Computer Vision, Homework 3: Big vs Small Models

111061548 游鎮遠

Problems

1. (30%) Finish the rest of the codes for Problem 1 and Problem 2 according to the hint. (2 code cells in total.)

Load Models

```
model_small = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', weights=None)
model_big = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', weights=None)


model_small.fc = nn.Linear(in_features=512, out_features=10, bias=True)
model_small = model_small.to(device)
model_big.fc = nn.Linear(in_features=2048, out_features=10, bias=True)
model_big = model_big.to(device)
```

Since ResNet18 and ResNet50 were originally designed for the ImageNet dataset, which has a total of 1000 output types, but this homework uses CIFAR10, which has only 10 output types, so we need to change the output layer to 10 types first.

And for ResNet50, we must modify the input features to 2048, which is different with ResNet18.

Training and Testing Models

```
def train(dataloader, model, loss_fn, optimizer):
    num_batches = len(dataloader)
    size = len(dataloader.sampler)
    epoch_loss = 0
    correct = 0
    model.train()
    print(' * training:')
    for X, y in tqdm(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        pred = pred.argmax(dim=1, keepdim=True)
        correct += pred.eq(y.view_as(pred)).sum().item()
    avg_epoch_loss = epoch_loss / num_batches
    avg_acc = correct / size
    return avg_epoch_loss, avg_acc

def test(dataloader, model, loss_fn):
    num_batches = len(dataloader)
    size = len(dataloader.sampler)
    epoch_loss = 0
    correct = 0
    model.eval()
    print(' * testing:')
    with torch.no_grad():
        for X, y in tqdm(dataloader):
            X, y = X.to(device), y.to(device)
            pred = model(X)
            epoch_loss += loss_fn(pred, y).item()
            pred = pred.argmax(dim=1, keepdim=True)
            correct += pred.eq(y.view_as(pred)).sum().item()
    avg_epoch_loss = epoch_loss / num_batches
    avg_acc = correct / size
    return avg_epoch_loss, avg_acc
```

For this part, just slightly modify the code provided in the tutorial.

2. Train small model (resnet18) and big model (resnet50) from scratch on ‘sixteenth train dataloader’, ‘half train dataloader’, and ‘train dataloader’ respectively.

