Custom data loading and data manipulation

Standard deviation and data normalization

- $\mathrm{avg}(x) = \mathbf{1}^T x/n$
- de-meaned vector is $\tilde{x} = x - \mathrm{avg}(x)\mathbf{1}$ (so $\mathrm{avg}(\tilde{x}) = 0$)
- standard deviation of x

$$\mathrm{std}(x) = \mathrm{rms}(\tilde{x}) = \frac{\|x - (\mathbf{1}^T x/n)\mathbf{1}\|}{\sqrt{n}}$$

- $\mathrm{std}(x)$ gives "typical amount" x_i vary from $\mathrm{avg}(x)$
- $\mathrm{std}(x) = 0$ only if $x = \alpha\mathbf{1}$ for some α
- a basic formula

$$\mathrm{rms}(x)^2 = \mathrm{avg}(x)^2 + \mathrm{std}(x)^2$$

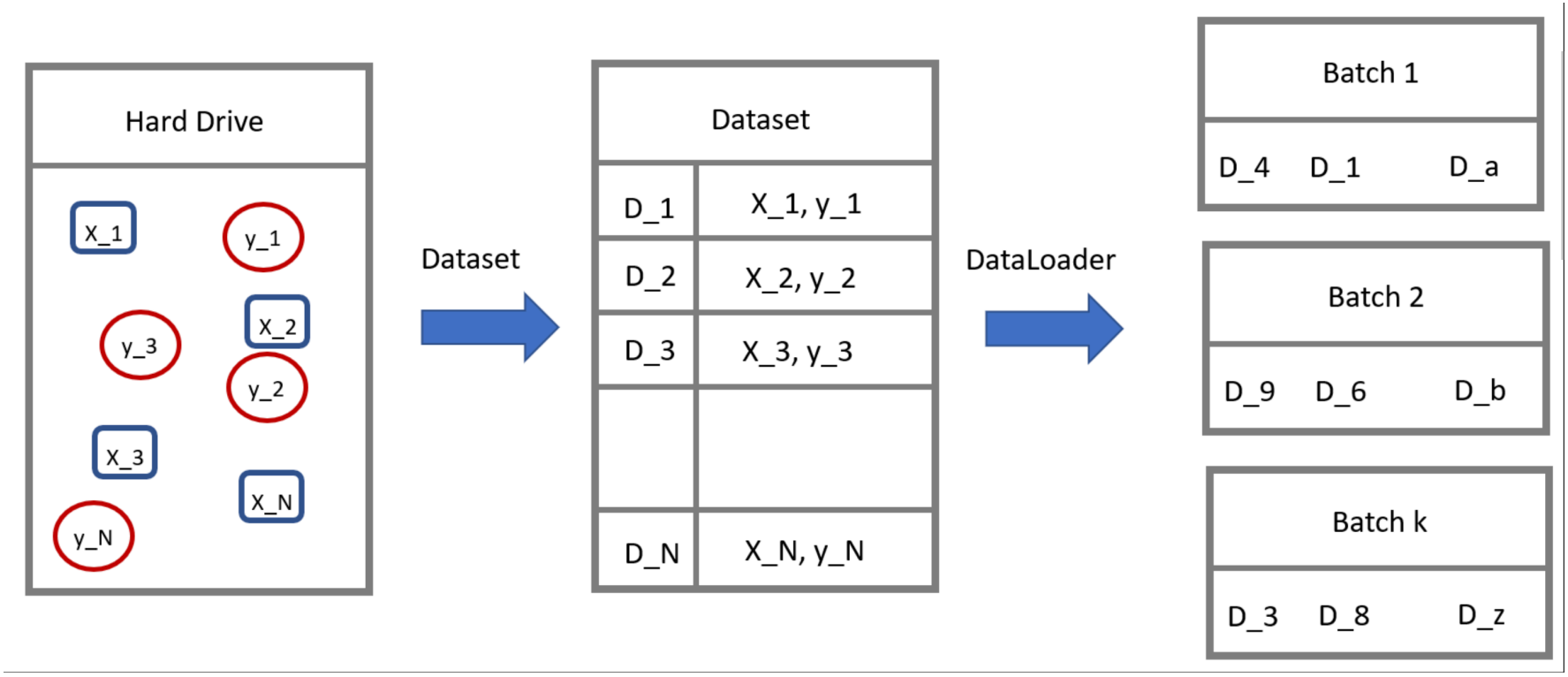
- standardization ($\mu = 0$, $\sigma = 1$, z-scores)

