PyTorch深度学习实战

泰迪智能科技（武汉）有限公司

讲师：骆明

目 录

contents

第一部分 使用PyTorch搭建线性回归模型

- 使用自动求导机制和简单计算函数模拟
- 调用PyTorch优化器搭建线性回归模型

使用PyTorch搭建线性回归模型

PyTorch

前面我们已经将PyTorch基础API和PyTorch自动求导机制进行了全面的讲解，接下来将通过一些实际的案例对基础API进行巩固，并以基础线性回归模型为例逐步开始使用PyTorch搭建神经网络模型。

首先，将使用PyTorch中的自动求导机制，结合梯度下降法去拟合一个线性回归模型。

然后，尝试将基础的代码替换为PyTorch中的API（如：模型、优化器、损失函数等等）重新搭建并拟合线性回归模型。

使用自动求导机制和简单计算函数模拟

PyTorch

数据构造

使用PyTorch中的随机正态分布生成100个随机数，并使用0初始化斜率和截距，由于要计算参数的梯度，所以将参数的requires_grad设置为True。

In [8]:
```
import torch
```
executed in 11ms, finished 14:25:32 2023-10-24

In [9]:
```
# 构造数据
x = torch.randn(100)
y = 3 *x + 2
```
executed in 15ms, finished 14:25:32 2023-10-24

In [10]:
```
# 初始化参数
k = torch.tensor([0.0], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)
```
executed in 6ms, finished 14:25:33 2023-10-24

前向计算预测值

```
# 迭代一次
y_pre = k*x + b
y_pre
```
executed in 14ms, finished 17:01:35 2023-10-23

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.], grad_fn=<AddBackward0>)
```

损失函数计算：损失函数使用MSE

```
lr = 0.015  # 设置学习速率
# 定义损失函数为MSE
def loss_fun(y_true, y_pre):
    return torch.sum(torch.square(y_pre-y_true))/len(y_pre)
```
executed in 13ms, finished 17:01:35 2023-10-23

```
# 计算损失函数
loss = loss_fun(y, y_pre)
loss
```
executed in 14ms, finished 17:01:35 2023-10-23

```
tensor(16.0647, grad_fn=<DivBackward0>)
```

使用自动求导机制和简单计算函数模拟

PyTorch

反向传播计算参数梯度

In [7]:
```
# 反向传播计算梯度
loss.backward()
```
executed in 28ms, finished 17:01:35 2023-10-23

In [8]:
```
# 获取梯度
print(k.grad)
print(b.grad)
```
executed in 14ms, finished 17:01:35 2023-10-23

```
tensor([-7.1164])
tensor([-5.3900])
```

使用梯度进行参数更新，更新完后将梯度置零。

```
# 更新参数
k.data -= k.grad * lr
b.data -= b.grad * lr
```
executed in 13ms, finished 17:01:35 2023-10-23

```
print(k)
print(b)
```
executed in 12ms, finished 17:01:35 2023-10-23

```
tensor([0.1067], requires_grad=True)
tensor([0.0809], requires_grad=True)
```

```
# 完成一个batch将梯度置零
k.grad.data.zero_()
b.grad.data.zero_()
```
executed in 14ms, finished 17:01:35 2023-10-23

```
tensor([0.])
```

至此，单轮迭代完成，完整训练只需将上述步骤进行循环迭代即可。

使用自动求导机制和简单计算函数模拟

PyTorch

循环迭代

```
# 迭代多次
for i in range(1000):
    y_pre = k*x + b
    # print(y_pre)
    loss = loss_fun(y, y_pre)
    loss.backward()
    # 获取梯度
    # print(k.grad)
    # print(b.grad)
    # 更新参数
    k.data -= k.grad * lr
    b.data -= b.grad * lr
    # 每训练10次打印一次记录
    if i % 100 == 0:
        print("Iter: %d, k: %.4f, b: %.4f, training loss: %.4f" %
              (i, k.item(), b.item(), loss.item()))
    k.grad.data.zero_()
    b.grad.data.zero_()
```

executed in 183ms, finished 17:01:35 2023-10-23

迭代结果展示

