Lab 0 (Classification) Instructions

In this notebook, we will learn how to do classification of MNIST handwritten digits using PyTorch. The notebook contains all the info that you need to understand the basic mechanism of the classification theory.

The notebook also contains full code, your task is just to understand the code, and tweak the hyper-parameters (including number of hidden units, number of hidden layers, learning rate, num of iterations and so on) to improve the performance of the model. The final block of the code computes the prediction accuracy of the model on the testing set, please do not change this block.

Make sure that your final submission is a notebook that can be run from beginning to end, and you should print out the accuracy at the end of the notebook (i.e. be sure to run the last block after training). It is possible to achieve >98% of the accuracy on this dataset with a more complex neural network architecture and careful tuning of hyper-parameters. **Your grade will depend on the final prediction accuracy**. However, if you tweak the evaluation code to report false result, you will receive no credit for this assignment.

We encourage the students to take what they learn from here and try some other classification tasks on their own. A number of datasets are available in torch, see here: https://pytorch.org/vision/0.15/datasets.html. A good dataset to start with is the cifar10 dataset.

```
%reload_ext autoreload
%autoreload 2
import torch
import torchvision
import matplotlib.pyplot as plt
```

Classification using PyTorch

In the introduction notebook, we walked through how to solve a supervised learning regression problem (i.e. where the labels are continuous values) from scratch. Now, you will build a model that solves a classification problem (i.e. where the labels are discrete values). We will use the MNIST hand written digit dataset, a toy benchmark for image classification models. Let's first load the dataset via the PyTorch API.

Dataset

Here X_{train}, Y_{train} denote the training data and X_{test}, Y_{test} denote the testing data. We train the model on training set and evaluate its performance on testing set (to evaluate potential under-fitting or over-fitting). As can be seen below, X_{train} contains 60000 examples with 28×28 pixels. Y_{train} contains 60000 corresponding examples with 10 classes.

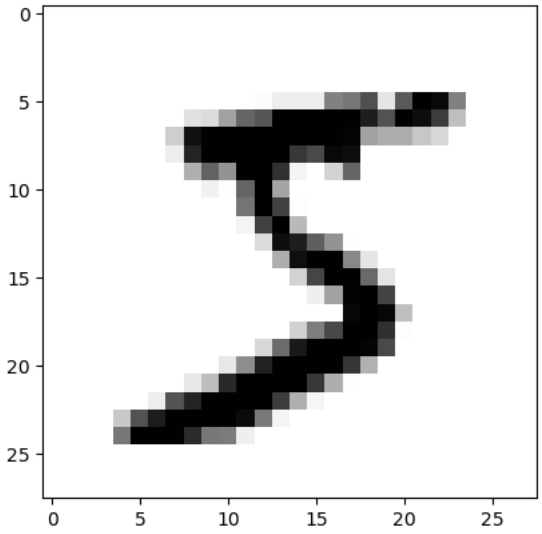
```
from torchvision import datasets, transforms

transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,)),
    ]
)

train_dataset = datasets.MNIST(
    "./mnist/MNIST_data/", download=True, train=True, transform=transform
)
test_dataset = datasets.MNIST(
    "./mnist/MNIST_data/", download=True, train=False, transform=transform
)

train_dataloader = torch.utils.data.DataLoader(
    train_dataset, batch_size=64, shuffle=True
)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=True)

print(len(train_dataset), len(test_dataset))
```

```
60000 10000
```

Let's look at one sample in the dataset:

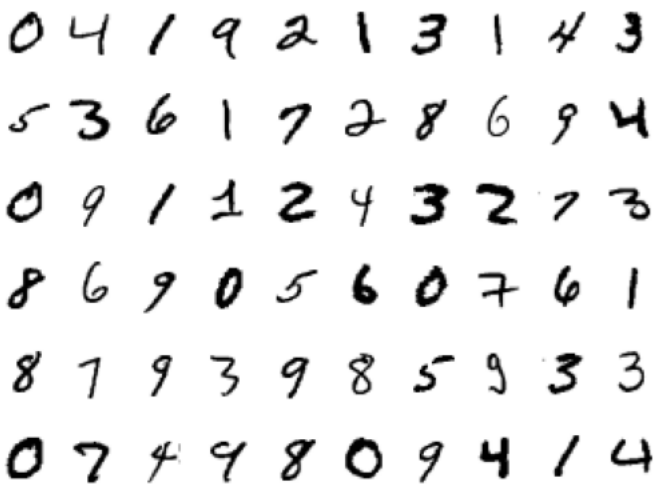
```
x, y = train_dataset[0]
print("Shape of x: ", x.shape)
print("Label: ", y)

plt.imshow(x.numpy().squeeze(), cmap="gray_r")
```

```
Shape of x:  torch.Size([1, 28, 28])
Label:  5
<matplotlib.image.AxesImage at 0x16a1367e0>
```



