Homework 1 — Exploring MLPs with PyTorch

Scope

In this assignment, you will explore multilayer perceptrons (MLPs) with PyTorch. We'll train the MLPs to classify handwritten digits with various settings, such as different loss functions and optimizers, and observe how they can change the performance of your model.

Instructions

- Deadline for this assignment is **January 26, 2024** via Gradescope
- The submission portal will remain open till mid February. Do keep in mind the late submission policy (Check slides from first lecture). Beyond that, we can open the submission portal on an individual basis as needed.
- Your submission has two parts: (i) a PDF report that details all the requested deliverables and (ii) the code for the assignment. You will upload the PDF report in Gradescope, marking which parts of the report corresponds to which deliverable. The code will be uploaded separately (instructions TBA), and we will run "software similarity" software on it to detect violations of the integrity policy.
- You can discuss this HW with others, but you are **not** allowed to share, borrow, copy or look at each other's codes.
- All code that you submit must be yours (except for the starter code that we provide). You are **not** allowed to use any software packages or code from the internet or other resources, except for a basic python installation of `torch`, `torchvision`, `numpy` and `matplotlib`.
- You are allowed to use Github co-pilot for help, and if you do, please specify this in your submission (which problems you have used it and to what extent).

Problem 1: Simple MLP for Binary Classification

In this problem, you will train a simple MLP to classify two handwritten digits: 0 vs 1. We provide some starter codes to do this task with steps. However, you do not need to follow the exact steps as long as you can complete the task in sections marked as [YOUR TASK].

1.1 Dataset Setup

We will use the MNIST dataset[1]. The `torchvision` package has supported this dataset. We can load the dataset in this way (the dataset will take up 63M of your disk space):

```
import torch
from torchvision import transforms, datasets


# define the data pre-processing
# convert the input to the range [-1, 1].
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize(0.5, 0.5)]
    )

# Load the MNIST dataset
# this command requires Internet to download the dataset
mnist = datasets.MNIST(root='./data',
                       train=True,
                       download=True,
                       transform=transform)
mnist_test = datasets.MNIST(root='./data',
                            train=False,
                            download=True,
                            transform=transform)
```

In Problem 1, we only focus on a binary classification between digits 0 and 1. Thus we filter the dataset to contain only samples of digits 0 and 1. Besides, we want to randomly split the original training data into two disjoint datasets: a new training set containing 80% original training samples and a validation dataset containing 20% original training samples. We provide the incomplete code as a hint:

```
from torch.utils.data import DataLoader

# Filter for digits 0 and 1
train_data = [data for data in mnist if data[1] < 2]
# Your code goes here
# test_data = ...

# Split training data into training and validation sets
# Your code goes here
# train_set = ...
```

[1]http://yann.lecun.com/exdb/mnist/

```
# val_set = ...

# Define DataLoaders to access data in batches
train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
# Your code goes here
# val_loader = ...
# test_loader = ...
```

1.2 Define an MLP

We want to define a simple MLP with only one hidden layer. You can use `torch.nn.Linear` to define a single MLP layer and pick an activation layer you like. Since our inputs are images with 28×28 pixels, the input dimension is $28 \times 28 = 784$. The problem is a binary classification, thus, the output dimension is 2.

```
import torch.nn as nn

# Define your MLP
class SimpleMLP(nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim):
        super(SimpleMLP, self).__init__()
        # Your code goes here
        # self.fc1 = ...
        # self.activation = ...
        # self.fc2 = ...


    def forward(self, x):
        # Your code goes here

hidden_dim = ...
model = SimpleMLP(in_dim=28 * 28,
                  hidden_dim=hidden_dim,
                  out_dim=2)
print(model)
```

1.3 Train the MLP

To train the model, we need to define a loss function (criterion) and an optimizer. The loss function tells us how far away the model's prediction is from the label. Once we have the loss, PyTorch can compute the gradient of the model automatically. The optimizer uses the gradient to update the model. For classification problems, we often use the Cross Entropy Loss. For the optimizer, we can use stochastic gradient descent optimizer or Adam optimizer:

```
criterion = nn.CrossEntropyLoss()
# optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

There are several hyper-parameters in the optimizer (please see the PyTorch document for details[2]). You can play with the hyper-parameters and see how they influence the training.

Now we have almost everything to train the model. We provide a sample code to complete the training loops:

```
num_epochs = 10
for epoch in range(num_epochs):
    for data, target in train_loader:
        # free the gradient from the previous batch
        optimizer.zero_grad()
        # reshape the image into a vector
        data = data.view(data.size(0), -1)
        # model forward
        output = model(data)
        # compute the loss
        loss = criterion(output, target)
        # model backward
        loss.backward()
        # update the model paramters
        optimizer.step()
```

After the training, we can use the validation dataset to know the performance of our model on new samples:

```
val_loss = count = 0
correct = total = 0
for data, target in val_loader:
    data = data.view(data.size(0), -1)
    output = model(data)
    val_loss += criterion(output, target).item()
    count += 1
    pred = output.argmax(dim=1)
    correct += (pred == target).sum().item()
    total += data.size(0)

val_loss = val_loss / count
val_acc = 100. * correct / total
print(f'Validation loss: {val_loss:.2f}, accuracy: {val_acc:.2f}%')
```

You can also perform validation after each epoch. But remember not to train (backward and update) on the validation dataset. Use the validation set to optimize performance. After you are done with this, report performance on the test set(You are encouraged not to use the test set for validation, i.e., use the test set only once after you are happy with the validation performance).

[YOUR TASK]

- Filter all samples representing digits ”0” or ”1” from the MNIST datasets.

[2]https://pytorch.org/docs/stable/optim.html

- Randomly split the training data into a training set (80% training samples) of a validation set (20% training samples).
- Define an MLP with 1 hidden layer and train the MLP to classify the digits ”0” vs ”1”. Report your MLP design and training details (which optimizer, number of epochs, learning rate, etc.)
- Keep other hyper-parameters the same, and train the model with different batch sizes: 2, 16, 128, 1024. Report the time cost, training, validation and test set accuracy of your model[3].

One tip about the hidden layer size is to begin with a small number, say $16 \sim 64$. Some people find hidden size $= \sqrt{\text{input size} \times \text{output size}}$ is a good choice in practice. If your model's training accuracy is too low, you can double the hidden layer size. However, if you find the training accuracy is high. Still, the validation accuracy is much lower, you may consider a smaller hidden layer size because your model has the risk of overfitting.

Problem 2: MNIST 10-class classification

Now we want to train an MLP to handle multi-class classification for all 10 digits in the MNIST dataset. We will use the full MNIST dataset without filtering for specific digits. You may modify the MLP so that it can be used for multi-class classification.

[YOUR TASK]

- Implement the training loop and evaluation section. Report the hyper-parameters you choose.
- Experiment with different numbers of neurons in the hidden layer and note any changes in performance.
- Write a brief analysis of the model's performance, including any challenges faced and how they were addressed. [4]

When you define a new model, remember to update the optimizer!

Problem 3: Handling Class Imbalance in MNIST Dataset

In this problem, we will explore how to handle class imbalance problems, which are very common in real-world applications. A modified MNIST dataset is created as follows: we choose all instances of digit “0”, and choose only 1% instances of digit “1” for both training and test sets:

```
# Filter for digits 0 and 1
train_0 = [data for data in mnist if data[1] == 0]
train_1 = [data for data in mnist if data[1] == 1]
train_1 = train_1[:len(train_1) // 100]
train_data = train_0 + train_1
```

[3]In our implementations, we trained our network for 10 epochs in about 10 seconds on a laptop, getting a test accuracy of 99%.

[4]In our implementations, we were able to train our network for 10 epochs in about 20 seconds on a laptop.

For such a class imbalance problem, accuracy may not be a good metric. Always predicting "0" regardless of the input can be 99% accurate. Instead, we use the F_1 score as the evaluation metric:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

where precision and recall are defined as:

$$\text{precision} = \frac{\text{number of instances correctly predicted as "1"}}{\text{number of instances predicted as "1"}}$$

$$\text{recall} = \frac{\text{number of instances correctly predicted as "1"}}{\text{number of instances labeled as "1"}}$$

To handle such a problem, some changes to the training may be necessary. Some suggestions include: 1) Adjusting the class weights in the loss function, i.e., use a larger weight for the minority class when computing the loss.
2) Implementing resampling techniques (either undersampling the majority class or oversampling the minority class).

[YOUR TASK]

- Create the imbalance datasets with all "0" digits and only 1% "1" digits.
- Implement the training loop and evaluation section (implementing the F_1 metric).
- Ignore the class imbalance problem and train the MLP. Report your hyper-parameter details and the F_1 score performance on the test set (as the baseline).
- Explore modifications to improve the performance of the class imbalance problem. Report your modifications and the F_1 scores performance on the test set.

[EXTRA BONUS]
If the hyper-parameters are chosen properly, the baseline can perform satisfactorily on the class imbalance problem with 1% digit "1". We want to challenge the baseline and handle more class-imbalanced datasets.

```
import random
N = 1000
# generate a class-imbalanced dataset controlled by "N"
train_0 = [data for data in mnist if data[1] == 0]
train_1 = [data for data in mnist if data[1] == 1]
random.shuffle(train_1)
train_1 = train_1[:len(train_1) // N]
train_data = train_0 + train_1
```

Can you propose new ways for the class imbalance problem and achieve stable and satisfactory performance for large $N = 500,\ 1000,\ \cdots$?

Problem 4: Reconstruct the MNIST images by Regression

In this problem, we want to train the MLP (with only one hidden layer) to complete a regression task: reconstruct the input image. The goal of this task is dimension reduction, and we set the hidden layer dimension to a smaller number, say 50. Once we can train the MLP to reconstruct the input images perfectly, we find an lower dimension representation of the MNIST images.

Since this is a reconstruction task, the labels of the images are not needed, and the target is the same as the inputs. Mean Squared Error (MSE) is recommended as the loss function:

```
criterion = nn.MSELoss()
```

Another tip is to add a tanh layer to the end of the model. Recall that our data pre-processing converts the data into the range $[-1, 1]$:

```
# define the data pre-processing
# convert the input to the range [-1, 1].
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize(0.5, 0.5)]
    )
```

Having a `torch.nn.Tanh()` activation layer at the end of the model can convert the output of the model into the range $[-1, 1]$, making the training easier.

[YOUR TASK]

- Define an MLP with only one hidden layer and set the hidden layer dimension as 50. Train the MLP to reconstruct input images from all 10 digits.
- Report the Mean Squared Error on the training, validation and test set. Report your hyper-parameter details.
- Pick 5 images for each digit from the test set. Visualize the original images and the reconstructed images using the MLP.