$$z = \frac{1}{\mathrm{std}(x)}(x - \mathrm{avg}(x)\mathbf{1})$$

Datasets and Dataloaders

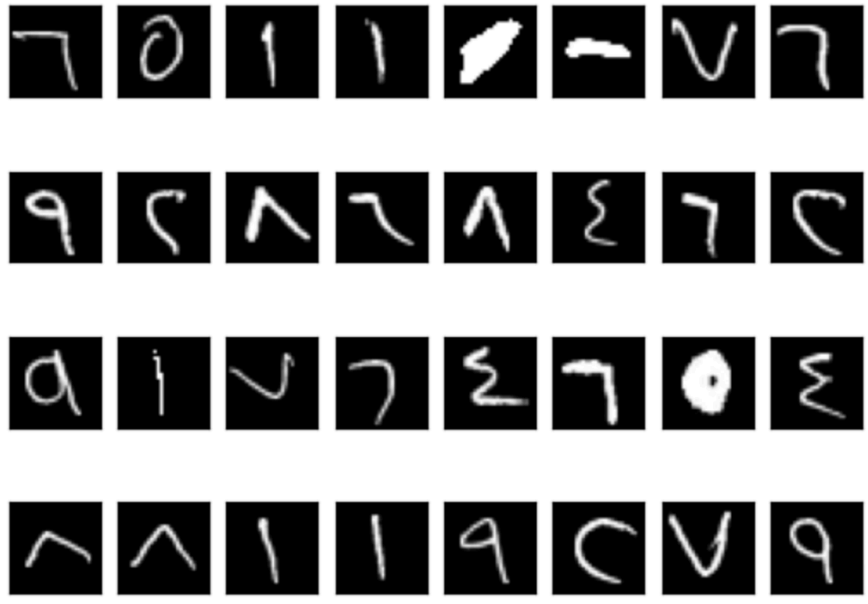
- code for processing data samples can get messy and hard to maintain
- we ideally want our dataset code to be decoupled from our model training code for better readability and modularity
- PyTorch provides two data primitives: ‘torch.utils.data.DataLoader‘ and ‘torch.utils.data.Dataset‘ that allow us to use pre-loaded datasets as well as our own data
- Dataset stores the samples and their corresponding labels
- DataLoader wraps an iterable around the Dataset to enable easy access to the samples.

Data sets and data loaders



Using our own data

- suppose we want to build an arabic digit classification system



- data is give to us in, say, two spreadsheets
  - images spreadsheet with columns containing `uint8` data
  - labels spreadsheet containing a single column of `int` data

Inheriting from the `Dataset` class

```
import pandas as pd
from torch.utils.data import Dataset

class AMNIST(Dataset):
    def __init__(self, imagesfile, labelsfile):
        super().__init__()
        df_x = pd.read_csv(imagesfile, header=None)
        df_y = pd.read_csv(labelsfile, header=None)
        self.X = df_x.iloc[:, :].to_numpy().reshape(-1,28,28) / 255.0
        self.y = df_y.iloc[:, 0].to_numpy()

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

    def __getitem__(self, idx):
        image = self.X[idx]
        image = torch.tensor(np.expand_dims(image, axis=0))
        label = torch.tensor(self.y[idx])
        return image, label
```

Data transformations

```
from torchvision import transforms

transform_train = transforms.Compose([
                        transforms.RandomCrop(28, padding=4),
                        transforms.ToTensor(),
                    ])
# Training dataset
train_data = MNIST(root='./datasets', train=True, download=True,
↪  transform=transform_train)

train_loader = torch.utils.data.DataLoader(train_data,
↪  batch_size=batch_sz, shuffle=True, pin_memory=True)
```

Guess what this does:

```
dataset = MyMNIST(image_dir, label_spreadsheet,
↪  transforms.Compose([transforms.ToTensor(),
↪  transforms.ColorJitter(...), transforms.RandomAffine(90)]))
train_loader=DataLoader(dataset, batch_size=10)

```

Saving and loading models

```
# saving/loading models
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

net = Net()
net.load_state_dict(torch.load(PATH))
```

Example of finetuning an existing network

```
from torchvision import models

net = models.resnet18(pretrained=True)
for layer in net.parameters():
  layer.requires_grad=False
net.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
↪  padding=(3, 3), bias=False)
net.fc = nn.Linear (in_features=512, out_features=10, bias=True)
```