Let's look at a bunch of samples!

```
figure = plt.figure()
num_of_images = 60
for i in range(1, num_of_images + 1):
    x, y = train_dataset[i]
    plt.subplot(6, 10, i)
    plt.axis("off")
    plt.imshow(x.numpy().squeeze(), cmap="gray_r")
```



Model

You are responsible for defining the model f(x) = y, where x is an image and y is the digit found in the image. We provide one way to do this, but you will need to explore different model architectures and training strategies to get sufficient performance:

The shape of any image x is 28x28. We can reshape the image so it is instead a vector of length 28*28 = 784. Then, we can use the methods described in the introduction notebook to map a 784-dimensional vector to a 10-dimensional vector of probabilities. The digit in the image will then correspond to the index of the highest probability entry in the predicted 10-dimensional vector.

```
model = torch.nn.Sequential(torch.nn.Linear(28 * 28, 10))

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Unlike regression, where the predicted output y is a scalar in $\mathbb{R}$, in classification the model outputs logits or scores, one for each class. In our problem, there are 10 classes so the output of f(x) is a 10-dimensional vector of scores.

In order to convert these scores to probabilities, we can simply normalize. Typically, the softmax operation is used to compute probabilties from logits or scores. After computing the prediction y = f(x), which is a 10-dimensional, we can compute the probability vector p:

$$p_i = \frac{e^{y_i}}{\sum_{j=1}^{10} e^{y_j}}, i = 1, \dots, 10, \text{ or } p = softmax(y)$$

```
def probabilties_from_scores(y):
    f = torch.nn.Softmax(dim=1)
    p = f(y)
    return p
```

Loss

For K classes, our labels y in the dataset will take on an integer value between [0,K]. We must convert these values to one-hot representations. For example, when $K = 3$, y = 1 as a one-hot vector would be $[1, 0, 0]$ to represent the first class. If the model predicts a probability $p \in \mathbb{R}^K$ for this training instance, the loss function is

$$L = -\sum_{i=1}^{K} y_i \log p_i$$

Therefore, in the example where y = 1, L = - probability of the first entry since the one-hot vector is [1, 0, 0]. The loss function motivates assigning high probability to true classes and low probability otherwise.

```
def loss_fn(y_hat: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
    # y_hat has shape (N, 10) for 10 classes, y has shape (N,)
    f = torch.nn.CrossEntropyLoss()
    return f(y_hat, y)
```

Optimization

Now we perform stochastic gradient descent on the model. We can first try to evaluate the model's initial performance on the testing dataset as a comparison.

```
def evaluate(dataloader, model, log=False):
    correct_count, total_count = 0, 0
    for images_batch, labels_batch in dataloader:
        images = images_batch.reshape(images_batch.shape[0], 28 * 28)
        # images = images_batch
        scores = model(images)
        probabilities = probabilties_from_scores(scores)

        pred_label = probabilities.max(1, keepdim=True)[1]
        correct_count += pred_label.eq(labels_batch.view_as(pred_label)).sum().item()
        total_count += labels_batch.shape[0]

    if log:
        print("Number Of Images Tested =", total_count)
        print("Model Accuracy =", (correct_count / total_count))

    return correct_count / total_count
```

```
result = evaluate(test_dataloader, model, log=True)
```

```
Number Of Images Tested = 10000
Model Accuracy = 0.1005
```

Training Loop

Optimize the model to improve the prediction accuracy.