```
Iter: 0, k: 0.2096, b: 0.1585, training loss: 14.8915
Iter: 100, k: 2.9116, b: 1.9883, training loss: 0.0093
Iter: 200, k: 2.9956, b: 2.0024, training loss: 0.0000
Iter: 300, k: 2.9997, b: 2.0003, training loss: 0.0000
Iter: 400, k: 3.0000, b: 2.0000, training loss: 0.0000
Iter: 500, k: 3.0000, b: 2.0000, training loss: 0.0000
Iter: 600, k: 3.0000, b: 2.0000, training loss: 0.0000
Iter: 700, k: 3.0000, b: 2.0000, training loss: 0.0000
Iter: 800, k: 3.0000, b: 2.0000, training loss: 0.0000
Iter: 900, k: 3.0000, b: 2.0000, training loss: 0.0000
```

从打印出来的训练过程可以看出，迭代到400轮的时候就已经收敛，k=3.0000，b=2.0000。

调用PyTorch优化器搭建线性回归模型

前面的方法仅仅使用到了PyTorch中的自动求导机制进行参数训练，其实对于这些训练过程在PyTorch中还有很多方法可以替代。

例如：

- 更新参数部分，可以在提前实例化一个优化器，反向计算完梯度后直接使用optimizer.step()去更新参数。
- 参数的梯度置零，可以直接调用优化器的optimizer.zero_grad()方法去将所有参数的梯度全部置零。

调用PyTorch优化器搭建线性回归模型

PyTorch

自定义模型类

在PyTorch中要搭建自己的模型，需要定义模型类并继承自torch.nn.Module类，并重写forward()方法。

```
1  class SimpleLinear(nn.Module):
2      def __init__(self):
3          super(SimpleLinear, self).__init__()
4          self.lr = nn.Linear(1, 1)
5
6      def forward(self, x):
7          return self.lr(x)
```

executed in 17ms, finished 14:21:50 2023-10-24

在此处，自定义模型类中仅包含一个线性层，并且线性层的神经元数量为1。（因为这里的任务是一元线性回归）

调用PyTorch优化器搭建线性回归模型

PyTorch

实例化优化器

在PyTorch中优化器在模块torch.optim下，例如：随机梯度下降法为torch.optim.SGD().

优化器在进行实例化时，需要传入模型参数作为输入，所以在实例化优化器时需要首先实例化模型。

```
1 linear = SimpleLinear()
```

executed in 14ms, finished 14:26:50 2023-10-24

```
1 optimizer = torch.optim.SGD(linear.parameters(), lr=0.015)
```

executed in 11ms, finished 14:26:50 2023-10-24

实例化损失函数

损失函数使用的是均方误差。

```
1 loss_fun = nn.MSELoss()
```

executed in 17ms, finished 14:27:48 2023-10-24

调用PyTorch优化器搭建线性回归模型

PyTorch

模型训练

在模型训练过程中与前面稍有不同，需要将x和y进行升维处理，使用unsqueeze()方法。

```
for epoch in range(500):
    # 计算模型输出
    output = linear(x.unsqueeze(-1))
    # 计算损失值
    loss = loss_fun(y.unsqueeze(-1), output)
    # 反向传播，根据计算图计算每个参数的梯度
    loss.backward()
    # 参数更新
    optimizer.step()
    # 参数梯度归零
    optimizer.zero_grad()
    if epoch % 50 == 0:
        print('Epoch:{0}, k:{1:.4f}, b:{2:.4f}, loss:{3:.6f}'.format(epoch, linear.lr.weight.item(), linear.lr.bias.item(), loss))
```

executed in 192ms, finished 14:29:41 2023-10-24

查看训练过程