- Small model (ResNet18) – train_dataloader

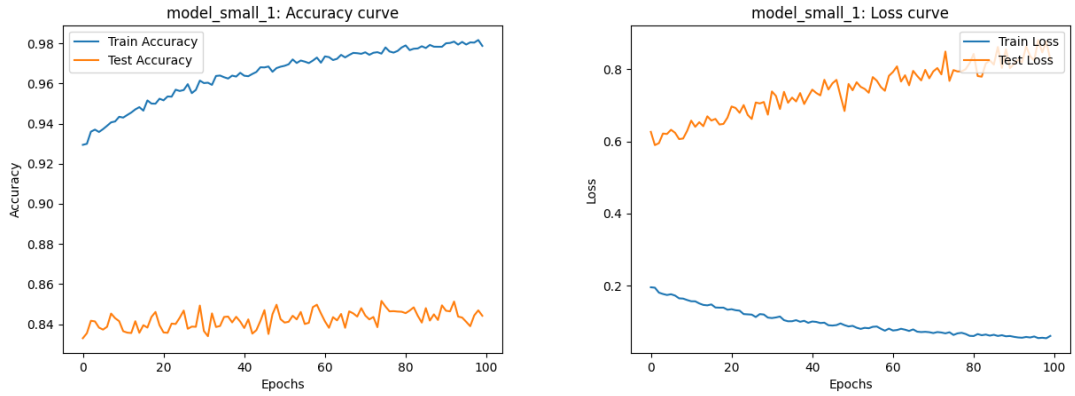

Figure 1: ResNet18 on train_dataloader

- Small model (ResNet18) – half_train_dataloader

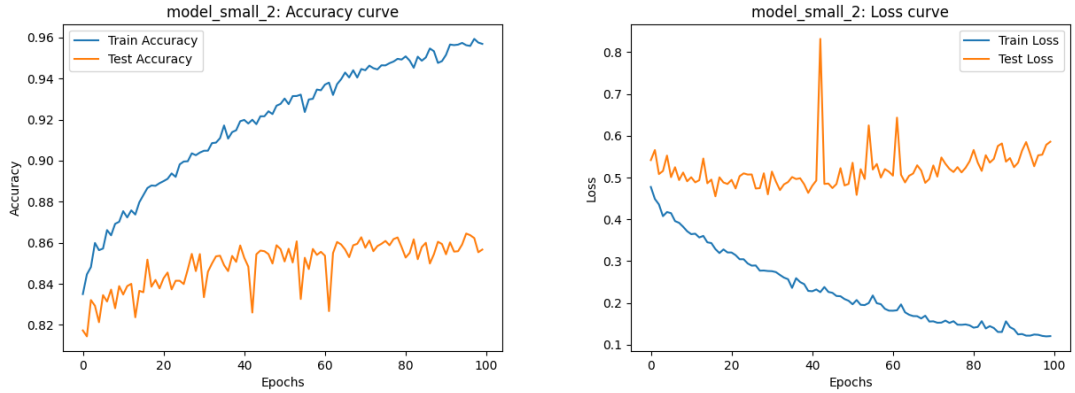

Figure 2: ResNet18 on half_train_dataloader

- Small model (ResNet18) – sixteenth_train_dataloader

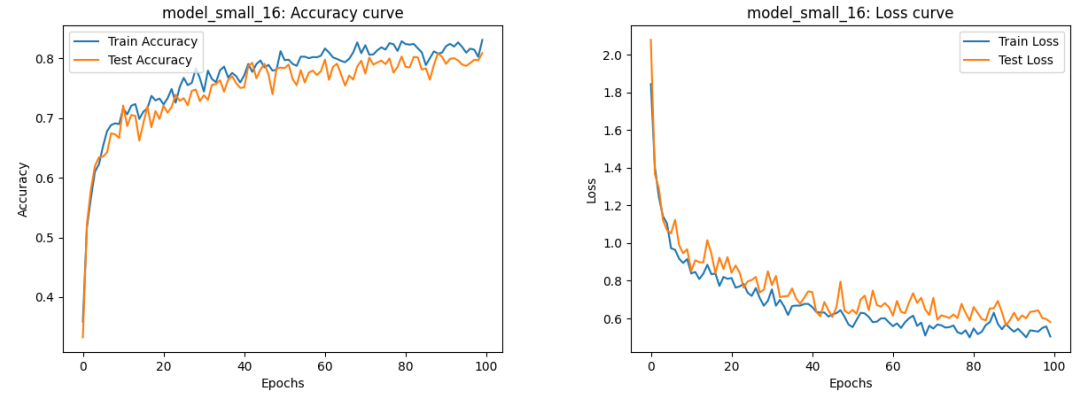

Figure 3: ResNet18 on sixteenth_train_dataloader

- Big model (ResNet18) – train_dataloader

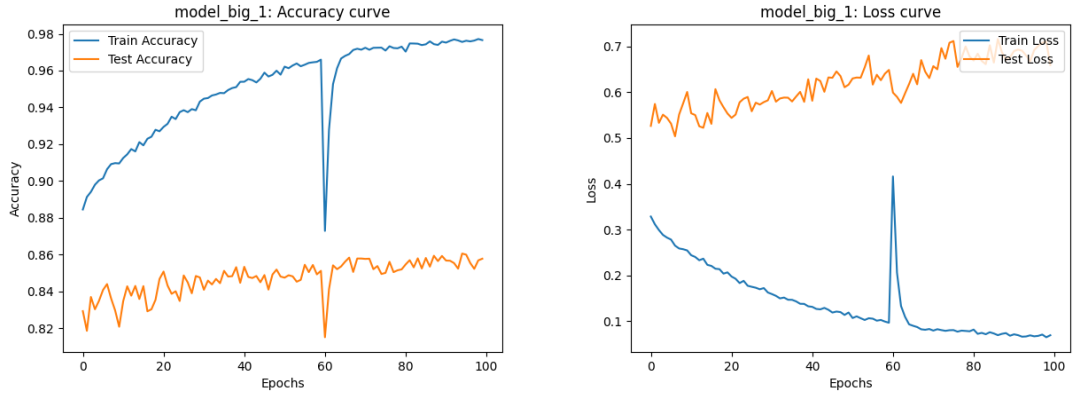

Figure 4: ResNet50 on train_dataloader

- Big model (ResNet50) – half_train_dataloader

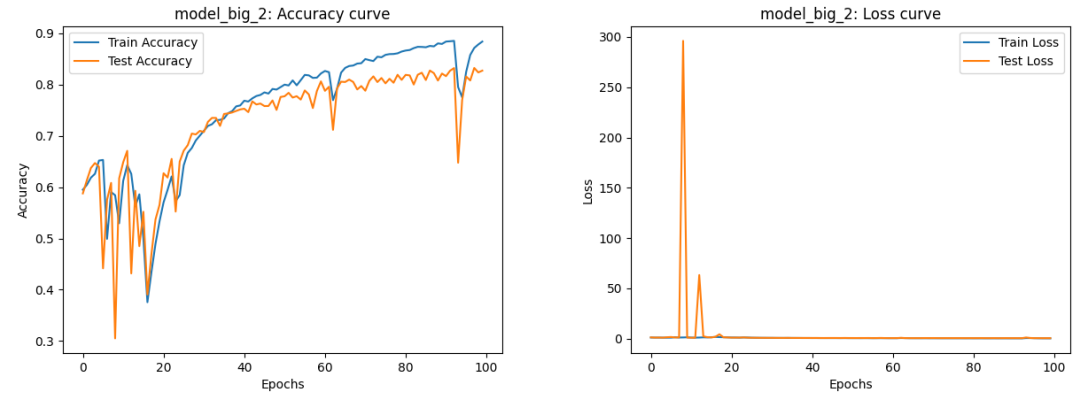

Figure 5: ResNet50 on half_train_dataloader

- Big model (ResNet50) – sixteenth_train_dataloader

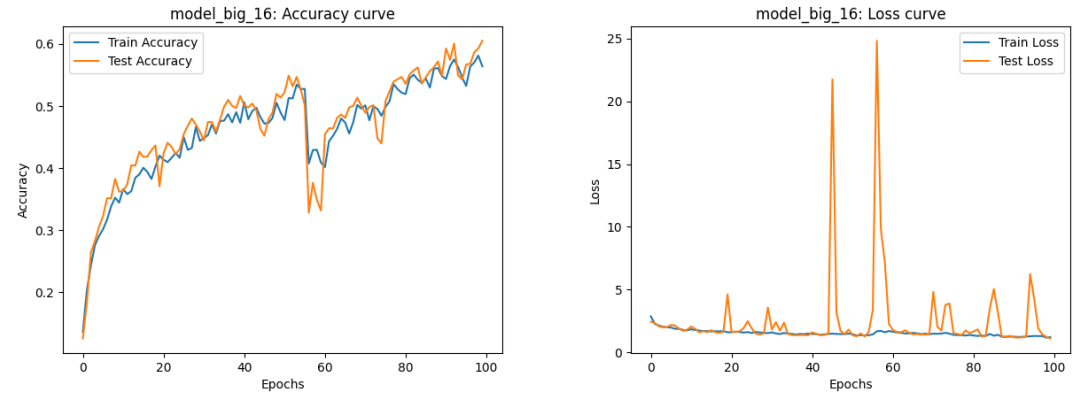

Figure 6: ResNet50 on sixteenth_train_dataloader

3. (30%) Achieve the best performance given all training data using whatever model and training strategy. (You cannot use the model that was pretrained on CIFAR10)

```
class CNN(nn.Module):
  def __init__(self, num_classes: int):
    super(CNN, self).__init__()
    self.num_classes = num_classes

    self.conv1 = nn.Sequential(
        nn.Conv2d(
