

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN



NHẬP MÔN CÔNG NGHỆ THÔNG TIN

Đồ Án

MÔ HÌNH NHẬN DIỆN CHỮ SỐ

Nhóm thực hiện: Nhóm 4

Thành phố Hồ Chí Minh, 12/2025

Mục lục

1 Dataloader	1
1.1 MNIST Dataset	1
1.2 Image Loader	1
1.3 Xây dựng dataSet	3
1.4 Xây dựng DataLoader	4
2 Kiến trúc mô hình CNN(Convolutional Neural Network)	5
2.1 Lý do chọn mô hình CNN(Convolutional Neural Network)	5
2.2 Phân tích cấu kiến trúc mô hình (<code>model.py</code>)	5
2.2.1 Giai đoạn 1: Trích xuất đặc trưng (Feature Extraction)	5
2.2.2 Giai đoạn 2: Phân loại (Classification)	6



1 Dataloader

1.1 MNIST Dataset

MNIST là một database cực kỳ lớn và nổi tiếng trong deeplearning, chứa các dữ liệu hình ảnh chữ số viết tay. Trong dự án này, ta sẽ sử dụng nó như là một nguồn dataset chính để huấn luyện mô hình.

Trong thư viện **PyTorch**, thì đã có sẵn `torchvision.datasets.MNIST` là một class dùng để tải về tập dữ liệu được sử dụng để huấn luyện mô hình.

1.2 Image Loader

Trong `dataloader.py`, ta khai báo và định nghĩa `class MnistDataset` là một lớp dùng để đồng thời tải dataset và gọi dataset ra.

Trong class `MnistDataset`, các thuộc tính được khởi tạo(`__init__`) được gồm: `root_dir` (Ta pass đường dẫn tới thư mục chứa dataset), `transform` và cuối cùng là `mnist_dataset`.

Bên cạnh các thuộc tính, các phương thức hỗ trợ như:

- `__len__(self)`: Xem số lượng hình ảnh trong 1 dataset)
- `__getitem__(self, index)`: Lấy hình ảnh ở vị trí `index` trong `self.mnist_dataset` và chuyển nó về định dạng thông qua `transform`.

Tức là, thuộc tính `transform` được khai báo, khi gọi class `MnistDataset`, ta chỉ cần pass vào `self.transform` bằng transform phù hợp để chuyển hình ảnh sang dạng dữ liệu phù hợp với huấn luyện mô hình.

Logic hoạt động của `MnistDataset`: Raw Image(PIL) $\xrightarrow{\text{Transform}}$ Tensor $\xrightarrow{\text{One-Hot}}$ Vector 10 chiều



Algorithm 1 Định nghĩa của class MnistDataset trong dataloader.py

```
1: Class MnistDataset(Dataset):
2:
3: function __INIT__(root_dir, train, transform)
4:   Input:
5:     root_dir: Đường dẫn thư mục chứa dữ liệu
6:     train = True
7:     transform: Các phép biến đổi ảnh cần áp dụng
8:
9:   Process:
10:    self.root_dir ← root_dir
11:    self.train ← train
12:    self.transform ← transform
13:    Tải dữ liệu MNIST gốc từ torchvision.datasets:
14:    self.mnist_data ← MNIST(root = root_dir, train = train, download = True,
    transform = None)
15:    *Lưu ý: transform=None để lấy dữ liệu thô, ta tự xử lý sau.
16: end function
17:
18: function __LEN__
19:   return Số lượng mẫu trong self.mnist_data
20: end function
21:
22: function __GETITEM__(index)
23:   Input: index (vị trí mẫu dữ liệu cần lấy)
24:
25:   Step 1: Lấy dữ liệu thô
26:   image, label ← self.mnist_data[index]
27:   (image là PIL Image, label là số nguyên)
28:
29:   Step 2: Xử lý ảnh (Image Transform)
30:   If self.transform is not None Then
31:     image ← self.transform(image)
32:   End If
33:
34:   Step 3: Xử lý nhãn (One-Hot Encoding)
35:   label_tensor ← chuyển label sang Tensor
36:   label_onehot ← OneHot(label_tensor, num_classes=10)
37:   label_onehot ← chuyển sang kiểu float
38:   (Ví dụ: 5 → [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
39:
40:   return (image, label_onehot)
41: end function
```

1.3 Xây dựng dataSet

Ta xây dựng sẽ chia tập dataset chính (`full_dataset`) thành ba tập chính: **train**, **validation**, **test** với tỉ lệ tương ứng **6:2:2**.

Lý do ta chia thành ba tập như vậy là để tránh hiện tượng **overfitting**. Giải thích đơn giản là hiện tượng mà mô hình nhận diện rất tốt trên dataset mà nó được train nhưng lại nhận diện rất kém trên dataset mà nó chưa từng thấy.

Trong thư viện **PyTorch**, hàm `random_split` giúp ta chia nhỏ chia nguồn dataset thành các dataset con với tỉ lệ tùy ý. Thêm vào đó khi hàm hoạt động, dữ liệu sẽ được chia nhỏ ngẫu nhiên, với các dữ liệu bên trong mỗi tập dataset con là đôi một khác nhau.

Trong `dataloader.py`, hàm `getDataSet(root_dir)` nhận đầu vào đường dẫn thư mục, khởi tạo nguồn dataset, sau đó là chia nguồn thành: `train_dataset`, `validation_dataset`, `test_dataset`, lần lượt là các loại dataset và size của dataset nguồn.