```
Epoch:0, k:-0.1556, b:-0.5560, loss:16.833160
Epoch:50, k:2.3117, b:1.3544, loss:0.902168
Epoch:100, k:2.8475, b:1.8398, loss:0.049314
Epoch:150, k:2.9658, b:1.9607, loss:0.002728
Epoch:200, k:2.9922, b:1.9905, loss:0.000152
Epoch:250, k:2.9982, b:1.9977, loss:0.000009
Epoch:300, k:2.9996, b:1.9994, loss:0.000000
Epoch:350, k:2.9999, b:1.9999, loss:0.000000
Epoch:400, k:3.0000, b:2.0000, loss:0.000000
Epoch:450, k:3.0000, b:2.0000, loss:0.000000
```

训练结果和前面效果一致。

第二部分 PyTorch模型搭建与训练模拟

- 自定义模型类
- 自定义优化器类
- 自定义损失函数类
- 模型训练函数编写

前面我们已经使用了PyTorch中内置的模型、优化器和损失函数类进行模型的训练，并最后训练好了一个简单的一元线性回归模型，但是我们对于这些API为什么可以直接组装完成模型的训练并不了解，看似一个简单的方法调用，其内部到底是如何运作的呢？

这一节，我们将带领大家使用基本的类去创建自定义模型，优化器与损失函数，并同样进行组装和模型的训练。

自定义模型类

在__init__()方法中，定义了一元线性回归模型的参数。

在forward()方法中，定义了模型进行前向计算的公式。

在parameters()方法中，需要返回模型的所有参数，这个方法在实例化优化器时会使用到。

最后__call__()方法，执行的是前向计算任务，该方法是类中的魔法方法，表示直接调用类时执行的方法。

```python
class SimpleLinear:
    # 初始化参数
    def __init__(self):
        self.k = torch.tensor([0.0], requires_grad=True)
        self.b = torch.tensor([0.0], requires_grad=True)

    # 模型前向计算
    def forward(self, x):
        y = self.k * x + self.b
        return y

    # 模型参数
    def parameters(self):
        return [self.k, self.b]

    # 调用模型自动开始计算
    def __call__(self, x):
        return self.forward(x)
```

executed in 13ms, finished 17:01:35 2023-10-23

自定义优化器类

在__init__()方法中，定义了优化器类实例化时需要传入的参数，模型参数parameters，学习率lr。

在step()方法中，定义了优化参数使用的方法，这里使用的是梯度下降法。

在zero_grad()方法中，定义了将模型的所有参数进行梯度置零操作。

```python
class Optimizer:
    # 初始化参数
    def __init__(self, parameters, lr):
        self.parameters = parameters
        self.lr = lr

    # 更新参数
    def step(self):
        for para in self.parameters:
            para.data -= para.grad * self.lr

    # 初始化梯度
    def zero_grad(self):
        for para in self.parameters:
            para.grad.zero_()
```

executed in 13ms, finished 17:01:35 2023-10-23

自定义损失函数类

在损失函数类中仅仅定义一个__call__()方法，该方法中的内容为损失函数的实现过程。

```
# 定义损失函数为MSE
class LossFun:
    def __call__(self, y_true, y_pre):
        return  torch.sum(torch.square(y_pre-y_true))/len(y_pre)
```

executed in 9ms, finished 16:02:41 2023-10-24

```
loss_fun = LossFun()
```

executed in 5ms, finished 16:02:42 2023-10-24

模型训练

模型训练

```
# 实例化模型
model = SimpleLinear()
# 实例化优化器
opt = Optimizer(model.parameters(), lr=0.015)
for epoch in range(500):
    # 计算模型输出
    output = model(x)
    # 计算损失值
    loss = loss_fun(y, output)
    # 反向传播，根据计算图计算每个参数的梯度
    loss.backward()
    # 参数更新
    opt.step()
    # 参数梯度归零
    opt.zero_grad()
    if epoch % 50 == 0:
        print('Epoch:{0}, k:{1:.4f}, b:{2:.4f}, loss:{3:.6f}'.format(epoch,
model.parameters()[0].item(), model.parameters()[1].item(), loss))
```

executed in 87ms, finished 16:02:46 2023-10-24

