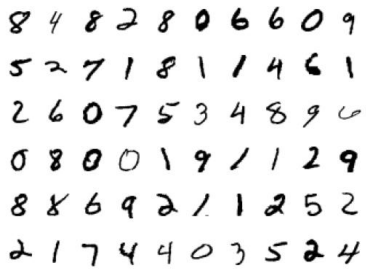
Computational MR imaging

Laboratory 10: Training neural networks with backpropagation

Nan Lan

1. Neural network Structure:

1.1 Plot an example of selected training images.



1.2 Define a fully connected neural network architecture

1) Hyperparameter setting

```python
def main():
    # hyperparameter setting
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=64, metavar='N',
                        help='input batch size for testing (default: 64)')
    parser.add_argument('--epochs', type=int, default=15, metavar='N',
                        help='number of epochs to train (default: 15)')
    parser.add_argument('--lr', type=float, default=0.001, metavar='LR',
                        help='learning rate (default: 1.0)')
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--dry-run', action='store_true', default=False,
                        help='quickly check a single pass')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=100, metavar='N',
                        help='how many batches to wait before logging training status')
```

```
    parser.add_argument('--save-model', action='store_true', default=False,
                        help='For Saving the current Model')
    args = parser.parse_args()
    use_cuda = not args.no_cuda and torch.cuda.is_available()

    torch.manual_seed(args.seed)

    device = torch.device("cuda" if use_cuda else "cpu")

    train_kwargs = {'batch_size': args.batch_size}
    test_kwargs = {'batch_size': args.test_batch_size}
    if use_cuda:
        cuda_kwargs = {'num_workers': 0,
                       'pin_memory': True,
                       'shuffle': True}
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)
```

2) Load data

```
    #load data
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
    dataset1 = datasets.MNIST('../data', train=True, download=True,
                        transform=transform)
    dataset2 = datasets.MNIST('../data', train=False,
                        transform=transform)
    train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
    test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)
```

3) Define and Instantiate model

```
    #define model
    model = Model().to(device)
    optimizer = optim.SGD(model.parameters(), lr=args.lr, momentum=0.9)
    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
    train_loss = torch.zeros(args.epochs)
    train_acc = torch.zeros(args.epochs)
    test_loss = torch.zeros(args.epochs)
    test_acc = torch.zeros(args.epochs)
```

4) Train and test

```
    # trainging and testing
    for epoch in range(1, args.epochs + 1):
        train_loss[epoch-1], train_acc[epoch-1] = train(args, model, device, train_loader, optimizer, epoch)
        test_loss[epoch-1], test_acc[epoch-1] = test(model, device, test_loader)
        scheduler.step()
```

Where the train and test function are defined as followed:

```python
# %%Train network
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    sum_loss = 0
    correct = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0), -1)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        sum_loss += loss.item()
        loss.backward()
        optimizer.step()
        pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
        correct += pred.eq(target.view_as(pred)).sum().item()

    print('Epoch{}: \nTrain set : Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
        epoch, sum_loss / len(train_loader.dataset), correct,
        len(train_loader.dataset),
        100. * correct / len(train_loader.dataset)))
    return sum_loss / len(train_loader.dataset), correct / len(train_loader.dataset)
```

```python
# %% Calculate validation accuracy
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0), -1)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'.format(
        test_loss / len(test_loader.dataset), correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
    return test_loss / len(test_loader.dataset), correct / len(test_loader.dataset)
```

