M2 Data Science: Health, Insurance, Finance

Semi-Supervised and Self-Supervised Learning with SimCLR on MNIST data with only 100 Labels

Deep Learning Project Report

16th January 2025

Students:
Richard CHEAM
Loïc XU
Janikson GARCIA BRITO

Lecturer:
Prof. Blaise Hanczar

Contents

1 Introduction

In recent years, image classification has been a center of focus in the field of deep learning, With numerous innovative approaches in this domain, powerful tools have been developed that aid people in various professional, academic, and personal contexts such as medical imaging, autonomous vehicles, and facial recognition systems.

Hence, in this project, we will explore the application of several neural network models such as multilayer perceptron (MLP), convolutional neural network (CNN), and notably the Simple Framework for Contrastive Learning of Visual Representations (SimCLR) [1] to address the handwritten digits classification task on the MNIST dataset.

Nevertheless, the challenge lies in the assumption that only 100 labeled images from the dataset are available, which reflects a common scenario in real-world settings.

2 MNIST dataset

The MNIST dataset (Modified National Institute of Standards and Technology) is a widely used benchmark dataset in machine learning and deep learning. It consists of 70,000 grayscale images of handwritten digits (0–9), each of size 28x28 pixels. The dataset is split into 60,000 training images and 10,000 test images.

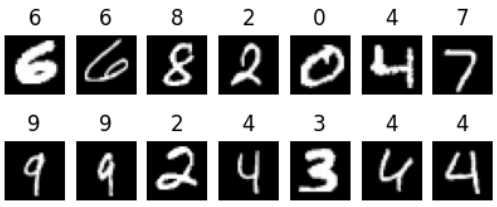


Figure 1: Examples of labeled MNIST dataset images

3 Methodology

For semi-supervised learning, we will focus on two network architectures: MLP and CNN. Our goal is to build a single robust network using data preprocessing techniques, particularly data augmentation, along with intuition and inspiration from external sources. First, we randomly sampled 100 images from the MNIST training dataset, 10 images per digit (see subsection 4.1). Then, to address the issue of insufficient labeled data, we will augment these images in various ways (see subsection 4.2). The trained model will be evaluated on a validation set and subsequently tested on

the MNIST test dataset. For self-supervised learning case with SimCLR, we will follow the method described in the paper [1] (see subsection 5.2). Finally, results of semi-supervised learning using baseline neural networks and self-supervised learning will be compared to evaluate their performance in handwritten digit recognition on the MNIST dataset, aiming to gain valuable insights from both approaches.

4 Semi-Supervised Learning Baseline Models

4.1 Data Preparation

100 labeled images were chosen randomly from the original MNIST train dataset (10 images per digit), which means 59,900 were unlabeled. Then, these images were split into a train set (80 images) and a validation set (20 images) with stratified proportions of 10 digits or classes. Furthermore, the data was also transformed to tensor (turned the input data pixel in the range of [0,255] to a 3-dimensional Tensor and scaled it to the range of [0,1]) and normalized by its mean and standard deviation (0.1307 and 0.3081 respectively, which is equivalent to normalizing zero mean and unit standard deviation).

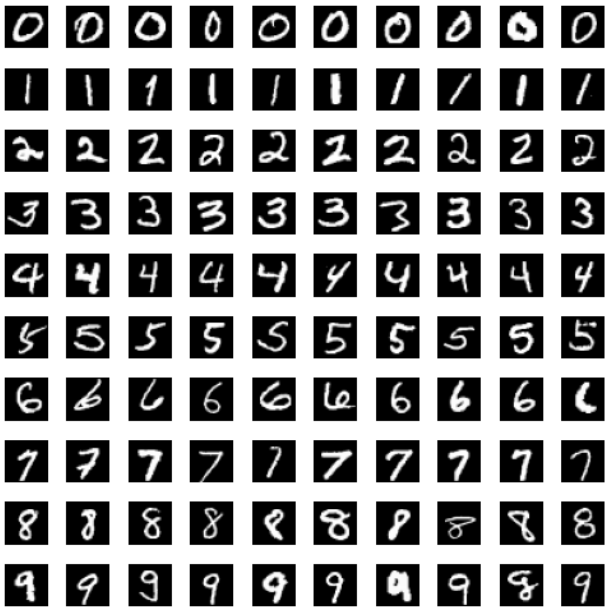


Figure 2: 100 digits chosen from MNIST training dataset

4.2 Data Augmentation

Since the networks, particularly the CNNs could work better with sufficient amount of data. Therefore, in order to deal with inadequate amount of labeled images,

the data augmentation method was considered to make the networks robust. Thus, several transformations were applied to increase the number of labeled data such as random rotation, random resize crop, translations, scale and shear.

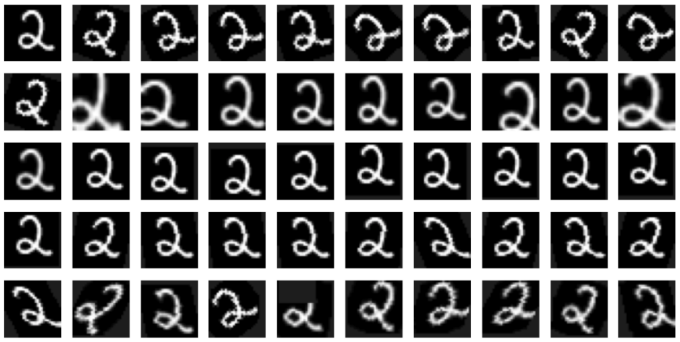


Figure 3: Some examples of augmented data, digit 2

4.3 Baseline Models

Layer (type)	Output Shape	Param #
Linear-1	[-1, 512]	401,920
BatchNorm1d-2	[-1, 512]	1,024
Linear-3	[-1, 128]	65,664
BatchNorm1d-4	[-1, 128]	256
Linear-5	[-1, 64]	8,256
BatchNorm1d-6	[-1, 64]	128
Linear-7	[-1, 10]	650

Total params: 477,898
Trainable params: 477,898
Non-trainable params: 0

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 28, 28]	320
BatchNorm2d-2	[-1, 32, 28, 28]	64
MaxPool2d-3	[-1, 32, 14, 14]	0
Conv2d-4	[-1, 64, 12, 12]	18,496
BatchNorm2d-5	[-1, 64, 12, 12]	128
MaxPool2d-6	[-1, 64, 6, 6]	0
Conv2d-7	[-1, 128, 4, 4]	73,856
BatchNorm2d-8	[-1, 128, 4, 4]	256
MaxPool2d-9	[-1, 128, 2, 2]	0
Linear-10	[-1, 192]	98,496
BatchNorm1d-11	[-1, 192]	384
Linear-12	[-1, 64]	12,352
BatchNorm1d-13	[-1, 64]	128
Linear-14	[-1, 10]	650

Total params: 205,130
Trainable params: 205,130
Non-trainable params: 0

Figure 4: Summary of MLP and CNN models

For Multi-Layer Perceptron (MLP), it consists of multiple fully connected layers with batch normalization applied after most layers as described belows:

- Linear and BatchNorm1d Layers: The network has four fully connected layers (512, 128, 64, and 10 units), with batch normalization following all layers except the final output layer. The batch normalization helps stabilize training and accelerate convergence.
- Output Layer: The final layer has 10 units, indicating a classification task with 10 classes.
- Parameter Summary: The model has a total of 477,898 trainable parameters, making it more complex compared to the CNN due to the larger number of dense connections typical of MLPs.

Whereas, for Convolutional Neural Network (CNN), it consists of multiple convolutional layers followed by batch normalization, max-pooling layers, and fully connected layers for final classification.

- Conv2d and BatchNorm2d Layers: The model starts with three sets of convolutional layers followed by batch normalization and max pooling. These layers extract features from the input images and reduce spatial dimensions while maintaining essential patterns.
- Fully Connected Layers: The model concludes with three linear (dense) layers, with batch normalization applied after the first two fully connected layers. These layers combine the extracted features for the final classification task.
- Parameter Summary: The model has a total of 205,130 trainable parameters, indicating a moderately complex network suitable for small to medium-scale image datasets.