```
Epoch:0, k:0.0915, b:0.0525, loss:12.655186
Epoch:50, k:2.3765, b:1.4947, loss:0.657512
Epoch:100, k:2.8640, b:1.8724, loss:0.035242
Epoch:150, k:2.9699, b:1.9683, loss:0.001927
Epoch:200, k:2.9932, b:1.9922, loss:0.000107
Epoch:250, k:2.9985, b:1.9981, loss:0.000006
Epoch:300, k:2.9996, b:1.9995, loss:0.000000
Epoch:350, k:2.9999, b:1.9999, loss:0.000000
Epoch:400, k:3.0000, b:2.0000, loss:0.000000
Epoch:450, k:3.0000, b:2.0000, loss:0.000000
```

同样可以达到训练效果

第三部分

使用PyTorch实现手写数字识别任务

- 自定义数据集类
- 使用卷积神经网络实现手写数字识别任务
- 模型检查点
- 使用循环神经网络实现手写数字识别任务

数据集获取

在PyTorch中给我们内置了大量的数据集，在使用的时候可以直接导入，如果数据文件不存在会自动到PyTorch官网去下载相应的数据文件，使用的时候直接导入即可。

常见的数据集包括：

- 人脸数据集：CelebA
- 图像10分类数据集：CIFAR10
- 图像100分类数据集：CIFAR100
- 服装MNIST数据集：FashionMNIST
- 手写数字识别数据集：MNIST
- ……

PyTorch数据集官网：https://pytorch.org/vision/stable/datasets.html

数据集获取

PyTorch

在线获取

在torchvision.datasets中调用MNIST函数。

datasets.MNIST(root, train, download, transform, target_transform)

- root：数据文件所在目录。
- train：当值为True时表示获取训练集，当值为False时表示获取测试集。
- download：当值为True时，如果数据文件在root目录中不存在，就会自动去下载。
- transform：数据集图片变换操作。使用torchvision.transforms.v2.Compose定义。具体见：https://pytorch.org/vision/stable/transforms.html
- target_transform：数据集标签变换操作。使用torchvision.transforms.v2.Compose定义。

```
mnist_train_dataset = datasets.MNIST('../data/MNIST',
                                     train=True,
                                     download=True,
                                     transform=transforms.Compose([
                                                  transforms.ToTensor(),
                                                  transforms.Normalize((0.1307,),(0.3081,))
                                     ]))
```

executed in 47ms, finished 17:01:36 2023-10-23

```
mnist_test_dataset = datasets.MNIST('../data/MNIST',
                                    train=False,
                                    download=True,
                                    transform=transforms.Compose([
                                                 transforms.ToTensor(),
                                                 transforms.Normalize((0.1307,),(0.3081,))
                                    ]))
```

executed in 30ms, finished 17:01:37 2023-10-23

自定义Dataset

基本格式如右图所示，要实现自己的Dataset类，需要继承自torch.utils.data.Dataset类，并重写三个方法：

1. __init__(self)：初始化方法，在该方法中定义了类在实例化时需要传入的参数，以及生成完整数据的基本逻辑。
2. __len__(self)：该方法需要返回整个数据量的长度。
3. __getitem__(self, idx)：该方法需要返回对应索引的数据和标签。实例化该类后，能够使用索引获取到具体的实际数据。

```python
# 基本格式
class MyDataset(Dataset):
    def __init__(self):
        pass

    def __len__(self):
        # 返回数据长度
        pass
        return len(img)