Dưới đây là mã nguồn của hàm `getDataSet`:

```
1  #dataloader.py
2  def getDataSet(root_dir) -> tuple:
3      transform = transforms.Compose([transforms.ToTensor()])
4      full_dataset= MnistDataset(root_dir, True, transform) #Initialize mother dataset
5      total_size = len(full_dataset) #Full size: 100%
6      train_size = int(0.6 * total_size) #Train size: 60%
7      val_size = int(0.2 * total_size) #Validation size: 20%
8      test_size = total_size - train_size - val_size #Test size = 100 - 60 - 20 = 20%
9
10     train_dataset, val_dataset, test_dataset = random_split( #Splitting full_data set
11         into
12         full_dataset,
13         [train_size, val_size, test_size],
14         generator=torch.Generator().manual_seed(42)
15     )
16     return train_dataset, val_dataset, test_dataset, total_size #Return a tuple
17         containing three different datasets and total size of the mother set
18         respectively.
```

Chú ý:

- Từ dòng **9-13**, `full_dataset` được chia nhỏ thành ba tập chính với tỉ lệ **6:2:2**
- Thông số `generator` với `manual_seed(42)` sẽ giúp giữ nguyên vị trí dataset của dữ liệu. (Ví dụ, hình #5 nằm trong tập **train** ở lần chạy 1, thì các lần chạy tiếp theo hình #5 vẫn sẽ nằm trong tập **train**).

1.4 Xây dựng DataLoader

Sau khi đã xây dựng xong **dataset**, ta sẽ đến phần xây dựng data loader để nạp dữ liệu cho việc huấn luyện model.

Ta sử dụng `torch.utils.data.Dataloader` để hình thành loader cho từng tập dataset.

```
1  #trainer.py
2  from torch.utils.data import DataLoader
3  from src.data.dataloader import getDataSet, DataCollator()
4  from omegaconf import OmegaConf
5  config = OmegaConf.load("configs/config.yaml") #Settings
6  collator = DataCollator()
7  train_dataset, val_dataset, test_dataset, total_size =
    getDataSet(config.data.root_dir) # Get datasets
8
9  train_loader = DataLoader(
10     dataset=train_dataset,
11     batch_size=config.data.batch_size,
12     shuffle=True,
13     num_workers=4,
14     collate_fn=collator)
15
16     test_loader = DataLoader(...) #Same, but shuffle = False
17     val_loader = DataLoader(...) #Same as test_loader
```

2 Kiến trúc mô hình CNN(Convolutional Neural Network)

2.1 Lý do chọn mô hình CNN(Convolutional Neural Network)

Ban đầu, chúng em tính làm theo cấu trúc MLP. Tuy nhiên, sau khi tham khảo một số nguồn thì chúng em đã rút ra được một số lợi thế mà CNN vượt trội hơn nhiều so với FNN:

1. Tiết kiệm thời gian tính toán
2. Mô hình CNN có độ chính xác cao hơn MLP trong mảng xử lý dữ liệu hình ảnh

2.2 Phân tích cấu kiến trúc mô hình (model.py)

Mô hình **DigitsClassifier** được xây dựng dựa trên kiến trúc Mạng Nơ-ron Tích chập (CNN) tiêu chuẩn, tối ưu cho bài toán nhận diện ảnh kích thước nhỏ như MNIST (28×28). Kiến trúc được chia thành hai giai đoạn chính:

2.2.1 Giai đoạn 1: Trích xuất đặc trưng (Feature Extraction)

Khối này chịu trách nhiệm học các họa tiết từ đơn giản đến phức tạp thông qua các lớp tích chập và pooling.

- **Lớp Tích chập 1 (conv1):**

- **Input:** Ảnh đơn kênh (Grayscale), kích thước 28×28 .
- **Cấu hình:** $in_channels = 1, out_channels = 32, kernel = 3, padding = 1$.
- **Cơ chế:** Sử dụng $padding = 1$ giúp bảo toàn kích thước không gian:

$$Output_Size = \lfloor \frac{28 + 2(1) - 3}{1} + 1 \rfloor = 28$$

- **Activation:** ReLU giúp khử tuyến tính.
- **Pooling:** MaxPool2d(2, 2) giảm kích thước đi một nửa ($28 \rightarrow 14$) [cite: 12].
- **Output Tensor:** ($Batch_Size, 32, 14, 14$).

- **Lớp Tích chập 2 (conv2):**

- **Input:** Output của lớp trước (32 kênh).



- **Cấu hình:** $in_channels = 32, out_channels = 64$. Tăng số lượng bộ lọc để bắt đặc trưng mức cao.
- **Pooling:** Giảm kích thước không gian ($14 \rightarrow 7$).
- **Output Tensor:** $(Batch_Size, 64, 7, 7)$.

2.2.2 Giai đoạn 2: Phân loại (Classification)

Dữ liệu 2D được duỗi phẳng để đưa vào mạng nơ-ron truyền thống (Fully Connected).

1. **Duỗi phẳng (Flatten):** Biến đổi Tensor $(64, 7, 7)$ thành vector. Kích thước vector đặc trưng:

$$64 \times 7 \times 7 = 3136 \text{ chiều}$$

Lệnh thực thi: `x.view(-1, self.flatten_size)`

2. **Lớp Kết nối đầy đủ (fc1):**

- Biến đổi từ $3136 \rightarrow 128$ chiều.
- Kết hợp hàm kích hoạt ReLU.

3. **Dropout:**

- Tỷ lệ: 0.25 (25%).
- Mục đích: Ngẫu nhiên tắt nơ-ron để ngăn chặn Overfitting.

4. **Output (fc2):**

- Ánh xạ từ $128 \rightarrow num_classes$ (10 lớp cho các chữ số 0-9).

Tài liệu tham khảo