4.4 Baseline Network Results

4.4.1 Training and Validation

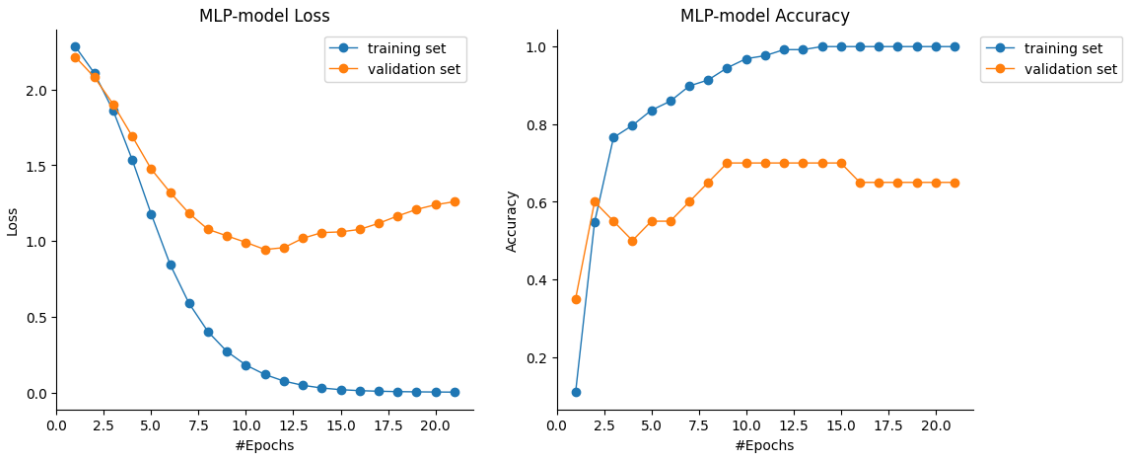


Figure 5: MLP loss and accuracy

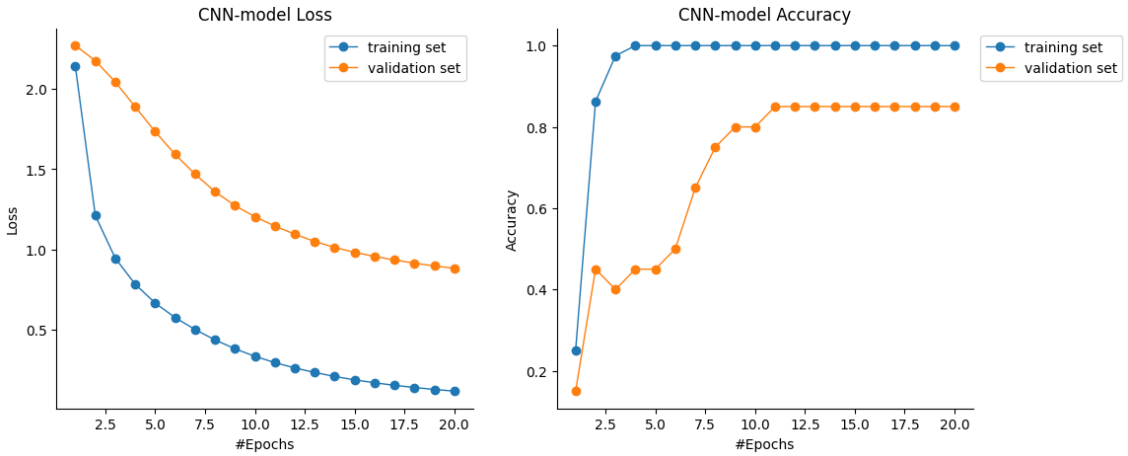


Figure 6: CNN loss and accuracy

As observed in the graphs, both models exhibit overfitting behavior. The training accuracy approaches nearly 100%, while the validation accuracy stagnates at around 70% for the MLP model and slightly above 80% for the CNN model. This discrepancy indicates that the models have learned the training data too well, capturing noise and specific patterns that do not generalize to unseen data.

4.4.2 Evaluation

Model	**Accuracy**
MLP	0.7021
CNN	0.8369

Table 1: Accuracy of MLP and CNN models on the test set

The table above summarizes the accuracy of the MLP and CNN models on the test set. It's important to note that these results were achieved using only 100 labeled samples from the MNIST dataset, with 80 labels for training and 20 labels for validation. Given the extremely limited amount of labeled data, achieving 70% accuracy on the validation set for the MLP model and 80% for the CNN model is a promising outcome. These results highlight the potential of the models to capture useful patterns in the data despite the significant constraint on labeled examples.

5 A Simple Framework for Contrastive Learning of Visual Representations

Contrastive Learning is a technique used to learn representations (images) by comparing data points. The goal is to make representations of similar data points close to each other and representations of dissimilar data points far apart. SimCLR [1] is a simplified version of contrastive self-supervised learning algorithms without requiring specialized architectures or a memory bank to store past representations for comparison.

5.1 SimCLR's Architecture

SimCLR [1] learns representations by maximizing agreement between differently augmented views of the same data example via a contrastive loss in the latent space. It consists of four essential components (see Figure 7):

- A data augmentation that transforms a given data (x) into two correlated views, which considered as a positive pair, noted $\tilde{x}_i$ and $\tilde{x}_j$. Three simple augmentations: random cropping followed by resize back to the original size, random color distortions, and random Gaussian blur.
- A neural network base encoder $f(.)$ that extracts representation vectors from augmented data. The framework allows various choices of the network architecture without any constraints. The result obtained is then noted as h.
- A small neural network projection head $g(.)$ that maps representations to the space where contrastive loss is applied. Then, a MLP with one hidden layer was utilized to get $z = g(h) = W^{(2)}\sigma(W^{(1)}h)$ where σ is a ReLU nonlinearity, $W^{(1)}$ and $W^{(2)}$ are the weight matrices of the hidden and output layer respectively in the MLP.
- A contrastive loss function defined for a contrastive prediction task. Given a set $\{\tilde{x}_k\}$ including a positive pair $\tilde{x}_i$ and $\tilde{x}_j$, the contrastive prediction task aims to identify $\tilde{x}_j$ in set $\{\tilde{x}_k\}_{k\neq i}$ for a given $\tilde{x}_i$. The NT-Xent (Normalized Temperature-Scaled Cross-Entropy Loss) is used to bring positive pairs (views of the same image) closer together in the representation space while pushing apart negative pairs (views of different images). Thus, the loss function for a positive pair of examples (i, j) is defined as

$$l_{i,j} = -\texttt{log}\frac{\texttt{exp}(\texttt{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} 1_{[k\neq i]}\texttt{exp}(\texttt{sim}(z_i, z_k)/\tau)}$$

where $\texttt{sim}(z_i, z_j) = \frac{z_i^T z_j}{||z_i|| ||z_j||}$ is pairwise similarity, $1_{[k \neq 1]}$ is an indicator function evaluating to 1 iff $k \neq i$ and τ is a temperature parameter which is a scaling factor.

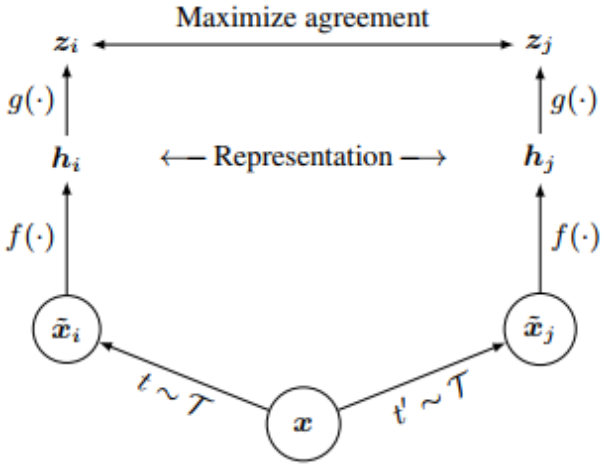


Figure 7: SimCLR's framework representation (Chen20 [1])

Moreover, SimCLR [1] trains with a large batch size N, which varies from 256 to 8192. For instance, if N = 8192, we start with 8192 images in a single batch; each image is augmented twice, creating 2 views per image, which is $2 \times 8192 = 16384$. As a result, it results in 16382 negative examples per positive pair because for a given image, all other views are negative examples except itself and itself and its counterpart that forms a positive pair; precisely, 16384 - 2 = 16382 negative examples per positive pair.

5.2 Model Setup and Training

For our implementation of SimCLR, we followed the methodology outlined in the original paper. The process began with data augmentation techniques, including random cropping, random color distortions, and random Gaussian blur. These augmentations were applied to generate different views of the same image, a crucial step in the SimCLR framework.
The architecture consisted of:

- **Encoder:** A ResNet-based architecture, following the paper's recommendation, to extract meaningful representations from the input images.
- **Projection Head:** A Multi-Layer Perceptron (MLP) to map the encoded representations to a lower-dimensional space suitable for contrastive learning.
- **Loss Function:** The NT-Xent (Normalized Temperature-scaled Cross-Entropy) loss function

After training the SimCLR framework, we utilized the trained encoder for classification tasks. For this, we added a classifier to the encoder. The classifier was implemented as an MLP with an output layer of 10 output features, corresponding to the 10 classes (0 to 9) in the dataset.

The model was trained on 80 labeled samples from the MNIST dataset, reserving 20 samples for the validation set to evaluate and select the best model.

To improve the model, we conducted several experiments:

- **Data Augmentation:**
  - Added random flipping.
  - Removed some augmentations to evaluate their individual impact on performance.
- **Encoder Architecture:**
  - Replaced the ResNet encoder with a basic CNN.
  - Experimented with more complex CNN architectures.
- **Projection Head:**
  - Adjusted the number of neurons and layers in the projection head to test its effect on the quality of representations.

We also performed hyperparameter optimization to optimize the model:

- **Optimizer:** Switched from ADAM to LARS for the encoder, as recommended in the paper for better convergence in contrastive learning setups.
- **Batch Size:** Increased the batch size, following the paper's observation that larger batch sizes improve the quality of learned representations.
- **Temperature Parameter:** Experimented with different temperature values in the NT-Xent loss function to observe their effect on model performance.