    def __getitem__(self, idx):
        # 根据idx索引返回对应的数据和标签
        pass
        return image, label
```

executed in 16ms, finished 16:40:31 2023-10-24

自定义Dataset

在该方法的初始化方法中我们定义的3个参数：

mnist_npa_filepath：npz数据路径。

train：当该值为True时返回训练数据，当该值为False时返回测试数据。

transforms：图片转换函数。

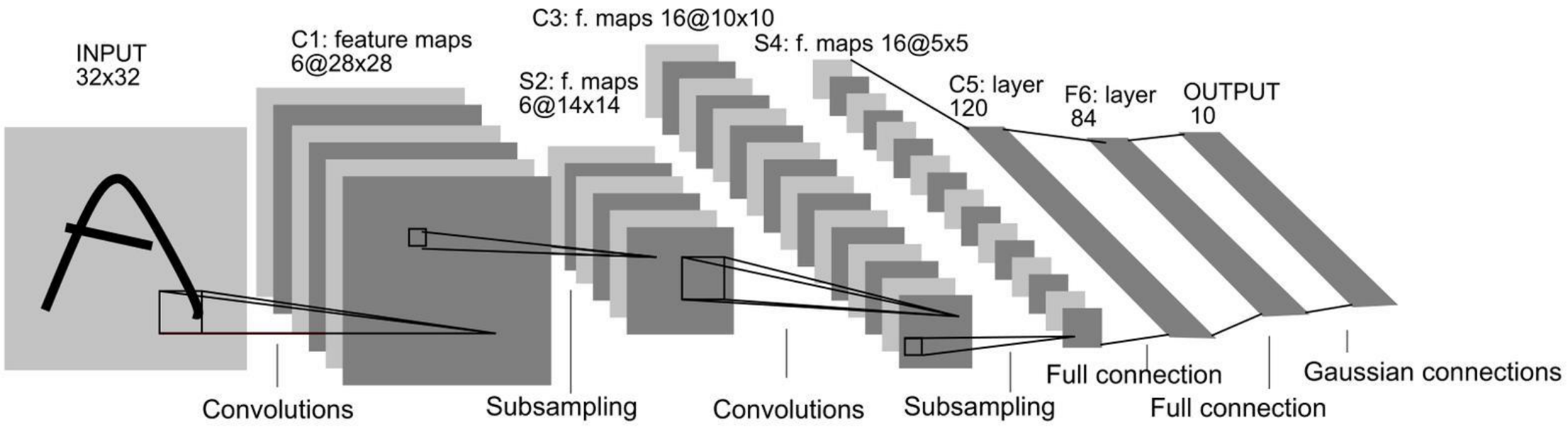
但是需要注意的时，我们在transforms中会引入ToTensor()方法，将数据转化为张量，该方法在处理的时候要求输入的数据为[W, H, C]格式。转化为张量后会处理为[C, W, H]格式。所以在对图片数据进行处理时，增加的通道维度增加到最后一个维度。

```python
class MNIST_Dataset(Dataset):
    def __init__(self, mnist_npz_filepath, train=True, transforms=None):
        super(MNIST_Dataset, self).__init__()
        mnist_data = np.load(mnist_npz_filepath)
        x_train = mnist_data['x_train']
        x_test = mnist_data['x_test']
        self.x = np.expand_dims(x_train, -1)
        self.y = mnist_data['y_train']

        if not train:
            self.x = np.expand_dims(x_test, -1)
            self.y = mnist_data['y_test']
        self.transforms = transforms

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

    def __getitem__(self, index):
        x = self.x[index]
        y = self.y[index]
        if self.transforms:
            # 图像处理预处理函数，要求输入为(channel, w, h)
            x = self.transforms(x)
        return x, y
```

executed in 19ms, finished 16:38:09 2023-10-24

卷积神经网络搭建

在此，搭建的卷积神经网络结构为LeNet-5，模型架构图如下所示。



卷积神经网络搭建

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, 1)
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(6, 16, 5, 1)
        self.dropout = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)

    # 重写forward方法
    def forward(self, x):
        x = self.conv1(x)  # 卷积
        x = self.pool(x)  # 池化
        x = F.relu(x)  # relu函数激活
        x = self.conv2(x)  # 卷积
        x = self.pool(x)  # 池化
        x = F.relu(x)  # relu函数激活
        x = torch.flatten(x, 1)  # 展平
        x = self.fc1(x)  # 全连接
        x = F.relu(x)  # relu函数激活
        x = self.dropout(x)  # dropout层
        x = self.fc2(x)  # relu函数激活
        return x
```

executed in 14ms, finished 17:01:37 2023-10-23

可以使用torchsummary库（pip install torchsummary）去打印模型详细结构。

In [31]:
```python
model = Net()
# 查看模型结构
summary(model.to('cuda'), (1, 28,28))
```

executed in 4.26s, finished 17:01:41 2023-10-23

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 24, 24]             156
         MaxPool2d-2            [-1, 6, 12, 12]               0
            Conv2d-3            [-1, 16, 8, 8]           2,416
         MaxPool2d-4            [-1, 16, 4, 4]               0
            Linear-5                  [-1, 128]          32,896
         Dropout2d-6                  [-1, 128]               0
            Linear-7                   [-1, 10]           1,290
================================================================
Total params: 36,758
Trainable params: 36,758
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.04
Params size (MB): 0.14
Estimated Total Size (MB): 0.19
----------------------------------------------------------------
```

定义训练函数

该训练函数接收的参数有6个：

1. model：实例化好的模型。
2. device：模型训练的设备。
3. train_loader：模型训练数据加载器。
4. criterion：模型损失函数。
5. optimizer：模型优化器。
6. epoch：模型当前训练的轮数。

在模型训练函数中需要在第一行去声明模型处于训练模式（model.train()），此时模型在进行前向计算时会计算参数的梯度。

