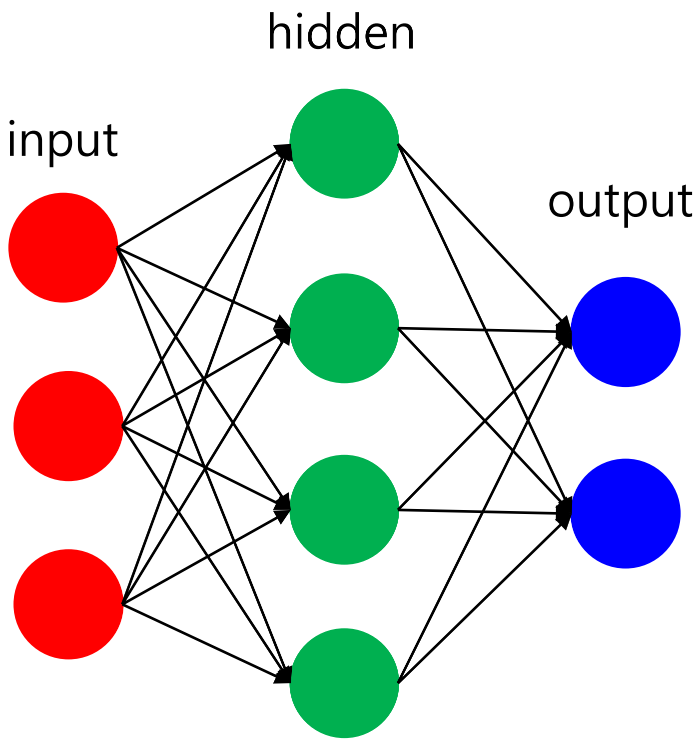
- Template:
  - `mnist.py`
- Modeling a MLP network:
  - Complete a MLP design part(forward function of Net class) in `mnist.py`.
  - The MLP has 2 hidden layers
  - Activation function: ReLU
  - Apply the softmax function to obtain class probabilities for classification
  - Please refer to the documentation of torch layer functions

Network Implementation Example

- CODE

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.fc1 = nn.Linear(3, 4)
        self.fc2 = nn.Linear(4, 2)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        output = self.fc2(x)

        return output
```

- Network



Torch layer functions

- torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)

  [Pytorch document link : torch.nn.linear]

- torch.nn.functional.relu(input, inplace=False)

  [Pytorch document link : torch.nn.functional.relu]

- Tensor.view(*shape) : return a new tensor with the same data as the ‘self’ tensor bit of different shape. **(you can use for flattening image)**

  [Pytorch document link : torch.tensor.view]

- torch.nn.functional.log_softmax(input, dim=None, _stacklevel=3, dtype=None)

  [Pytorch document link : torch.nn.functional.log_softmax]

Why log_softmax not softmax?

Why should we use log_softmax method with NLL loss?

ptrblck Apr '18

Separating log and softmax might lead to numerical instability which is why you should use `log_softmax` as one function.
For `NLLLoss` you need to pass `log_softmax(x)` into the criterion. It you prefer to handle raw logits, you can use `CrossEntropyLoss`, which adds `LogSoftmax` by itself.

✔ Solution

Because of numerical instability, NLLlos doesn’t contain Log computation.
So we should use log_softmax method with NLL

[**Pytorch Discuss Link : Why there is no LOG operator in implementation of torch.nn.NLLLoss**]()

Analysis - Activation function

We've implemented MLP using ReLU as activation function. Explore the different performances when using different activation functions. Report the performance of each activation function.

Analysis – Activation function:

- Use different activation functions instead of ReLU in "forward" method of Class "net"
- i) Use **Tanh**. ii) Use **Sigmoid**. iii) Use **LeakyReLU** with c = 0.01
- Pytorch document : Non-linear activation functions

Analysis – Learning rate

Let’s check how changing the learning rate affects the model’s train and test error.

Find the best learning rate for ReLU and each activation function.

Analysis – Learning rate:

- Plot the performance with respect to a learning rate and find the optimal learning rate

How to change the learning rate

if you want to initialize the learning rate as 0.001 (default: 0.03)

Command : “python mnist.py --lr 0.001”

Analysis – Depth of hidden layers

The current MLP has 2 hidden layers.

Check how the number of hidden layers affects the model's performance. Make sure that each hidden layer in your MLP should have the same number of neurons.

- Analysis – Depth of hidden layers:
  - Define new fully connected layer in "__init__" method of Class Net.
  - Add new fully connected layer in "forward" method of Class Net.
  - Example)

  Current MLP : (784, 50) -> (50, 50) -> (50, 10)

  Add hidden layers : (784, 50) -> (50, 50) -> (50, 50) -> (50, 10)

Analysis - Hidden layer width

Analyze the MLP's behavior with respect to the width of hidden layers

Analysis – Hidden layer width:

- Change the fully connected layer's number of input and output in "__init__" method of Class Net.
- Do not change input layer dimension(784, 1) and output layer dimension(10, 1)

HW5 -submission

Deadline: ~2023.11.10 (FRI) on Cyber campus

Submission:

1) Python code: your implementation
2) Report: Document the analysis of the four factors(activation function, learning rate, hidden layer depth and width) in a report.
3) Agent file: When the 'mnist.py' file runs successfully, the model agent is automatically saved after both training and testing are completed
4) Zip file: include three files above

File format:

1. Python MLP code: MNIST.py
2. Report: CSE4152_학번_HW05.pdf
3. Agent file: mnist_agent.pt
4. Zip file: CSE4152_학번_hw5.zip

If you have any questions, feel free to ask in the cyber campus Q&A or send them to the TA's email(kangpiljae@gmail.com)

End

Model Training

Learning rate Scheduling

Learning rate scheduling is a technique in deep learning that involves dynamically adjusting the learning rate during training.

- Improving Convergence: Learning rate scheduling allows you to start with a larger learning rate in the early stages of training and gradually reduce it as training progresses, thereby improving convergence speed.

- Maintaining Stability: Using a large learning rate at the beginning of training can lead to non-convergence or divergence issues. Learning rate scheduling helps maintain training stability.

Model Training

Learning rate Scheduling

1) Fixed Learning Rate:

It is suitable for simple models or data, but selecting an appropriate initial learning rate is crucial.

2) Exponential Decay:

Learning rate is decreased exponentially over time. It starts with a large learning rate and gradually decreases it as training progresses.

3) Cosine Annealing:

Periodically dropped significantly and then recovered. It involves cycles of reducing and gradually increasing the learning rate.

Data Loader

Torchvision allows easy access to popular datasets like MNIST, CIFAR-10 and ImageNet, simplifying the process of obtaining and working with well-known image datasets for various computer vision tasks.

```
from torchvision import datasets, transforms
```

```
dataset1 = datasets.MNIST('../data', train=True, download=True,
                    transform=transform)
dataset2 = datasets.MNIST('../data', train=False,
                    transform=transform)
train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)
```

Part of Data Loading

Environment Setting

Window
- python 및 pytorch & torchvision 설치
- 참고사이트: https://chancoding.tistory.com/3
 https://pytorch.org/get-started/locally/

Mac
- python 및 pytorch & torchvision 설치
- 참고사이트: https://leettle.tistory.com/2
https://pytorch.org/get-started/locally/

GPU 없는 경우 CPU 버전으로 다운로드