# Tutorial 6

## Outline

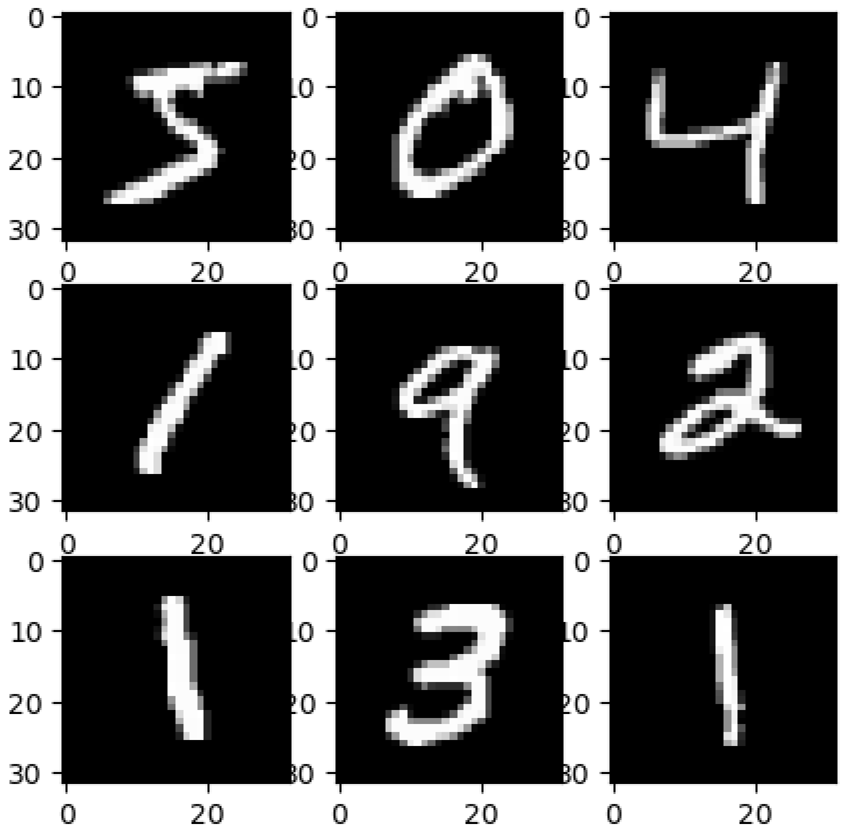
- Principal Component Analysis (PCA) in sklearn
- Dropout and L2 regularization in PyTorch
- Dataset and DataLoader in PyTorch

## MNIST Dataset

In [ ]:
```
import pickle
import torch

def load_dataset(path):
    with open(path, 'rb') as f:
        train_data, test_data = pickle.load(f)

    X_train = torch.tensor(train_data[0], dtype=torch.float)
    y_train = torch.tensor(train_data[1], dtype=torch.long)
    X_test = torch.tensor(test_data[0], dtype=torch.float)
    y_test = torch.tensor(test_data[1], dtype=torch.long)
    return X_train, y_train, X_test, y_test

X_train, y_train, X_test, y_test = load_dataset("Datasets/mnist.pkl")
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: torch.Size([60000, 32, 32])
X_test shape: torch.Size([10000, 32, 32])
y_train shape: torch.Size([60000])
y_test shape: torch.Size([10000])
```

In [ ]:
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(3, 3, figsize=(5, 5))
for i, ax in enumerate(axes.flatten()):
    ax.imshow(X_train[i], cmap='gray')
```



# Principal Component Analysis (PCA)

In [ ]:
```
# Flatten the inputs & normalization
X_train = X_train.reshape(X_train.shape[0], -1) / torch.max(X_train)
X_test = X_test.reshape(X_test.shape[0], -1) / torch.max(X_test)
print(X_train.shape)
```

```
torch.Size([60000, 1024])
```

In [ ]:
```
from sklearn.decomposition import PCA

# keeping specific number of features
pca = PCA(n_components=256)
# fit
pca.fit(X_train)
# transform
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
print(X_train_pca.shape, X_test_pca.shape)
```

```
(60000, 256) (10000, 256)
```

In [ ]:
```
# keeping amount of variance
pca = PCA(n_components=0.99)
# fit
pca.fit(X_train)
# transform
X_train_pca = torch.tensor(pca.transform(X_train), dtype=torch.float)
```