```python
# model：模型  device：模型训练场所  optimizer：优化器  epoch：模型训练轮次
def train(model, device, train_loader, criterion, optimizer, epoch):
    model.train() # 声明训练函数，参数的梯度要更新
    total = 0 # 记录已经训练的数据个数
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        total += len(data)
        progress = math.ceil(batch_idx / len(train_loader) * 50)
        print("\rTrain epoch %d: %d/%d, [%-51s] %d%%" %
              (epoch, total, len(train_loader.dataset),
               '-' * progress + '>', progress * 2), end='')
```

executed in 15ms, finished 17:01:41 2023-10-23

定义测试函数

该测试函数接收的参数有4个：

1. model：训练好的模型。
2. device：模型测试的设备。
3. test_loader：模型测试数据加载器。
4. criterion：模型损失函数。

在模型测试函数中需要在第一行去声明模型处于测试模式或验证模型（model.eval()），此时模型在进行前向计算时不会计算参数的梯度，提高计算效率。

```python
def test(model, device, test_loader, criterion):
    model.eval()  # 声明验证函数，禁止所有梯度进行更新
    test_loss = 0
    correct = 0
    # 强制后面的计算不生成计算图，加快测试效率
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()  # 对每个batch的loss进行求和
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)

    print('\nTest: average loss: {:.4f}, accuracy: {}/{} ({:.0f}%)'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

executed in 14ms, finished 17:01:41 2023-10-23

模型训练

模型训练设置的超参数如下所示：

epochs：迭代次数。

batch_size：批处理数量。

torch.manual_seed(2021)：模型种子数。

device：模型运行设备。

train_loader：训练数据加载器。

test_loader：测试数据加载器

```
epochs = 10  # 迭代次数
batch_size = 256
torch.manual_seed(2021)

# 查看GPU是否可用，如果可用就用GPU否则用CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 训练集的定义
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
# 测试集的定义
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)
# 模型定义并加载至GPU

model = Net().to(device)
# 随机梯度下降
optimizer = torch.optim.SGD(model.parameters(), lr=0.025, momentum=0.9)
criterion = nn.CrossEntropyLoss()

for epoch in range(1, epochs+1):
    train(model, device, train_loader, criterion, optimizer, epoch)
    test(model, device, test_loader, criterion)
    print('-------------------------')
```

executed in 1m 4.63s, finished 17:02:46 2023-10-23

模型训练结果

```
Train epoch 1: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0003, accuracy: 9756/10000 (98%)
-------------------------
Train epoch 2: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0002, accuracy: 9803/10000 (98%)
-------------------------
Train epoch 3: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0002, accuracy: 9856/10000 (99%)
-------------------------
Train epoch 4: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9875/10000 (99%)
-------------------------
Train epoch 5: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9894/10000 (99%)
Train epoch 6: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9891/10000 (99%)
-------------------------
Train epoch 7: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9881/10000 (99%)
-------------------------
Train epoch 8: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9898/10000 (99%)
-------------------------
Train epoch 9: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9909/10000 (99%)
-------------------------
Train epoch 10: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9905/10000 (99%)
```

模型参数查看

```
for i,v in model.named_parameters():
    print(i, v.shape)
```

```
conv1.weight torch.Size([6, 1, 5, 5])
conv1.bias torch.Size([6])
conv2.weight torch.Size([16, 6, 5, 5])
conv2.bias torch.Size([16])
fc1.weight torch.Size([128, 256])
fc1.bias torch.Size([128])
fc2.weight torch.Size([10, 128])
fc2.bias torch.Size([10])
```