6 Result and Comparison

Encoder	Batch Size	Data Augmentation	Val/Test Accuracy
ResNet	128	Random Crop, Color Distortions, Blur	0.65 | 0.77
ResNet	256	Random Crop, Color Distortions, Blur	0.65 | 0.78
ResNet	512	Random Crop, Color Distortions, Blur	0.50 | 0.37
Basic CNN	128	Random Crop, Color Distortions, Blur	0.65 | 0.71
Basic CNN	512	Random Crop, Color Distortions, Blur	0.50 | 0.53
ResNet	128	Random Crop, Random Flip	0.70 | 0.81
ResNet	256	Random Crop, Random Flip	0.65 | 0.78
ResNet	512	Random Crop, Random Flip	0.50 | 0.50
Basic CNN	128	Random Crop, Random Flip	0.65 | 0.80
Basic CNN	512	Random Crop, Random Flip	0.50 | 0.61
Complex CNN	128	Random Crop, Random Flip	**1 | 0.88**
Complex CNN	512	Random Crop, Random Flip	0.95 | 0.81

Table 2: Comparison of models with different encoders, batch sizes, and data augmentation strategies.

The best setup we found out was :

The data augmentation

- Random Horizontal Flip: Randomly flips the image horizontally.
- Random Resized Crop: Crops the image to a random size and aspect ratio, with a scale parameter set to $(0.2, 1.0)$.
- Normalization: Normalizes the image with mean (0.1307) and standard deviation (0.3081).

The encoder architecture, referred to as `EncoderImproved`

- Layer 1:
  - Convolutional layer with 1 input channel, 64 output channels, kernel size 3, stride 1, and padding 1.
  - Batch Normalization.
  - ReLU activation.
  - MaxPooling with kernel size 2 and stride 2.
  - Dropout with $p = 0.3$.

- Layer 2:
  - Convolutional layer with 64 input channels, 128 output channels, kernel size 3, stride 1, and padding 1.
  - Batch Normalization.
  - ReLU activation.
  - MaxPooling with kernel size 2 and stride 2.
  - Dropout with $p = 0.3$.
- Layer 3:
  - Convolutional layer with 128 input channels, 256 output channels, kernel size 3, stride 1, and padding 1.
  - Batch Normalization.
  - ReLU activation.
  - MaxPooling with kernel size 2, stride 2, and padding 1.
  - Dropout with $p = 0.3$.
- Flattening: The output is flattened to a vector for further processing.

The projection head

- Input Layer: A fully connected layer with 4096 input features and 128 output features.
- Activation Function: ReLU.
- Output Layer: A fully connected layer with 128 input features and 64 output features.

The batch size used during training was 128.
The temperature of the loss function was 0.5.

The classifier

- Input Layer: A fully connected layer with 4096 input features and 64 output features.
- Activation Function: ReLU.
- Output Layer: A fully connected layer with 64 input features and 10 output features, corresponding to the 10 classes (0 to 9) in the dataset.

It feels strange that every improvement suggested in the paper actually lowers the accuracy in our project. One possible reason could be that the MNIST dataset consists of very "simple" grayscale images, and applying complex data augmentations like color distortion and Gaussian blur might not be the best fit for this type of data. These augmentations might be adding unnecessary noise instead of helping the model generalize. Also, using a ResNet architecture might be overkill for MNIST, as it's designed for much more complex datasets. This mismatch between the dataset and the methods we're using could explain why we're seeing worse performance.

7 Conclusion

Our implementation and experiments demonstrated that even with a limited dataset of 100 labeled samples, the SimCLR framework could learn meaningful representations, achieving promising results on the classification task. We got an improvement of approximately 5% of accuracy compared to the baseline CNN model. Further improvements could involve exploring other architecture of encoder and classifier.

References

[1] Chen, Ting & Kornblith, Simon & Norouzi, Mohammad & Hinton, Geoffrey. (2020). A Simple Framework for Contrastive Learning of Visual Representations. 10.48550/arXiv.2002.05709.

A Annex

A.1 Code and Output

Deep Learning Project

**Partner: Richard CHEAM, Loïc XU, Janikson GARCIA BRITO **

Import libraries

```
##### base #####
import matplotlib.pyplot as plt
import numpy as np
import random
from collections import Counter
from tqdm import tqdm
##### sklearn #####
from sklearn.model_selection import train_test_split
##### torch ######
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
from torch.utils.data import DataLoader, Subset, ConcatDataset, Dataset, TensorDataset
import torch.nn.functional as F
import torch.optim as optim
from torchsummary import summary
```

Setup

```
torch.cuda.is_available()
```

```
True
```

```
seed = 42
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
np.random.seed(seed)

device = "cuda:0" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

```
Using cuda:0 device
```

Load and check MNIST dataset

```
# mean and std for MNIST dataset
mean, std = 0.1307, 0.3081

# Define data transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

## Load MNIST training and testing dataset
original_train_ds = torchvision.datasets.MNIST(root = 'data/', train = True, transform = transform, download = True)
original_test_ds = torchvision.datasets.MNIST(root = 'data/', train = False, transform = transform, download = True)

print('Train dataset:', len(original_train_ds))
print('Test dataset:', len(original_test_ds))
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9.91M/9.91M [00:02<00:00, 4.10MB/s]
Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>
```

```
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28.9k/28.9k [00:00<00:00, 132kB/s]
Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1.65M/1.65M [00:01<00:00, 1.07MB/s]
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4.54k/4.54k [00:00<00:00, 3.92MB/s]Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw

Train dataset: 60000
Test dataset: 10000
```

```
classes = original_train_ds.classes
classes
```

```
['0 - zero',
 '1 - one',
 '2 - two',
 '3 - three',
 '4 - four',
 '5 - five',
 '6 - six',
 '7 - seven',
 '8 - eight',
 '9 - nine']
```

```
# data[a][b]: a is an image, b is its info where data[a][0]: data, data[a][1]: label
class_count = [original_train_ds[img][1] for img in range(len(original_train_ds))] # iterate through each image
print("Labels count:", dict(Counter(class_count)))
```

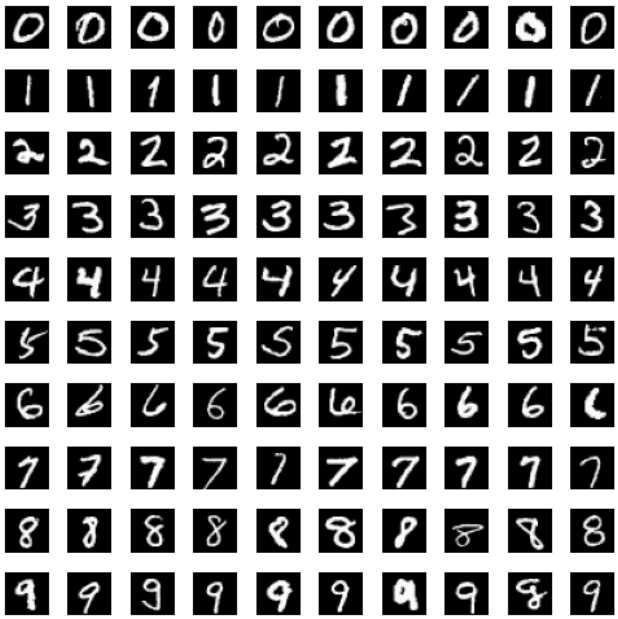
```
Labels count: {5: 5421, 0: 5923, 4: 5842, 1: 6742, 9: 5949, 2: 5958, 3: 6131, 6: 5918, 7: 6265, 8: 5851}
```

100 labeled MNIST dataset

```
def sample_100_digits(data):
    random.seed(42)  # For reproducibility
    labeled_indices = []

    # Loop through digits 0-9 and sample 10 examples per digit
    for digit in range(10):
        label_indices = torch.where(data.targets == digit)[0]
        subset_indices = random.sample(label_indices.tolist(), 10)
        labeled_indices.extend(subset_indices)

    # Create the labeled dataset (100 samples)
    labeled_dataset = Subset(data, labeled_indices)

    # Create the unlabeled dataset by removing the labeled indices
    all_indices = set(range(len(data)))
    unlabeled_indices = list(all_indices - set(labeled_indices))
    unlabeled_dataset = Subset(data, unlabeled_indices)

    return labeled_dataset, unlabeled_dataset

# Plotting the 100 samples
def plot_mnist_samples(dataset):
    fig, axes = plt.subplots(10, 10, figsize=(5, 5))
    axes = axes.flatten()

    for i, (img, label) in enumerate(dataset):
        axes[i].imshow(img[0], cmap='gray')  # Convert the 1-channel tensor to 2D image
        axes[i].axis('off')
        if i % 10 == 0:  # Mark the row with the digit label
            axes[i].set_ylabel(f'Digit {label}', rotation=0, labelpad=10, fontsize=10)

    plt.tight_layout()
    plt.show()
```

```
ds_100_labels, ds_unlabeled = sample_100_digits(original_train_ds)

class_count = [ds_100_labels[img][1] for img in range(len(ds_100_labels))]
print("Labels count:", dict(Counter(class_count)), "\n")