        in_channels=3,
        out_channels=16,
        kernel_size=5,
        stride=1,
        padding=2),
        nn.ReLU(True),
        nn.MaxPool2d(kernel_size=2))
    self.conv2 = nn.Sequential(
        nn.Conv2d(16, 32, 5, 1, 2),
        nn.ReLU(True),
        nn.MaxPool2d(2))
    self.conv3 = nn.Sequential(
        nn.Conv2d(32, 64, 5, 1, 2),
        nn.ReLU(True),
        nn.MaxPool2d(2))
    self.fc1 = nn.Sequential(
        nn.Linear(1024, 512),
        nn.ReLU(True))
    self.fc2 = nn.Sequential(
        nn.Linear(512, 256),
        nn.ReLU(True))
    self.fc3 = nn.Sequential(
        nn.Linear(256, 128),
        nn.ReLU(True))
    self.fc4 = nn.Sequential(
        nn.Linear(128, 64),
        nn.ReLU(True))
    self.fc5 = nn.Sequential(
        nn.Linear(64, 32),
        nn.ReLU(True))
    self.fc6 = nn.Sequential(
        nn.Linear(32, 16),
        nn.ReLU(True))
    self.out = nn.Linear(16, self.num_classes)
```

```
  def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.fc2(x)
    x = self.fc3(x)
    x = self.fc4(x)
    x = self.fc5(x)
    x = self.fc6(x)
    output = self.out(x)
    return output


loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn.parameters(), lr=3e-3)
```

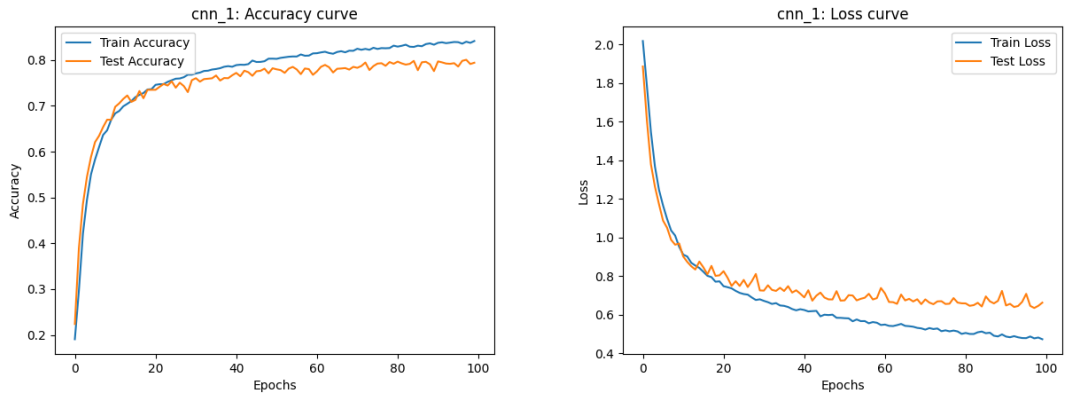

Figure 7: cnn on train_dataloader

In this part, I searched some related neural networks on the Internet and tried to modify them to achieve better results, but the results were not very satisfactory, and this also shows how powerful the classification of ResNet is.

Discussion

(30%) The relationship between the accuracy, model size, and the training dataset size. (Total 6 models. Small model trains on the sixteenth, half, and all data. Big model trains on the sixteenth, half, and all data. If the result is different from Fig. 1, please explain the possible reasons.)

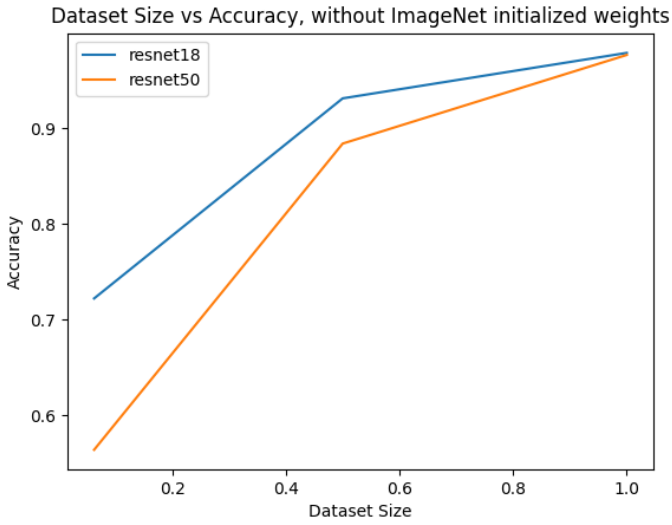

Figure 8: Dataset Size vs Accuracy, without ImageNet initialized weights

From Figure 8, it can be seen that the big model (ResNet50) does not perform as well as the small model (ResNet18) when the dataset is small, while the performance gradually catches up with the small model (ResNet18) when the dataset is larger, which may be due to the fact that the big model (ResNet50) goes through an overfitting when the dataset is small.