```
X_test_pca = torch.tensor(pca.transform(X_test), dtype=torch.float)
print(X_train_pca.shape, X_test_pca.shape)
```

```
torch.Size([60000, 331]) torch.Size([10000, 331])
```

In [ ]:
```
sum(pca.explained_variance_ratio_)
```

Out[ ]:
```
0.9900129424929288
```

# Dropout & L2

Use `nn.Dropout` layer:

During training, randomly zeroes some of the elements of the input tensor with probability p.

In [ ]:
```
import torch.nn as nn


class NetDropout(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(331, 100),
            nn.Dropout(p=0.1),
            nn.Sigmoid(),
            nn.Linear(100, 10),
            nn.Dropout(p=0.1),
            nn.Sigmoid()
        )

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

Set `weight_decay` to use L2 regularization.

$$\text{Loss\_L2} = \text{Loss} + \lambda \sum \theta_i^2$$

In [ ]:
```
model = NetDropout()
optimizer = torch.optim.Adam(model.parameters(), 1e-3, weight_decay=1e-5)
```

# Dataset & DataLoader

In [ ]:
```
from torch.utils.data import Dataset, DataLoader


class MnistDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
```

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

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

train_data = MnistDataset(X_train, y_train)
test_data = MnistDataset(X_test, y_test)
```

In [ ]:
```
len(train_data)
```

Out[ ]:
```
60000
```

In [ ]:
```
train_data = MnistDataset(X_train_pca, y_train)
test_data = MnistDataset(X_test_pca, y_test)
train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=128, shuffle=True)
```

In [ ]:
```
for X_batch, y_batch in train_loader:
    print(X_batch.shape, y_batch.shape)
    break
```

```
torch.Size([128, 331]) torch.Size([128])
```

# Trainer

In [ ]:
```
import numpy as np
from tqdm import tqdm
class Trainer:

    def __init__(self, model, opt_method, learning_rate, batch_size, epoch, l2):
        self.model = model

        if opt_method == "adam":
            self.optimizer = torch.optim.Adam(model.parameters(), learning_rate, we
        else:
            raise NotImplementedError("This optimization is not supported")

        self.epoch = epoch
        self.batch_size = batch_size

    def train(self, train_data, val_data, early_stop=True, verbose=True, draw_curve
        train_loader = DataLoader(train_data, batch_size=self.batch_size, shuffle=T

        train_loss_list, train_acc_list = [], []
        val_loss_list, val_acc_list = [], []
        weights = self.model.state_dict()
        lowest_val_loss = np.inf
        loss_func = nn.CrossEntropyLoss()
        for n in tqdm(range(self.epoch), leave=False):
            # enable train mode
            self.model.train()
            epoch_loss, epoch_acc = 0.0, 0.0
            for X_batch, y_batch in train_loader:
```

```
                # batch_importance is the ratio of batch_size
                batch_importance = y_batch.shape[0] / len(train_data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)

                self.optimizer.zero_grad()
                batch_loss.backward()
                self.optimizer.step()

                epoch_loss += batch_loss.detach().cpu().item() * batch_importance
                batch_acc = torch.sum(torch.argmax(y_pred, axis=1) == y_batch) / y_
                epoch_acc += batch_acc.detach().cpu().item() * batch_importance
            train_loss_list.append(epoch_loss)
            train_acc_list.append(epoch_acc)
            val_loss, val_acc = self.evaluate(val_data)
            val_loss_list.append(val_loss)
            val_acc_list.append(val_acc)

            if early_stop:
                if val_loss < lowest_val_loss:
                    lowest_val_loss = val_loss
                    weights = self.model.state_dict()

        if draw_curve:
            x_axis = np.arange(self.epoch)
            fig, axes = plt.subplots(1, 2, figsize=(10, 4))
            axes[0].plot(x_axis, train_loss_list, label="Train")
            axes[0].plot(x_axis, val_loss_list, label="Validation")
            axes[0].set_title("Loss")
            axes[0].legend()
            axes[1].plot(x_axis, train_acc_list, label='Train')
            axes[1].plot(x_axis, val_acc_list, label='Validation')
            axes[1].set_title("Accuracy")
            axes[1].legend()

        if early_stop:
            self.model.load_state_dict(weights)

        return {
            "train_loss_list": train_loss_list,
            "train_acc_list": train_acc_list,
            "val_loss_list": val_loss_list,
            "val_acc_list": val_acc_list,
        }

    def evaluate(self, data, print_acc=False):
        # enable evaluation mode
        self.model.eval()
        loader = DataLoader(data, batch_size=self.batch_size, shuffle=True)
        loss_func = nn.CrossEntropyLoss()
        acc, loss = 0.0, 0.0
        for X_batch, y_batch in loader:
            with torch.no_grad():
                batch_importance = y_batch.shape[0] / len(data)
                y_pred = self.model(X_batch)
                batch_loss = loss_func(y_pred, y_batch)
```