```
import torch.nn as nn


class SmallCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.feat = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.cls = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 10),  # logits
        )

    def forward(self, x):
        x = self.feat(x)
        return self.cls(x)
```

```
model = SmallCNN()
```

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


def train_one_epoch(
    model: nn.Module,
    loader: torch.utils.data.DataLoader,
    optimizer: torch.optim.Optimizer,
):
    model.train()
    running = 0.0
    for xb, yb in loader:
        xb: torch.Tensor
        yb: torch.Tensor
        xb, yb = xb.to(device), yb.to(device)
        optimizer.zero_grad(set_to_none=True)
        logits = model(xb)
        loss = loss_fn(logits, yb)
        loss.backward()
        optimizer.step()

        running += loss.item() * xb.size(0)
    return running / len(loader.dataset)


@torch.no_grad()
def evaluate(model: nn.Module, loader: torch.utils.data.DataLoader, log: bool = False):
    model.eval()
    total_loss, correct, n = 0.0, 0, 0
    for xb, yb in loader:
        xb: torch.Tensor
        yb: torch.Tensor
        xb, yb = xb.to(device), yb.to(device)
        logits = model(xb)
        total_loss += loss_fn(logits, yb).item() * xb.size(0)
        pred = logits.argmax(dim=1)
        correct += (pred == yb).sum().item()
        n += xb.size(0)

    if log:
        print("Number Of Images Tested =", n)
        print("Model Accuracy =", (correct / n))
    return total_loss / n, correct / n
```

```
model = SmallCNN().to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
```

```
def should_stop(acc_history, patience=3, min_delta=0.001):
    """
    Early stopping based on validation accuracy.
    Stops if accuracy does not improve by min_delta for 'patience' epochs.
    """
    if len(acc_history) < patience + 1:
        return False
    best_acc = max(acc_history[:-patience])
    for acc in acc_history[-patience:]:
        if acc > best_acc + min_delta:
            return False
    return True


def train_with_early_stopping(
    model,
    train_dataloader,
    test_dataloader,
    optimizer,
    max_epochs=10,
    patience=2,
    min_delta=0.001,
):
    acc_history = []
    for epoch in range(max_epochs):
        train_loss = train_one_epoch(model, train_dataloader, optimizer)
        val_loss, val_acc = evaluate(model, test_dataloader)
        acc_history.append(val_acc)
        print(
            f"Epoch {epoch+1:02d} | train {train_loss:.4f} | test {val_loss:.4f} | acc {val_acc*100:.2f}%"
        )
        if should_stop(acc_history, patience=patience, min_delta=min_delta):
            print(f"Early stopping at epoch {epoch+1}")
            break
    return acc_history


train_with_early_stopping(model, train_dataloader, test_dataloader, optimizer)
```

```
Epoch 01 | train 0.1778 | test 0.0571 | acc 98.21%
Epoch 02 | train 0.0575 | test 0.0334 | acc 98.84%
Epoch 03 | train 0.0405 | test 0.0345 | acc 98.80%
Epoch 04 | train 0.0348 | test 0.0315 | acc 99.07%
Epoch 05 | train 0.0296 | test 0.0265 | acc 99.17%
Epoch 06 | train 0.0255 | test 0.0268 | acc 99.13%
Early stopping at epoch 6
[0.9821, 0.9884, 0.988, 0.9907, 0.9917, 0.9913]
```

```
Start coding or generate with AI.
```

```
'\nepochs = 15\nfor epoch in range(epochs):\n    running_loss = 0\n    for images_batch, labels_batch in
train_dataloader:\n        images = images_batch.reshape(\n            images_batch.shape[0], 28 * 28\n        )  #
Flatten MNIST images into a 784 long vector\n\n        # forward pass\n        y_hat = model(images)\n        y =
labels_batch\n        L = loss(y_hat, y)\n\n        # backward pass\n        optimizer.zero_grad()\n
L.backward()\n\n        # update parameters\n        optimizer.step()\n\n        running_loss += L.item()\n\n
training_loss = running_loss / len(train_dataloader)\n    train_accuracy = evaluate(train_dataloader, model)\n
test_accuracy = evaluate(test_dataloader, model)\n\n    if epoch % 1 == 0:\n        print(\n            "Epoch {} -
Training loss: {}  Train Accuracy: {}  Test Accuracy: {}".format(\n                epoch, training_loss,
train_accuracy, test_accuracy\n            )\n        )\n'
```

Evaluation

Test the model's accuracy on the unseen test dataset.

```
result = evaluate(model, test_dataloader, log=True)
```

```
Number Of Images Tested = 10000
Model Accuracy = 0.9913
```

See that the model's performance improves from 10% accuracy to about 92%. For this simple task, getting an accuracy of 95% is not quite impressive. Try to tweak the parameters and neural network architecture to get better predictions! You should be able to get >= 98% test accuracy.