(10%) What if we train the ResNet with ImageNet initialized weights (weights="IMAGENET1K V1"). Please explain why the relationship changed this way?

Load Models

```
model_small_pre = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18', weights='ResNet18_Weights.IMAGENET1K_V1')
model_big_pre = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', weights='ResNet50_Weights.IMAGENET1K_V1')


model_small_pre.fc = nn.Linear(in_features=512, out_features=10, bias=True)
model_small_pre = model_small_pre.to(device)
model_big_pre.fc = nn.Linear(in_features=2048, out_features=10, bias=True)
model_big_pre = model_big_pre.to(device)
```

- Small model (ResNet18) with ImageNet initialized weights – train_dataloader

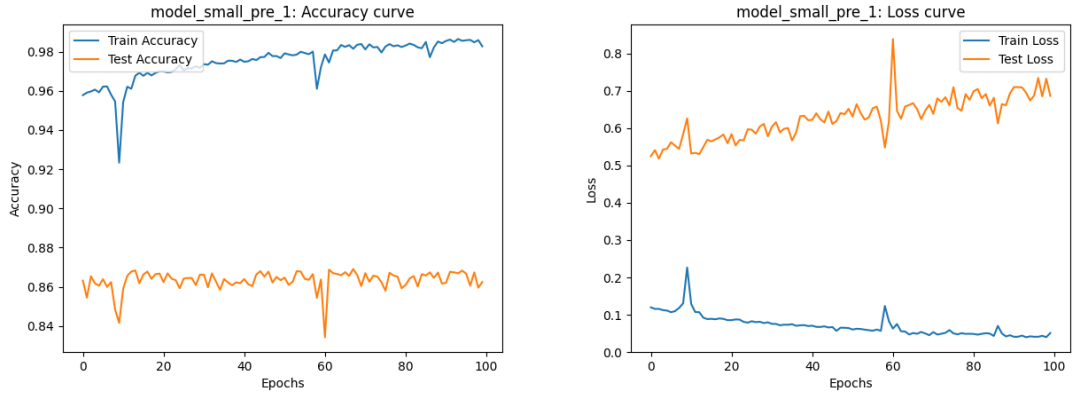

Figure 9: ResNet18 with ImageNet initialized weights on train_dataloader

- Small model (ResNet18) with ImageNet initialized weights – half_train_dataloader

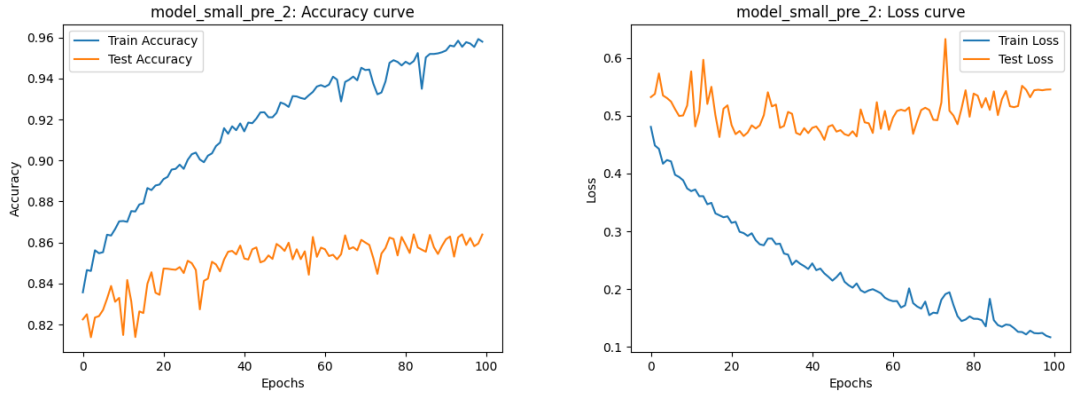

Figure 10: ResNet18 with ImageNet initialized weights on half_train_dataloader

- Small model (ResNet18) with ImageNet initialized weights – sixteenth_train_dataloader