模型检查点

随着模型结构的复杂化，数据量的复杂化，模型的训练所消耗的时间也会逐渐增大，这个时候就需要注意，如果我们的模型训练一半，由于一些不可控因素导致训练中断（例如：服务器资源竞争导致程序被杀死、意外断电等等），那么此时如果再从头开始训练会导致前面训练所消耗的时间和资源白白浪费掉了。

为了应对这种情况的发生，模型检查点技术就应运而生，模型检查点是在模型在每轮的训练过程中，存储模型训练中间状态的一种技术。 模型检查点一般会保存以下几个指标：

1. epoch：当前训练的轮数
2. step：当前轮数对应的批次数（使用频率低）
3. model_state_dict：模型参数
4. optimizer_state_dict：优化器参数
5. loss：当前模型参数损失值

模型检查点存储

```
save_file = f'checkpoint_{epoch}.pt'
torch.save({
    'epoch': epoch, # 存储当前训练轮数，常用
    'step': step, # 存储此轮训练的遍历数，该参数就非常具体，具体到某批训练样本，一般用的很少，这个参数如果引用了，那么在训练过程中每个批次都需要去比较该批数据优化后模型是否更优，更优的话然后就保存
    'model_state_dict': model.state_dict(), # 模型参数，必须保存
    'optimizer_state_dict': optimizer.state_dict(), # 优化器参数，必须保存
    'loss': loss, # 此轮的损失值，用的也相对较少，每轮也会计算出新的值
}, save_file)
```

模型检查点存储

```
resume = f'checkpoint_{epoch}.pt'  # 指定恢复文件
if resume != "":
    #  加载之前训过的模型的参数文件
    print(f"loading from {resume}")
    checkpoint = torch.load(resume, map_location=torch.device("cuda:0")) #可以是cpu,cuda,cuda:index
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    start_step = checkpoint['step']
    loss = checkpoint['loss']
```

将模型检查点引入训练过程

检查点的引入一般有以下两种方式：

1. 固定轮数保存，例如：每5轮训练保存一次检查点。
2. 指标监测保存，例如：监测loss值，当loss值有下降时存储一次，也即存储最优模型。

将模型检查点引入训练过程

修改测试函数，为了比较模型效果是否提高，将测试的损失值返回。

```python
def test(model, device, test_loader, criterion):
    model.eval() # 声明验证函数，禁止所有梯度进行更新
    test_loss = 0
    correct = 0
    # 强制后面的计算不生成计算图，加快测试效率
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item() # 对每个batch的loss进行求和
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)

    print('\nTest: average loss: {:.4f}, accuracy: {}/{} ({:.0f}%)'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
    return test_loss # 将测试损失返回
```

executed in 10ms, finished 17:03:06 2023-10-23

将模型检查点引入训练过程

为了比较模型是否更优，使用的指标是测试集的损失值，初始状态下是正无穷大(torch.inf)，比较每轮测试集的损失值是否更优，如果更优则保存最优模型，并存储当前损失值。

```
epochs = 10
batch_size = 256
torch.manual_seed(2021)

# 查看GPU是否可用，如果可用就用GPU否则用CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 训练集的定义
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
# 测试集的定义
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

# 模型定义并加载至GPU
model = Net().to(device)
# 随机梯度下降
optimizer = torch.optim.SGD(model.parameters(), lr=0.025, momentum=0.9)
criterion = nn.CrossEntropyLoss()

min_loss = torch.inf
start_epoch = 0
delta = 1e-4
```

```
for epoch in range(start_epoch + 1, start_epoch + epochs+1):
    train(model, device, train_loader, criterion, optimizer, epoch)
    loss = test(model, device, test_loader, criterion)
    if loss < min_loss and not torch.isclose(torch.tensor([min_loss]),
torch.tensor([loss]), delta): # 监测loss，当loss下降时保存模型，敏感度设置为
1e-4（默认为1e-5）
        print(f'Loss Reduce {min_loss} to {loss}')
        min_loss = loss
        save_file = f'cnn_checkpoint_best.pt'
        torch.save({
            'epoch': epoch+1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss
        }, save_file)
        print(f'Save checkpoint to {save_file}')
    print('-------------------------')
```