2. Result and conclusion

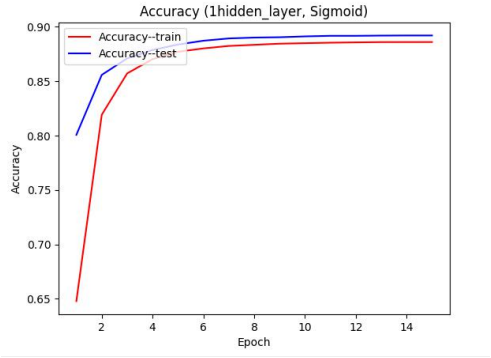
1) 1 hidden layer, Sigmoid

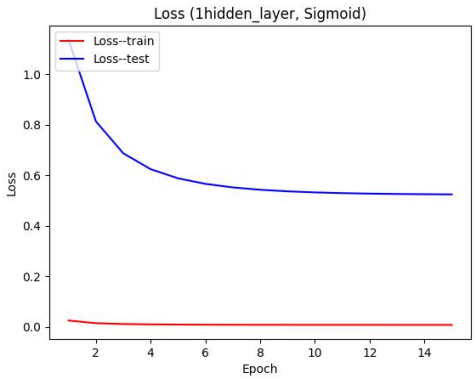
The definition of model is as followed.

```python
self.FC_1hidden = torch.nn.Sequential(torch.nn.Linear(28 * 28, 30),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(30, 10),
                                      torch.nn.LogSoftmax(dim=1))
```

There is the accuracy and loss. The classification effect is ok.

```
Epoch15:
Train set : Average loss: 0.0084, Accuracy: 53159/60000 (89%)
Test set: Average loss: 0.5250, Accuracy: 8920/10000 (89%)
```



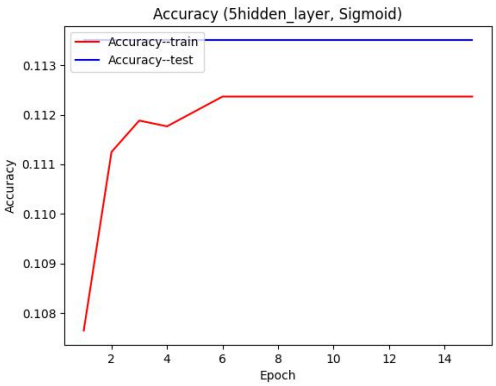


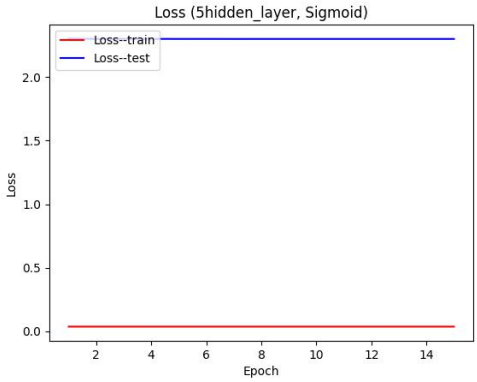
2) 5 hidden layer, Sigmoid

```
self.FC_5hidden = torch.nn.Sequential(torch.nn.Linear(28 * 28, 30),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(30, 50),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(50, 80),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(80, 100),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(100, 120),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(120, 150),
                                      torch.nn.Sigmoid(),
                                      torch.nn.Linear(150, 10),
                                      torch.nn.LogSoftmax(dim=1))
```

There is the accuracy and loss. The classification effect is super bad due to the overfitting. Batch normalization layers are added after each linear layer to adjust Internal Covariate Shift.

```
Epoch15:
Train set : Average loss: 0.0360, Accuracy: 6742/60000 (11%)
Test set: Average loss: 2.3010, Accuracy: 1135/10000 (11%)
```

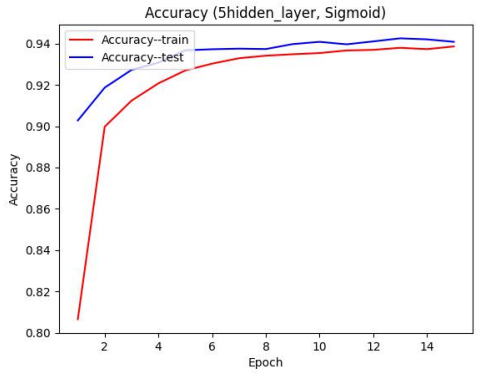


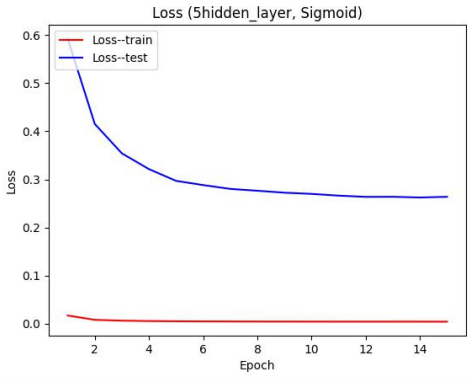


3) 5 hidden layer, Sigmoid(add Batch Normalization after each linear layer)