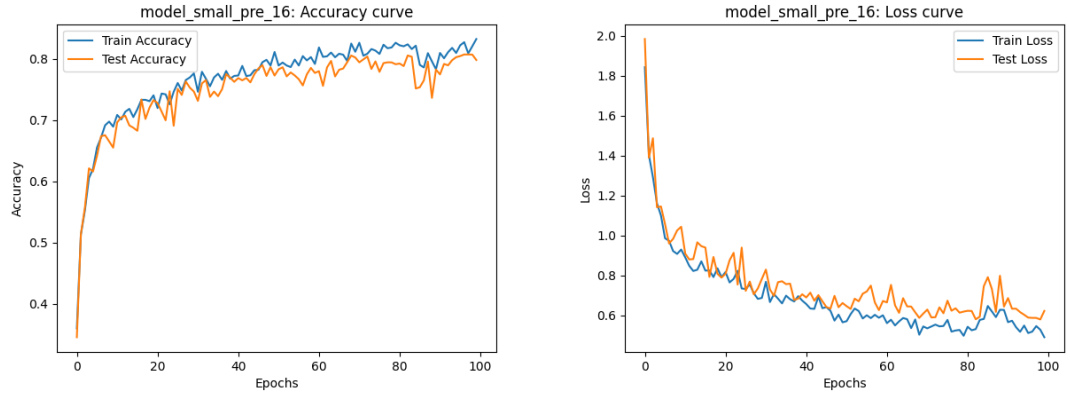

Figure 11: ResNet18 with ImageNet initialized weights on sixteenth_train_dataloader