# plot 100 labeled data
plot_mnist_samples(ds_100_labels)
```

```
Labels count: {0: 10, 1: 10, 2: 10, 3: 10, 4: 10, 5: 10, 6: 10, 7: 10, 8: 10, 9: 10}
```



Data preparation

```
# Get labels to stratify the split
labels = []
for i in range(len(ds_100_labels)) :
  _, label = ds_100_labels[i]
  labels.append(label)

# Split indices if the dataset is composed of images
train_idx, val_idx = train_test_split(list(range(len(labels))),
                                      test_size = 0.2,
                                      stratify = labels,
                                      random_state = 42,
                                      shuffle=True)

train_set = torch.utils.data.Subset(ds_100_labels, train_idx)
val_set = torch.utils.data.Subset(ds_100_labels, val_idx)

# data[a][b]: a is an image, b is its info where data[a][0]: data, data[a][1]: label
print("Labels count for training set:", dict(Counter([train_set[img][1] for img in range(len(train_set))])))
```

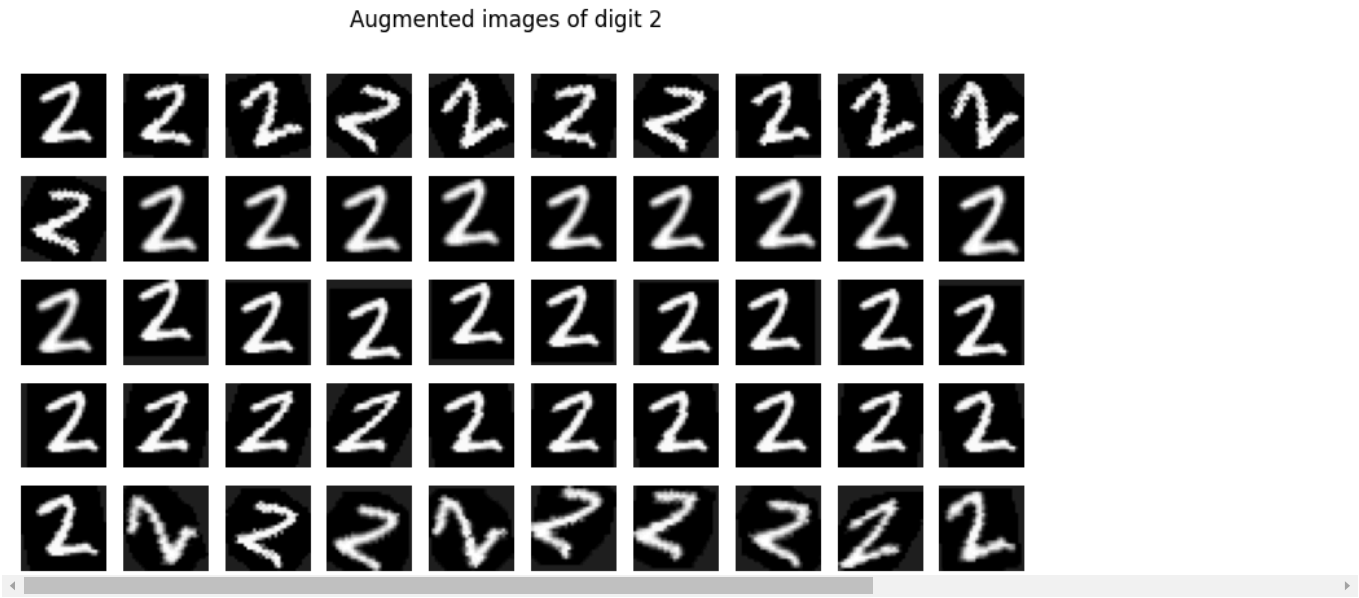
```
Labels count for training set: {9: 8, 5: 8, 6: 8, 2: 8, 3: 8, 0: 8, 4: 8, 7: 8, 1: 8, 8: 8}
```

```
# transformer for data augmentation
def data_transformation(data, transformer_list = [], num_per_img = 10):
  """
  data augmentation applying each transforms for each original image,
  num_per_img is the number of images augmented from each original image for each transform
  """
  torch.manual_seed(42)
  augmented_data = []
  augmented_labels = []
  for i in range(len(data)):
    # Data augmentation
    img, label = data[i]
    augmented_data.append(img)
    augmented_labels.append(label)
    for t in transformer_list:
      # Generate new images from the original image
      for _ in range(num_per_img) :
        random.seed(42)
        new_img = t(img)
        augmented_data.append(new_img)
      # Update new labels
      augmented_labels.extend([label] * (num_per_img))
```

```
  augmented_data = torch.stack(augmented_data) # Concatenate a sequence of tensors
  augmented_labels = torch.tensor(augmented_labels)
  return TensorDataset(augmented_data, augmented_labels)

# Visualize the augmented data
def visualize_images(data, title='', rows = 10, cols =10 , fig_size = (5,5)):
  # function to visualize images

  plt.figure(figsize=fig_size)  # Set the figure size to 5x5
  for idx, img in enumerate(data):
    plt.subplot(rows, cols, idx + 1)
    plt.imshow(img[0].squeeze(), cmap='gray')
    plt.axis('off')
  plt.suptitle(title)
  plt.show()
```

```
rotation45 = transforms.Compose([transforms.RandomRotation(45),
                                 transforms.Normalize(mean, std)])

crop_resize = transforms.Compose([transforms.RandomResizedCrop(28, scale=(0.8, 1), ratio = (1,1)),
                                  transforms.Normalize(mean, std)])

translation = transforms.Compose([transforms.RandomAffine(degrees = 0, translate=(0.1, 0.1)),
                                  transforms.Normalize(mean, std)])

shear30 = transforms.Compose([transforms.RandomAffine(0,shear=30),
                              transforms.Normalize(mean, std)])

transform_pipeline = transforms.Compose([transforms.RandomRotation(45),
                                         transforms.RandomResizedCrop(28, scale=(0.8, 1), ratio = (1,1)),
                                         transforms.RandomAffine(degrees = 0, translate=(0.1, 0.1)),
                                         transforms.RandomAffine(degrees = 0, shear = 30),
                                         transforms.Normalize(mean, std)])

transformer_list = [rotation45, crop_resize, translation, shear30, transform_pipeline]

data_aug = data_transformation(train_set, transformer_list, num_per_img = 10)
print("Augmented data size:", len(data_aug), "\n")

# select a digit from augmented data to plot
selected_digit = 2
targets_aug = []
for img in data_aug:
  targets_aug.append(img[1])
indices_label = [i for i in range(len(targets_aug)) if targets_aug[i] == selected_digit]

# visualize selected digit
visualize_images(Subset(data_aug, indices_label[:50]), title=f'Augmented images of digit {selected_digit}' , rows = 5, cols = 10, fig_size =
```

```
Augmented data size: 4080
```



Model architecture

```
def get_memory_params_model(model):
  # Model architecture
  print(model)
```

```
  print("Model memory allocation : {:.2e}".format(torch.cuda.memory_reserved(0) - torch.cuda.memory_allocated(0)))

  # Find total parameters and trainable parameters
  total_params = sum(p.numel() for p in model.parameters())
  print("{} total parameters.".format(total_params))
  total_trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
  print("{} training parameters.".format(total_trainable_params))

##################### MLP #####################

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.lin1 = nn.Linear(in_features=784, out_features=512)
        self.batchnorm1 = nn.BatchNorm1d(512)
        self.lin2 = nn.Linear(in_features=512, out_features=128)
        self.batchnorm2 = nn.BatchNorm1d(128)
        self.lin3 = nn.Linear(in_features=128, out_features=64)
        self.batchnorm3 = nn.BatchNorm1d(64)
        self.lin4 = nn.Linear(in_features=64, out_features=10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = F.relu(self.lin1(x))
        x = self.batchnorm1(x)
        x = F.relu(self.lin2(x))
        x = self.batchnorm2(x)
        x = F.relu(self.lin3(x))
        x = self.batchnorm3(x)
        x = self.lin4(x)
        return x

##################### CNN #####################

class CNN(nn.Module):
  def __init__(self):
    super().__init__() # always subclass
    self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding = "same") # conv layer 28x28x32
    self.batchnorm1 = nn.BatchNorm2d(32)
    self.pool1 = nn.MaxPool2d(2) # maxpooling 14x14x32

    self.conv2 = nn.Conv2d(32, 64, kernel_size=3) # first conv layer 12x12x64
    self.batchnorm2 = nn.BatchNorm2d(64)
    self.pool2 = nn.MaxPool2d(2) # maxpooling 6x6x64

    self.conv3 = nn.Conv2d(64, 128, kernel_size=3) # conv lazyer 4x4x128
    self.batchnorm3 = nn.BatchNorm2d(128)
    self.pool3 = nn.MaxPool2d(2) # maxpooling 2x2x128

    self.fc1 = nn.Linear(128*2*2, 192) # we have 10 probability classes to predict so 10 output features
    self.batchnorm4 = nn.BatchNorm1d(192)
    self.fc2 = nn.Linear(192, 64)
    self.batchnorm5 = nn.BatchNorm1d(64)
    self.fc3 = nn.Linear(64, 10)

  def forward(self, x):
    x = self.conv1(x)
    x = self.batchnorm1(x)
    x = F.relu(x)
    x = self.pool1(x)

    x = self.conv2(x)
    x = self.batchnorm2(x)
    x = F.relu(x)
    x = self.pool2(x)

    x = self.conv3(x)
    x = self.batchnorm3(x)
    x = F.relu(x)
    x = self.pool3(x)