```
                batch_acc = torch.sum(torch.argmax(y_pred, axis=1) == y_batch) / y_
                acc += batch_acc.detach().cpu().item() * batch_importance
                loss += batch_loss.detach().cpu().item() * batch_importance
        if print_acc:
            print(f"Accuracy: {acc:.3f}")
        return loss, acc
```

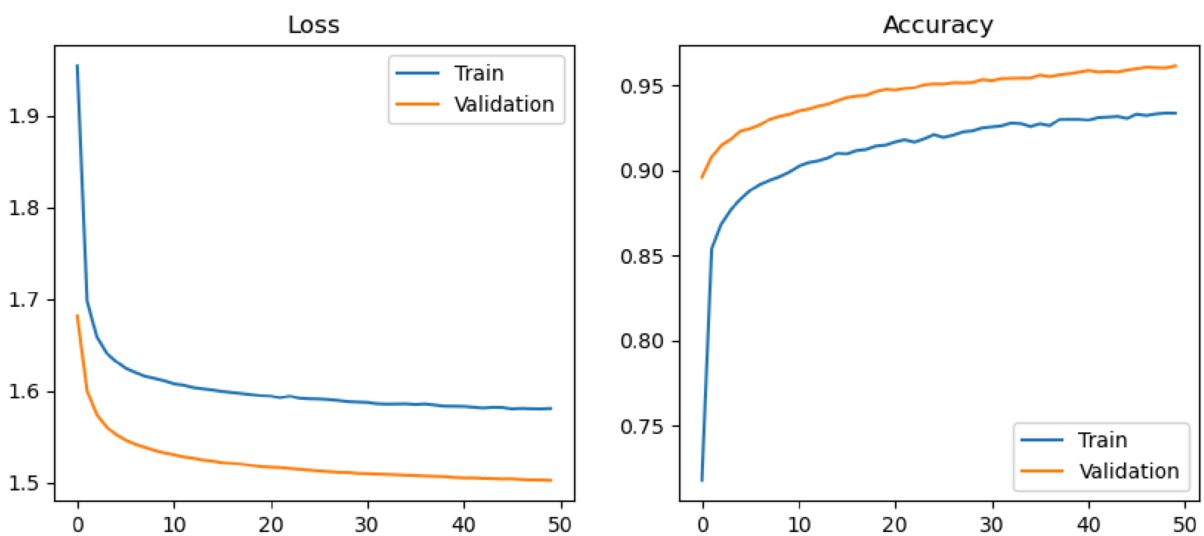
In [ ]:
```
trainer = Trainer(model, "adam", 1e-3, 128, 50, 1e-5)
trainer.train(train_data, test_data)
```

```
Out[ ]: {'train_loss_list': [1.953854073270164,
  1.6976252922058113,
  1.6583766642888382,
  1.6410647208531701,
  1.6318049741109202,
  1.6242043141682927,
  1.6194607844034818,
  1.6155706138610821,
  1.613178472391765,
  1.6106192827224728,
  1.6074101020812999,
  1.6059594261805212,
  1.603333020528158,
  1.6019683596293128,
  1.6007484980901092,
  1.5991668181737277,
  1.5979803078333548,
  1.5969474837621058,
  1.5956249532063802,
  1.5944269642511997,
  1.5939445273717245,
  1.5921851320266711,
  1.593940658060709,
  1.5914958676656084,
  1.5908963903427127,
  1.5906592164993283,
  1.5900011643091834,
  1.5890466676712023,
  1.5878049591700247,
  1.5873095454533899,
  1.5869221786499044,
  1.5855643744786592,
  1.5850921821594237,
  1.58522227935791,
  1.585398869132996,
  1.5846959139506027,
  1.585231760724385,
  1.584027119445801,
  1.5828676795323695,
  1.582756776682535,
  1.5826043802261356,
  1.5817362233479813,
  1.580856419881184,
  1.5816309234619157,
  1.5814246044794726,
  1.579927519289651,
  1.580421921793621,
  1.5800342045466114,
  1.5798836651484163,
  1.5803612693786613],
 'train_acc_list': [0.7178666666666695,
  0.8538666666984613,
  0.8685166666348825,
  0.8769833333651273,
  0.8829000000317938,
  0.8880333333333376,
```