After adding batch normalization layers, the classification effect(94%) turns quite good (Better than the structure of 1 hidden layer with Sigmoid, 89%).

```
Epoch15:
Train set : Average loss: 0.0042, Accuracy: 56320/60000 (94%)
Test set: Average loss: 0.2638, Accuracy: 9409/10000 (94%)
```

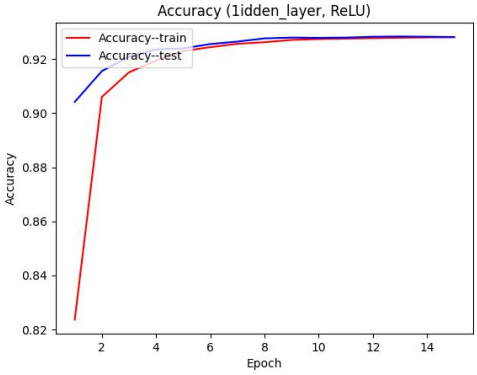


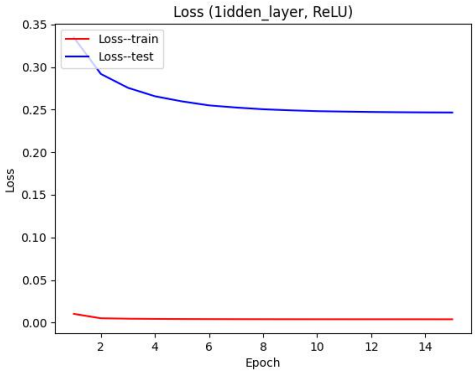


4) 1 hidden layer, ReLU

After changing the activation function from Sigmoid to ReLU, the classification effect gets better.

```
Epoch15:
Train set : Average loss: 0.0039, Accuracy: 55687/60000 (93%)
Test set: Average loss: 0.2466, Accuracy: 9282/10000 (93%)
```

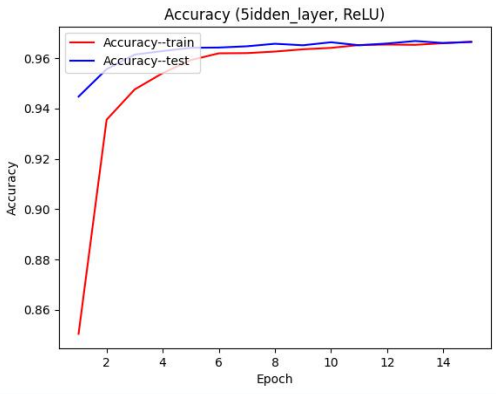


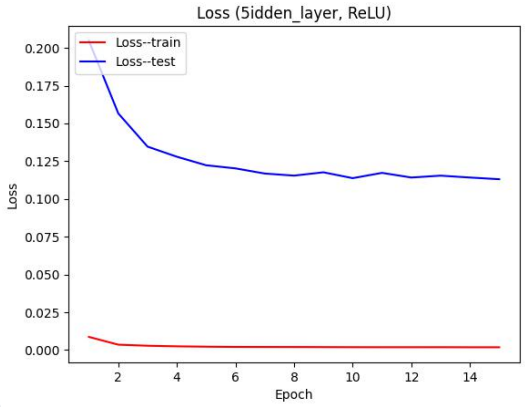


5) 5 hidden layer, ReLU(add Batch Normalization after each linear layer)

The classification effect is the best in this structure.

```
Epoch15:
Train set : Average loss: 0.0019, Accuracy: 57998/60000 (97%)
Test set: Average loss: 0.1131, Accuracy: 9665/10000 (97%)
```





The conclusion is:

Complicated network tends to get stuck in overfitting.

Under the precondition of fixing overfting, the more complicated the network is, the better the classification effect.

ReLU works better than Sigmoid function. The reason is that Sigmoid function has the problem of gradient vanishing. During backpropagation, the derivative of the sigmoid function is very small. After multiple accumulations in many layers , the gradient is close to 0, and the weight is basically not updated. So the training of the deep network cannot be completed.