    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = self.batchnorm4(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = self.batchnorm5(x)
    x = F.relu(x)
    x = self.fc3(x)

    return x

print("##################### MLP #####################")

mlp = MLP().to(device)
```

```
summary(mlp, input_size=(1, 28, 28))
print("\n\n")
get_memory_params_model(mlp)

print("\n#################### CNN ####################")

cnn = CNN().to(device)
get_memory_params_model(cnn)
print("\n\n")
summary(cnn, input_size=(1, 28, 28))
```

```
#################### MLP ####################
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 512]         401,920
       BatchNorm1d-2                  [-1, 512]           1,024
            Linear-3                  [-1, 128]          65,664
       BatchNorm1d-4                  [-1, 128]             256
            Linear-5                   [-1, 64]           8,256
       BatchNorm1d-6                   [-1, 64]             128
            Linear-7                   [-1, 10]             650
================================================================
Total params: 477,898
Trainable params: 477,898
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 1.82
Estimated Total Size (MB): 1.84
----------------------------------------------------------------



MLP(
  (lin1): Linear(in_features=784, out_features=512, bias=True)
  (batchnorm1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (lin2): Linear(in_features=512, out_features=128, bias=True)
  (batchnorm2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (lin3): Linear(in_features=128, out_features=64, bias=True)
  (batchnorm3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (lin4): Linear(in_features=64, out_features=10, bias=True)
)
Model memory allocation : 1.26e+07
477898 total parameters.
477898 training parameters.

#################### CNN ####################
CNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=same)
  (batchnorm1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (batchnorm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (batchnorm3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=512, out_features=192, bias=True)
  (batchnorm4): BatchNorm1d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=192, out_features=64, bias=True)
  (batchnorm5): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
)
Model memory allocation : 1.18e+07
205130 total parameters.
205130 training parameters.
```

Functions for training and evaluation phase

```
# Data Loader
def load_data(train, val, test, train_batch, val_batch, test_batch):
  torch.manual_seed(42)
  train_loader = torch.utils.data.DataLoader(train, batch_size=train_batch,
                                             shuffle=True, num_workers=2)
  val_loader = torch.utils.data.DataLoader(val, batch_size=val_batch,
                                           shuffle=False, num_workers=2)
  test_loader = torch.utils.data.DataLoader(test, batch_size=test_batch,
                                           shuffle=False, num_workers=2)
  return train_loader, val_loader, test_loader

def get_accuracy(y_true, y_pred):
    return int(np.sum(np.equal(y_true,y_pred))) / y_true.shape[0]

# class to implement Early Stopping
```

```
class EarlyStopping:
  """Early stops the training if validation loss doesn't improve after a given patience."""
  def __init__(self, patience=1, min_delta=0):
    self.patience = patience
    self.min_delta = min_delta
    self.counter = 0
    self.min_validation_loss = float('inf')

  def early_stop(self, validation_loss):
    if validation_loss < self.min_validation_loss:
      self.min_validation_loss = validation_loss
      self.counter = 0
    elif validation_loss > (self.min_validation_loss + self.min_delta):
      self.counter += 1
      print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
      if self.counter >= self.patience:
        return True
    return False

# Train function
def train_model(model, epochs, train_loader, val_loader, optimizer_name = 'Adam', patience = 1, early_stopping = True, learning_rate = 0.001
  torch.manual_seed(42)
  # Init
  output_fn = torch.nn.Softmax(dim=1) # we instantiate the softmax activation function for the output probabilities
  criterion = nn.CrossEntropyLoss() # we instantiate the loss function
  optimizer = getattr(optim, optimizer_name)(model.parameters(), lr=learning_rate) # we instantiate Adam optimizer that takes as inputs the

  loss_valid,acc_valid =[],[]
  loss_train,acc_train =[],[]

  # initialize the early_stopping object
  if early_stopping:
    early_stopper = EarlyStopping(patience=patience)

  for epoch in tqdm(range(epochs)):
    torch.manual_seed(42)
    # Training loop
    model.train() # always specify that the model is in training mode
    running_loss = 0.0 # init loss
    running_acc = 0.

    # Loop over batches returned by the data loader
    for idx, batch in enumerate(train_loader):

      # get the inputs; batch is a tuple of (inputs, labels)
      inputs, labels = batch
      inputs = inputs.to(device) # put the data on the same device as the model
      labels = labels.to(device)

      # put to zero the parameters gradients at each iteration to avoid accumulations
      optimizer.zero_grad()

      # forward pass + backward pass + update the model parameters
      out = model(x=inputs) # get predictions
      loss = criterion(out, labels) # compute loss
      loss.backward() # compute gradients
      optimizer.step() # update model parameters according to these gradients and our optimizer strategy

      # Iteration train metrics
      running_loss += loss.view(1).item()
      t_out = output_fn(out.detach()).cpu().numpy() # compute softmax (previously instantiated) and detach predictions from the model graph
      t_out=t_out.argmax(axis=1)  # the class with the highest energy is what we choose as prediction
      ground_truth = labels.cpu().numpy() # detach the labels from GPU device
      running_acc += get_accuracy(ground_truth, t_out)

    ### Epochs train metrics ###
    acc_train.append(running_acc/len(train_loader))
    loss_train.append(running_loss/len(train_loader))

    # compute loss and accuracy after an epoch on the train and valid set
    model.eval() # put the model in evaluation mode (this prevents the use of dropout layers for instance)

    ### VALIDATION DATA ###
    with torch.no_grad(): # since we're not training, we don't need to calculate the gradients for our outputs
      idx = 0
      for batch in val_loader:
        inputs,labels=batch
        inputs=inputs.to(device)
        labels=labels.to(device)
        if idx==0:
          t_out = model(x=inputs)
          t_loss = criterion(t_out, labels).view(1).item()
          t_out = output_fn(t_out).detach().cpu().numpy() # compute softmax (previously instantiated) and detach predictions from the model
          t_out=t_out.argmax(axis=1)  # the class with the highest energy is what we choose as prediction
          ground_truth = labels.cpu().numpy() # detach the labels from GPU device
```

```
        else:
          out = model(x=inputs)
          t_loss = np.hstack((t_loss,criterion(out, labels).item()))
          t_out = np.hstack((t_out,output_fn(out).argmax(axis=1).detach().cpu().numpy()))
          ground_truth = np.hstack((ground_truth,labels.detach().cpu().numpy()))
        idx+=1

      acc_valid.append(get_accuracy(ground_truth,t_out))
      loss_valid.append(np.mean(t_loss))

    print('| Epoch: {}/{} | Train: Loss {:.4f} Accuracy : {:.4f} '\
        '| Val: Loss {:.4f} Accuracy : {:.4f}\n'.format(epoch+1,epochs,loss_train[epoch],acc_train[epoch],loss_valid[epoch],acc_valid[epoch]

    # early_stopping check if the validation loss has decresed, if yes, it will make a checkpoint of the current model
    if early_stopping:
      stop_bool = early_stopper.early_stop(loss_valid[epoch])
      if stop_bool:
        print("Early stopping")
        break

  # load the last checkpoint with the best model
  #if early_stopping:
    #model.load_state_dict(torch.load('checkpoint.pt'))

  return model, loss_train, loss_valid, acc_train, acc_valid

def plot_accuracy_loss(model_name, loss_train, loss_valid, acc_train, acc_valid) :
  """plot the accuracy and loss functions (for each epoch)
    early_stop_point = True: visualize the early stopping
  """

  fig = plt.figure(figsize = (12, 5))

  # --- Metrics plot
  def plot_metric(model_name, metric_train, metric_valid, metric_name) :
    """plot metrics of both datasets"""
    plt.plot(range(1, len(metric_train) + 1), metric_train, label='training set', marker='o', linestyle='solid',linewidth=1, markersize=6)
    plt.plot(range(1, len(metric_valid) + 1), metric_valid, label='validation set', marker='o', linestyle='solid',linewidth=1, markersize=6)
    # find position of lowest validation loss
    plt.title(f"{model_name}-model {metric_name}")
    plt.xlabel('#Epochs')
    plt.ylabel(f'{metric_name}')
    plt.legend(bbox_to_anchor=( 1., 1.))