```
 0.89161666669846,
 0.8939333333333375,
 0.8959999999682149,
 0.8986666666984596,
 0.9021166666666701,
 0.90416666666667,
 0.9052333333651258,
 0.9068999999682139,
 0.9098166666984588,
 0.9095166666666695,
 0.9115333333651253,
 0.9121166666348802,
 0.9141000000000027,
 0.9146333333333359,
 0.9165166666348799,
 0.9177166666666691,
 0.9163666666984583,
 0.918316666666669,
 0.9207499999682128,
 0.9190833333651247,
 0.9204333333015463,
 0.9222999999682129,
 0.9228833333015461,
 0.9246000000317909,
 0.9252166666666682,
 0.9257666666348793,
 0.9274166666984576,
 0.9270833333015457,
 0.9253833333333349,
 0.9269833333651242,
 0.9259166666348791,
 0.9296999999682124,
 0.9297833333015455,
 0.9297499999682122,
 0.9293000000000013,
 0.9308333333333346,
 0.9311166666984572,
 0.9315333333333347,
 0.9303666666348787,
 0.9328166666348786,
 0.9320666666666678,
 0.9330166666666676,
 0.9334833333015452,
 0.9334166666348785],
'val_loss_list': [1.6813423219680785,
 1.5996861772537232,
 1.573583877944947,
 1.5604929203033449,
 1.5518888536453244,
 1.5457319673538206,
 1.541252144241333,
 1.538062788772584,
 1.5346213188171387,
 1.532030919075012,
 1.5299967666625978,
 1.5277033630371095,
```

```
 1.5263164354324343,
 1.524244584465027,
 1.5232496160507207,
 1.5213337221145629,
 1.5206755182266238,
 1.5197183851242069,
 1.5183185609817504,
 1.5168994853973388,
 1.5162738540649412,
 1.51587453994751,
 1.514870117759705,
 1.5139693050384526,
 1.5130793724060057,
 1.5122437261581418,
 1.5114508285522463,
 1.5107213468551637,
 1.5105401058197019,
 1.5094262332916262,
 1.509212912940979,
 1.508816691207886,
 1.508539065742493,
 1.5080633462905888,
 1.5075619117736812,
 1.5072073335647582,
 1.5067236515045161,
 1.5063345474243164,
 1.5060622175216678,
 1.5051812122344976,
 1.5046455471038827,
 1.504726784133911,
 1.5041710704803475,
 1.5039700677871706,
 1.503520392608643,
 1.5036680366516118,
 1.5028916490554804,
 1.5024889087677005,
 1.5024474925994873,
 1.5020515573501583],
'val_acc_list': [0.8956999999999999,
 0.9076999999999997,
 0.9146000000000005,
 0.9182,
 0.9228000000000002,
 0.9242000000000005,
 0.9265000000000004,
 0.9297000000000003,
 0.9315000000000002,
 0.9327000000000002,
 0.9347000000000002,
 0.9357000000000006,
 0.9374000000000002,
 0.9386000000000002,
 0.9406000000000004,
 0.9424,
 0.9432999999999997,
 0.9437,
```

```
0.9458999999999996,
0.9471999999999997,
0.9467999999999999,
0.9477999999999998,
0.9482999999999998,
0.9500999999999996,
0.9505999999999998,
0.9504999999999999,
0.9512999999999999,
0.9511999999999997,
0.9513999999999997,
0.9530999999999995,
0.9524999999999997,
0.9536999999999999,
0.9537999999999996,
0.9539999999999993,
0.9539999999999994,
0.9556999999999998,
0.9547999999999998,
0.9558999999999994,
0.9564999999999995,
0.9573999999999994,
0.9583999999999993,
0.9574999999999996,
0.9577999999999999,
0.9574999999999994,
0.9585999999999997,
0.9593999999999998,
0.9602999999999995,
0.9599999999999992,
0.9598999999999996,
0.9609999999999993]}
```




```
In [ ]:
```