- Big model (ResNet50) with ImageNet initialized weights – train_dataloader

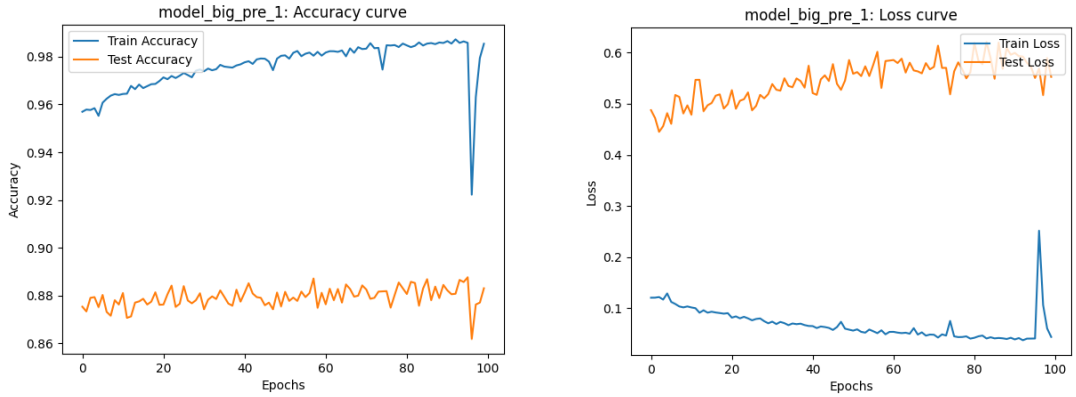

Figure 12: ResNet50 with ImageNet initialized weights on train_dataloader

- Big model (ResNet50) with ImageNet initialized weights – half_train_dataloader

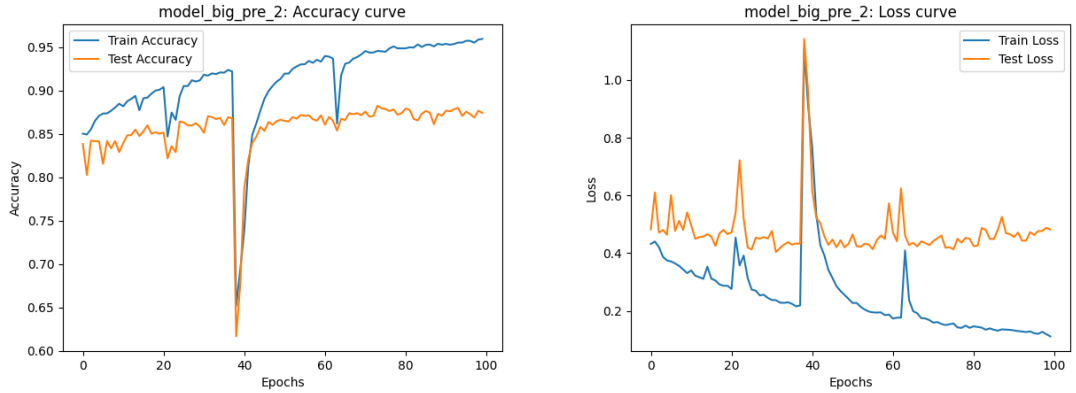

Figure 13: ResNet50 with ImageNet initialized weights on half_train_dataloader

- Big model (ResNet50) with ImageNet initialized weights – sixteenth_train_dataloader

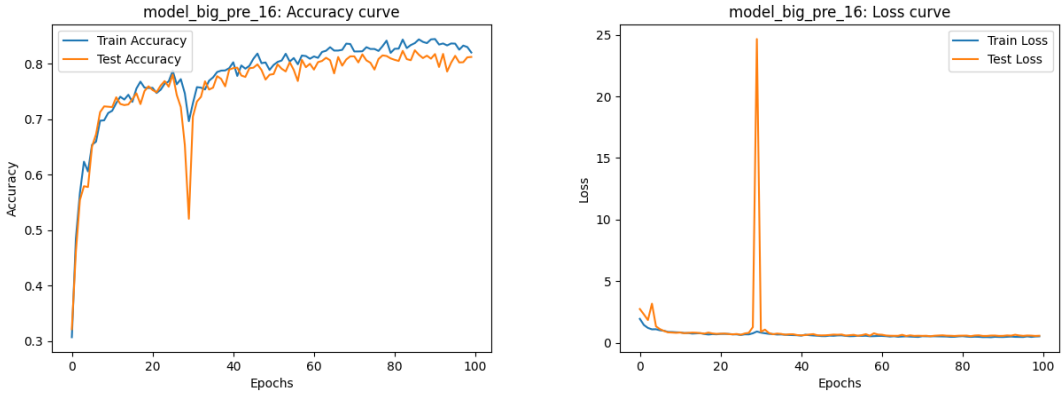

Figure 14: ResNet50 with ImageNet initialized weights on sixteenth_train_dataloader

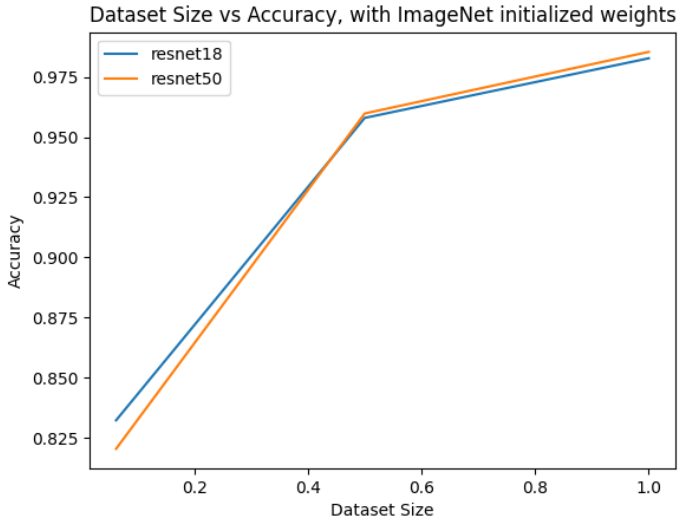

Figure 15: Dataset Size vs Accuracy, with ImageNet initialized weights

Compared to Figure 8, Figure 15 shows the results with using ImageNet initialized weights, not only the accuracy of all points is improved, but also we can observe that the big model (ResNet50) has a significant increase when using half_train_dataloader, and it is even more accurate than the small model (ResNet18) when using train_dataloader, this might because of eliminate the overfitting situation.

And this result is more similar to Figure 1 in CV_hw3.pdf.