  # Plot loss functions
  ax = fig.add_subplot(121)
  for side in ['right', 'top']:
    ax.spines[side].set_visible(False)
  plot_metric(model_name, loss_train, loss_valid, "Loss")

  # Plot accuracy function
  ax = fig.add_subplot(122)
  for side in ['right', 'top']:
    ax.spines[side].set_visible(False)
  plot_metric(model_name, acc_train, acc_valid, "Accuracy")

def test_model(model, test_loader):
  output_fn = torch.nn.Softmax(dim=1)
  model.eval()
  torch.manual_seed(42)
  with torch.no_grad():
    idx = 0
    for batch in test_loader:
      inputs,labels=batch
      inputs=inputs.to(device)
      labels=labels.to(device)
      if idx==0:
        t_out = model(x=inputs)
        t_out = output_fn(t_out).detach().cpu().numpy()
        t_out=t_out.argmax(axis=1)
        ground_truth = labels.detach().cpu().numpy()
      else:
        out = model(x=inputs)
        t_out = np.hstack((t_out,output_fn(out).argmax(axis=1).detach().cpu().numpy()))
        ground_truth = np.hstack((ground_truth,labels.detach().cpu().numpy()))
      idx+=1

  return get_accuracy(ground_truth,t_out)
```

Training, Validating, Testing

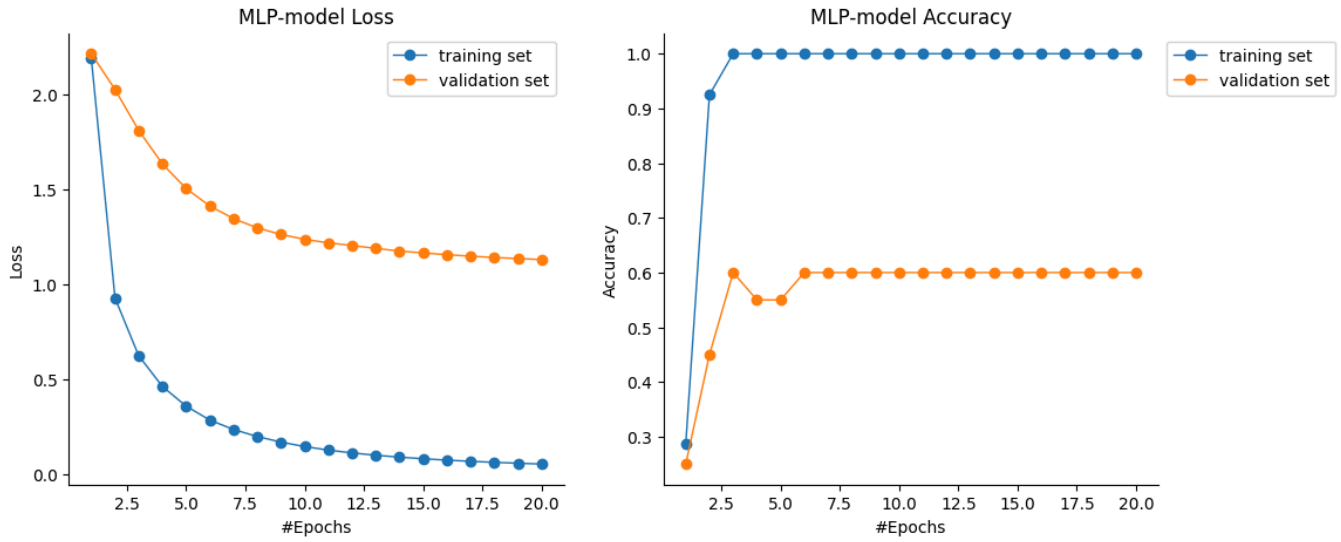
1. Élément de liste

2. Élément de liste

```
# Load data for train, validate, test
train_ld, val_ld, test_ld = load_data(train_set, val_set, original_test_ds, 20, 20, 20)
```

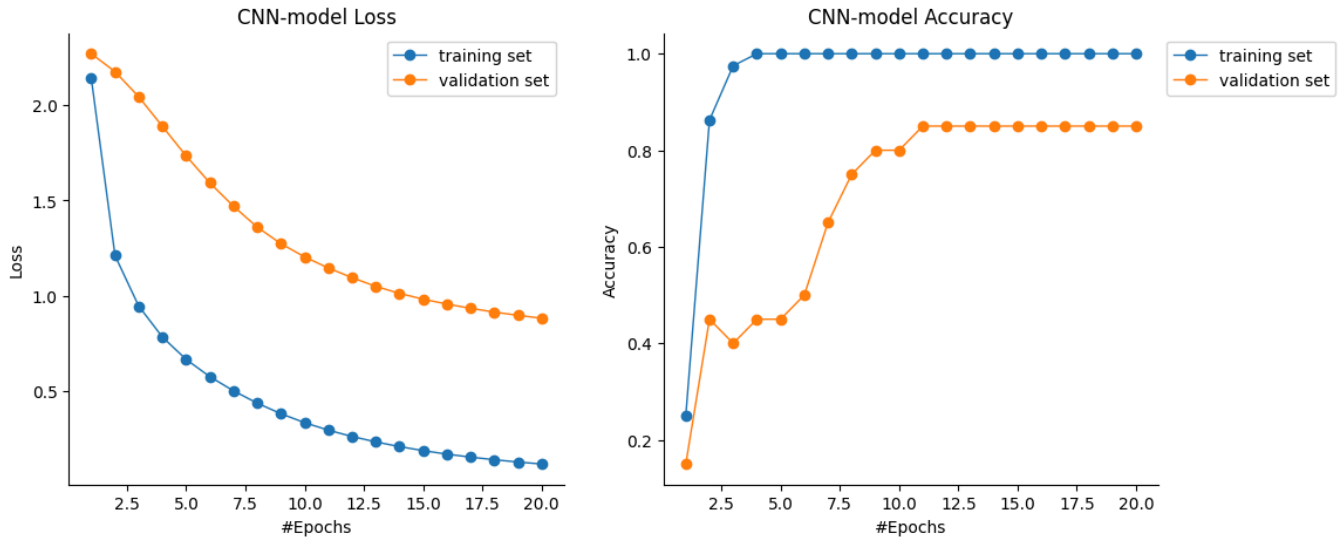
```
# train
mlp_trained, loss_train, loss_valid, acc_train, acc_valid = train_model(mlp, 20, train_ld, val_ld, optimizer_name = 'Adam', patience = 10, e
plot_accuracy_loss('MLP', loss_train, loss_valid, acc_train, acc_valid)
```

```
  5%|▌         | 1/20 [00:00<00:12,  1.55it/s]| Epoch: 1/20 | Train: Loss 2.1879 Accuracy : 0.2875 | Val: Loss 2.2136 Accuracy : 0.2500
 10%|█         | 2/20 [00:00<00:07,  2.54it/s]| Epoch: 2/20 | Train: Loss 0.9274 Accuracy : 0.9250 | Val: Loss 2.0251 Accuracy : 0.4500
 15%|█▌        | 3/20 [00:01<00:05,  3.14it/s]| Epoch: 3/20 | Train: Loss 0.6240 Accuracy : 1.0000 | Val: Loss 1.8100 Accuracy : 0.6000
 20%|██        | 4/20 [00:01<00:04,  3.59it/s]| Epoch: 4/20 | Train: Loss 0.4623 Accuracy : 1.0000 | Val: Loss 1.6333 Accuracy : 0.5500
 25%|██▌       | 5/20 [00:01<00:03,  3.93it/s]| Epoch: 5/20 | Train: Loss 0.3590 Accuracy : 1.0000 | Val: Loss 1.5041 Accuracy : 0.5500
 30%|███       | 6/20 [00:01<00:03,  4.23it/s]| Epoch: 6/20 | Train: Loss 0.2856 Accuracy : 1.0000 | Val: Loss 1.4121 Accuracy : 0.6000
 35%|███▌      | 7/20 [00:01<00:02,  4.36it/s]| Epoch: 7/20 | Train: Loss 0.2361 Accuracy : 1.0000 | Val: Loss 1.3458 Accuracy : 0.6000
 40%|████      | 8/20 [00:02<00:02,  4.26it/s]| Epoch: 8/20 | Train: Loss 0.1990 Accuracy : 1.0000 | Val: Loss 1.2971 Accuracy : 0.6000
 45%|████▌     | 9/20 [00:02<00:02,  4.45it/s]| Epoch: 9/20 | Train: Loss 0.1697 Accuracy : 1.0000 | Val: Loss 1.2623 Accuracy : 0.6000
 50%|█████     | 10/20 [00:02<00:02,  4.56it/s]| Epoch: 10/20 | Train: Loss 0.1462 Accuracy : 1.0000 | Val: Loss 1.2368 Accuracy : 0.600
 55%|█████▌    | 11/20 [00:02<00:01,  4.63it/s]| Epoch: 11/20 | Train: Loss 0.1276 Accuracy : 1.0000 | Val: Loss 1.2181 Accuracy : 0.600
 60%|██████    | 12/20 [00:03<00:01,  4.51it/s]| Epoch: 12/20 | Train: Loss 0.1127 Accuracy : 1.0000 | Val: Loss 1.2043 Accuracy : 0.600
 65%|██████▌   | 13/20 [00:03<00:01,  4.59it/s]| Epoch: 13/20 | Train: Loss 0.1005 Accuracy : 1.0000 | Val: Loss 1.1890 Accuracy : 0.600
 70%|███████   | 14/20 [00:03<00:01,  4.48it/s]| Epoch: 14/20 | Train: Loss 0.0905 Accuracy : 1.0000 | Val: Loss 1.1752 Accuracy : 0.600
 75%|███████▌  | 15/20 [00:03<00:01,  4.54it/s]| Epoch: 15/20 | Train: Loss 0.0821 Accuracy : 1.0000 | Val: Loss 1.1651 Accuracy : 0.600
 80%|████████  | 16/20 [00:03<00:00,  4.57it/s]| Epoch: 16/20 | Train: Loss 0.0748 Accuracy : 1.0000 | Val: Loss 1.1558 Accuracy : 0.600
 85%|████████▌ | 17/20 [00:04<00:00,  4.37it/s]| Epoch: 17/20 | Train: Loss 0.0686 Accuracy : 1.0000 | Val: Loss 1.1481 Accuracy : 0.600
 90%|█████████ | 18/20 [00:04<00:00,  4.48it/s]| Epoch: 18/20 | Train: Loss 0.0632 Accuracy : 1.0000 | Val: Loss 1.1412 Accuracy : 0.600
 95%|█████████▌| 19/20 [00:04<00:00,  4.53it/s]| Epoch: 19/20 | Train: Loss 0.0585 Accuracy : 1.0000 | Val: Loss 1.1355 Accuracy : 0.600
100%|██████████| 20/20 [00:04<00:00,  4.16it/s]| Epoch: 20/20 | Train: Loss 0.0543 Accuracy : 1.0000 | Val: Loss 1.1304 Accuracy : 0.600
```




```
# train
cnn_trained, loss_train, loss_valid, acc_train, acc_valid = train_model(cnn, 20, train_ld, val_ld, optimizer_name='Adam', early_stopping = F
plot_accuracy_loss('CNN', loss_train, loss_valid, acc_train, acc_valid)
```

```
  5%|▌         | 1/20 [00:00<00:07,  2.55it/s]| Epoch: 1/20 | Train: Loss 2.1382 Accuracy : 0.2500 | Val: Loss 2.2684 Accuracy : 0.1500
 10%|█         | 2/20 [00:00<00:05,  3.40it/s]| Epoch: 2/20 | Train: Loss 1.2113 Accuracy : 0.8625 | Val: Loss 2.1746 Accuracy : 0.4500
 15%|█▌        | 3/20 [00:00<00:04,  3.60it/s]| Epoch: 3/20 | Train: Loss 0.9468 Accuracy : 0.9750 | Val: Loss 2.0427 Accuracy : 0.4000
 20%|██        | 4/20 [00:01<00:04,  3.95it/s]| Epoch: 4/20 | Train: Loss 0.7846 Accuracy : 1.0000 | Val: Loss 1.8879 Accuracy : 0.4500
 25%|██▌       | 5/20 [00:01<00:03,  4.08it/s]| Epoch: 5/20 | Train: Loss 0.6678 Accuracy : 1.0000 | Val: Loss 1.7333 Accuracy : 0.4500
 30%|███       | 6/20 [00:01<00:03,  4.19it/s]| Epoch: 6/20 | Train: Loss 0.5773 Accuracy : 1.0000 | Val: Loss 1.5919 Accuracy : 0.5000
 35%|███▌      | 7/20 [00:01<00:03,  3.60it/s]| Epoch: 7/20 | Train: Loss 0.5031 Accuracy : 1.0000 | Val: Loss 1.4692 Accuracy : 0.6500
 40%|████      | 8/20 [00:02<00:03,  3.34it/s]| Epoch: 8/20 | Train: Loss 0.4386 Accuracy : 1.0000 | Val: Loss 1.3604 Accuracy : 0.7500
 45%|████▌     | 9/20 [00:02<00:03,  3.28it/s]| Epoch: 9/20 | Train: Loss 0.3833 Accuracy : 1.0000 | Val: Loss 1.2735 Accuracy : 0.8000
 50%|█████     | 10/20 [00:02<00:03,  3.10it/s]| Epoch: 10/20 | Train: Loss 0.3365 Accuracy : 1.0000 | Val: Loss 1.2043 Accuracy : 0.800
 55%|█████▌    | 11/20 [00:03<00:02,  3.09it/s]| Epoch: 11/20 | Train: Loss 0.2972 Accuracy : 1.0000 | Val: Loss 1.1464 Accuracy : 0.850
 60%|██████    | 12/20 [00:03<00:02,  3.03it/s]| Epoch: 12/20 | Train: Loss 0.2641 Accuracy : 1.0000 | Val: Loss 1.0958 Accuracy : 0.850
 65%|██████▌   | 13/20 [00:03<00:02,  2.96it/s]| Epoch: 13/20 | Train: Loss 0.2357 Accuracy : 1.0000 | Val: Loss 1.0502 Accuracy : 0.850
 70%|███████   | 14/20 [00:04<00:02,  2.97it/s]| Epoch: 14/20 | Train: Loss 0.2112 Accuracy : 1.0000 | Val: Loss 1.0134 Accuracy : 0.850
 75%|███████▌  | 15/20 [00:04<00:01,  3.05it/s]| Epoch: 15/20 | Train: Loss 0.1900 Accuracy : 1.0000 | Val: Loss 0.9831 Accuracy : 0.850
 80%|████████  | 16/20 [00:04<00:01,  3.34it/s]| Epoch: 16/20 | Train: Loss 0.1718 Accuracy : 1.0000 | Val: Loss 0.9574 Accuracy : 0.850
 85%|████████▌ | 17/20 [00:05<00:00,  3.56it/s]| Epoch: 17/20 | Train: Loss 0.1561 Accuracy : 1.0000 | Val: Loss 0.9353 Accuracy : 0.850
 90%|█████████ | 18/20 [00:05<00:00,  3.75it/s]| Epoch: 18/20 | Train: Loss 0.1423 Accuracy : 1.0000 | Val: Loss 0.9155 Accuracy : 0.850
 95%|█████████▌| 19/20 [00:05<00:00,  3.93it/s]| Epoch: 19/20 | Train: Loss 0.1304 Accuracy : 1.0000 | Val: Loss 0.8984 Accuracy : 0.850
100%|██████████| 20/20 [00:05<00:00,  3.48it/s]| Epoch: 20/20 | Train: Loss 0.1199 Accuracy : 1.0000 | Val: Loss 0.8833 Accuracy : 0.850
```




```
# test
print("Accuracy for MLP:", test_model(mlp_trained, test_ld))
print("Accuracy for CNN:", test_model(cnn_trained, test_ld))
```

```
Accuracy for MLP: 0.6915
Accuracy for CNN: 0.8369
```

SimCLR

```
## Load MNIST training and testing dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

original_train_ds = torchvision.datasets.MNIST(root = 'data/', train = True, transform = transform, download = True)
original_test_ds = torchvision.datasets.MNIST(root = 'data/', train = False, transform = transform, download = True)
```

```
print('Train dataset:', len(original_train_ds))
print('Test dataset:', len(original_test_ds))
```

```
Train dataset: 60000
Test dataset: 10000
```

```
# Data augmentation for SimCLR
transform = transforms.Compose(
        [
            transforms.RandomHorizontalFlip(),
            transforms.RandomResizedCrop(size=28, scale=(0.2, 1.0)),
            transforms.ToTensor(),

    transforms.Normalize((0.1307,), (0.3081,))
])
```

```
# SimCLR dataset
class SimCLRDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, transform):
        self.dataset = dataset
        self.transform = transform

    def __getitem__(self, idx):
        img, label = self.dataset[idx]


        if isinstance(img, torch.Tensor):
            img = transforms.ToPILImage()(img)

        img1 = self.transform(img)  # Augmented view 1
        img2 = self.transform(img)  # Augmented view 2
        # img1 and img2 are a positive pair and all the other view are negative pairs for each image

        return img1, img2, label

    def __len__(self):
        return len(self.dataset)


simclr_train_ds = SimCLRDataset(original_train_ds, transform)
```

```
# Basic encoder
# class Encoder(nn.Module):
#     def __init__(self):
#         super(Encoder, self).__init__()
#         self.conv = nn.Sequential(
#             nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
#             nn.ReLU(),
#             nn.MaxPool2d(kernel_size=2),
#             nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
#             nn.ReLU(),
#             nn.MaxPool2d(kernel_size=2))

#     def forward(self, x):
#         x = self.conv(x)
#         x = x.view(x.size(0), -1)
#         return x
```

```
# class ResidualBlock(nn.Module):
#     def __init__(self, in_channels, out_channels, stride=1):
#         super(ResidualBlock, self).__init__()
#         self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
#         self.bn1 = nn.BatchNorm2d(out_channels)
#         self.relu = nn.ReLU(inplace=True)
#         self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
#         self.bn2 = nn.BatchNorm2d(out_channels)

#         self.shortcut = nn.Sequential()
#         if stride != 1 or in_channels != out_channels:
#             self.shortcut = nn.Sequential(
#                 nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride),
#                 nn.BatchNorm2d(out_channels)
#             )

#     def forward(self, x):
#         identity = self.shortcut(x)
#         out = self.conv1(x)
#         out = self.bn1(out)
#         out = self.relu(out)
#         out = self.conv2(out)
#         out = self.bn2(out)
```

```
#         out += identity # residual connexion
#         out = self.relu(out)
#         return out

# class ResNetEncoder(nn.Module):
#     def __init__(self):
#         super(ResNetEncoder, self).__init__()
#         self.initial = nn.Sequential(
#             nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
#             nn.BatchNorm2d(32),
#             nn.ReLU()
#         )

#         # 3 residual blocks
#         self.layer1 = ResidualBlock(32, 32, stride=1)
#         self.layer2 = ResidualBlock(32, 64, stride=2)
#         self.layer3 = ResidualBlock(64, 64, stride=2)

#     def forward(self, x):
#         x = self.initial(x)
#         x = self.layer1(x)
#         x = self.layer2(x)
#         x = self.layer3(x)
#         x = x.view(x.size(0), -1)
#         return x
```

```
class EncoderImproved(nn.Module):
    def __init__(self):
        super(EncoderImproved, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(p=0.3),
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(p=0.3),
        )

        self.layer3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=1),
            nn.Dropout(p=0.3),
        )
    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = x.view(x.size(0), -1)  # Flatten
        return x
```

```
class ProjectionHead(nn.Module):
    def __init__(self, input_dim=4096, output_dim=64):
        super(ProjectionHead, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, output_dim)
        )

    def forward(self, x):
        return self.mlp(x)
```

```
import torch.nn.functional as F

class NTXentLoss(nn.Module):
    def __init__(self, temperature=0.5):
        super(NTXentLoss, self).__init__()
        self.temperature = temperature

    def forward(self, z_i, z_j):
        z_i = F.normalize(z_i, dim=1)
```

```
        z_j = F.normalize(z_j, dim=1)
        similarity_matrix = torch.mm(z_i, z_j.T)
        labels = torch.arange(z_i.size(0)).to(z_i.device)
        loss = F.cross_entropy(similarity_matrix / self.temperature, labels)
        return loss
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

encoder = EncoderImproved().to(device)
projection_head = ProjectionHead().to(device)
criterion = NTXentLoss(temperature=0.5)
optimizer = torch.optim.Adam(list(encoder.parameters()) + list(projection_head.parameters()), lr=1e-3)

for epoch in range(5):
    encoder.train()
    projection_head.train()
    total_loss = 0
    for img1, img2, _ in DataLoader(simclr_train_ds, batch_size=128, shuffle=True):
        img1, img2 = img1.to(device), img2.to(device)
        h1, h2 = encoder(img1), encoder(img2)
        z1, z2 = projection_head(h1), projection_head(h2)
        loss = criterion(z1, z2)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {total_loss / len(simclr_train_ds)}")
```

```
Epoch 1, Loss: 0.03057633110284805
Epoch 2, Loss: 0.028999692515532177
Epoch 3, Loss: 0.02839210557937622
Epoch 4, Loss: 0.028001441045602163
Epoch 5, Loss: 0.027795663368701935
```

```
class Classifier(nn.Module):
    def __init__(self, encoder):
        super(Classifier, self).__init__()
        self.encoder = encoder
        self.mlp = nn.Sequential(
            nn.Linear(4096, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
            x = self.encoder(x)
            x = self.mlp(x)
            return x
```

```
classifier = Classifier(encoder).to(device)
for param in classifier.encoder.parameters():
    param.requires_grad = False # Freeze the parameters of the encoder that we already trained
```

```
# Get labels to stratify the split
labels = []
for i in range(len(ds_100_labels)) :
  _, label = ds_100_labels[i]
  labels.append(label)

# Split indices if the dataset is composed of images
train_idx, val_idx = train_test_split(list(range(len(labels))),
                                      test_size = 0.2,
                                      stratify = labels,
                                      random_state = 42,
                                      shuffle=True)

train_set = torch.utils.data.Subset(ds_100_labels, train_idx)
val_set = torch.utils.data.Subset(ds_100_labels, val_idx)

# data[a][b]: a is an image, b is its info where data[a][0]: data, data[a][1]: label
print("Labels count for training set:", dict(Counter([train_set[img][1] for img in range(len(train_set))])))
```

```
Labels count for training set: {9: 8, 5: 8, 6: 8, 2: 8, 3: 8, 0: 8, 4: 8, 7: 8, 1: 8, 8: 8}
```

```
train_loader = DataLoader(ds_100_labels, batch_size=16, shuffle=True) # replace ds_100_labels by train_set to do validation accuracy
val_loader = DataLoader(val_set, batch_size=8, shuffle=False)

import torch.optim as optim

optimizer = optim.Adam(classifier.mlp.parameters(), lr=1e-3)
```

```
optimizer = optim.Adam(classifier.parameters(), lr=0.0001)

criterion = torch.nn.CrossEntropyLoss()

def train_classifier(model, train_loader, val_loader, criterion, optimizer, device, epochs=10):
    model.train()

    for epoch in range(epochs):
        total_loss = 0
        correct = 0
        total = 0

        # Training loop
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Loss
            total_loss += loss.item()
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)

        # Training accuracy
        train_accuracy = correct / total * 100

        # Validation loop
        model.eval()
        val_correct = 0
        val_total = 0
        with torch.no_grad():
            for val_images, val_labels in val_loader:
                val_images, val_labels = val_images.to(device), val_labels.to(device)

                val_outputs = model(val_images)
                val_predictions = torch.argmax(val_outputs, dim=1)
                val_correct += (val_predictions == val_labels).sum().item()
                val_total += val_labels.size(0)

        val_accuracy = val_correct / val_total * 100

        print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss / len(train_loader):.4f}, "
              f"Train Accuracy: {train_accuracy:.2f}%, Validation Accuracy: {val_accuracy:.2f}%")

train_classifier(classifier, train_loader, val_loader, criterion, optimizer, device, epochs=20)
```

```
Epoch 1/20, Loss: 2.2270, Train Accuracy: 18.00%, Validation Accuracy: 10.00%
Epoch 2/20, Loss: 1.6865, Train Accuracy: 43.00%, Validation Accuracy: 55.00%
Epoch 3/20, Loss: 1.2010, Train Accuracy: 62.00%, Validation Accuracy: 80.00%
Epoch 4/20, Loss: 0.9547, Train Accuracy: 70.00%, Validation Accuracy: 85.00%
Epoch 5/20, Loss: 0.7420, Train Accuracy: 79.00%, Validation Accuracy: 90.00%
Epoch 6/20, Loss: 0.5947, Train Accuracy: 83.00%, Validation Accuracy: 90.00%
Epoch 7/20, Loss: 0.5338, Train Accuracy: 85.00%, Validation Accuracy: 85.00%
Epoch 8/20, Loss: 0.4161, Train Accuracy: 91.00%, Validation Accuracy: 90.00%
Epoch 9/20, Loss: 0.4068, Train Accuracy: 91.00%, Validation Accuracy: 95.00%
Epoch 10/20, Loss: 0.3418, Train Accuracy: 92.00%, Validation Accuracy: 95.00%
Epoch 11/20, Loss: 0.3211, Train Accuracy: 94.00%, Validation Accuracy: 95.00%
Epoch 12/20, Loss: 0.2994, Train Accuracy: 94.00%, Validation Accuracy: 100.00%
Epoch 13/20, Loss: 0.2891, Train Accuracy: 94.00%, Validation Accuracy: 95.00%
Epoch 14/20, Loss: 0.2315, Train Accuracy: 92.00%, Validation Accuracy: 95.00%
Epoch 15/20, Loss: 0.1894, Train Accuracy: 95.00%, Validation Accuracy: 100.00%
Epoch 16/20, Loss: 0.1264, Train Accuracy: 99.00%, Validation Accuracy: 95.00%
Epoch 17/20, Loss: 0.1141, Train Accuracy: 99.00%, Validation Accuracy: 100.00%
Epoch 18/20, Loss: 0.1007, Train Accuracy: 99.00%, Validation Accuracy: 100.00%
Epoch 19/20, Loss: 0.0829, Train Accuracy: 100.00%, Validation Accuracy: 100.00%
Epoch 20/20, Loss: 0.1047, Train Accuracy: 100.00%, Validation Accuracy: 100.00%
```

```
test_loader = DataLoader(original_test_ds, batch_size=128, shuffle=False)

def evaluate(model, dataloader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
```

```
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == labels).sum().item()
            total += labels.size(0)

    accuracy = correct / total * 100
    return accuracy

accuracy = evaluate(classifier, test_loader, device)
print(f"Test Accuracy after training on 100 labels: {accuracy:.2f}%")
```

```
Test Accuracy after training on 100 labels: 87.44%
```