executed in 1m 5.58s, finished 17:04:54 2023-10-23

PyTorch

模型训练结果

```
Train epoch 1: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0003, accuracy: 9757/10000 (98%)
Loss Reduce inf to 0.00031736000403761864
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 2: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0002, accuracy: 9804/10000 (98%)
Loss Reduce 0.00031736000403761864 to 0.0002453371422481723
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 3: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0002, accuracy: 9859/10000 (99%)
Loss Reduce 0.0002453371422481723 to 0.00018928093751892448
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 4: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9873/10000 (99%)
Loss Reduce 0.00018928093751892448 to 0.00014866217281669377
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 5: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9892/10000 (99%)
Loss Reduce 0.00014866217281669377 to 0.00013459483720362186
Save checkpoint to cnn_checkpoint_best.pt
Train epoch 6: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9881/10000 (99%)
-------------------------
Train epoch 7: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9885/10000 (99%)
-------------------------
Train epoch 8: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9896/10000 (99%)
Loss Reduce 0.00013459483720362186 to 0.00012158382774214260
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 9: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9909/10000 (99%)
Loss Reduce 0.00012158382774214260 to 0.00011229447936639190
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 10: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9893/10000 (99%)
```

加载检查点继续训练模型

加载检查点，将检查点中的模型参数，优化器参数，上个模型结束的epoch等加载到训练过程。

```python
epochs = 10
batch_size = 256
torch.manual_seed(2021)

# 查看GPU是否可用，如果可用就用GPU否则用CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 训练集的定义
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
# 测试集的定义
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

# 模型定义并加载至GPU
model = Net().to(device)
# 随机梯度下降
optimizer = torch.optim.SGD(model.parameters(), lr=0.025, momentum=0.9)
criterion = nn.CrossEntropyLoss()

min_loss = torch.inf
start_epoch = 0
delta = 1e-4

# 指定检查点
resume = f'cnn_checkpoint_best.pt'
# 加载检查点
if resume:
    print(f'loading from {resume}')
    checkpoint = torch.load(resume, map_location=torch.device("cuda:0"))
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    min_loss = checkpoint['loss']

for epoch in range(start_epoch + 1, start_epoch + epochs+1):
    train(model, device, train_loader, criterion, optimizer, epoch)
    loss = test(model, device, test_loader, criterion)
    if loss < min_loss and not torch.isclose(torch.tensor([min_loss]),
torch.tensor([loss]), delta): # 监测loss，当loss下降时保存模型，敏感度设置为
1e-4（默认为1e-5）
        print(f'Loss Reduce {min_loss} to {loss}')
        min_loss = loss
        save_file = f'cnn_checkpoint_best.pt'
        torch.save({
            'epoch': epoch+1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss
        }, save_file)
        print(f'Save checkpoint to {save_file}')
    print('-------------------------')
```

模型训练结果

```
loading from cnn_checkpoint_best.pt
Train epoch 11: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9904/10000 (99%)
-------------------------
Train epoch 12: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9893/10000 (99%)
-------------------------
Train epoch 13: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9915/10000 (99%)
-------------------------
Train epoch 14: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9918/10000 (99%)
Loss Reduce 0.0001122944793663919 to 0.00010782728097401559
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 15: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9908/10000 (99%)
-------------------------
Train epoch 16: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9915/10000 (99%)
-------------------------
Train epoch 17: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9905/10000 (99%)
-------------------------
Train epoch 18: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9928/10000 (99%)
Loss Reduce 0.00010782728097401559 to 0.00010413678948243614
Save checkpoint to cnn_checkpoint_best.pt
-------------------------
Train epoch 19: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9922/10000 (99%)
-------------------------
Train epoch 20: 60000/60000, [-------------------------------------------->] 100%
Test: average loss: 0.0001, accuracy: 9919/10000 (99%)
```

数据集DataSet类重构

RNN要求数据输入的格式为3维：[batch_size, sequence_length, hidden_size]，所以在数据层面不需要升维。

```python
class MNIST_Dataset(Dataset):
    def __init__(self, mnist_npz_filepath, train=True, transforms=None):
        super(MNIST_Dataset, self).__init__()
        mnist_data = np.load(mnist_npz_filepath)
        x_train = mnist_data['x_train']
        x_test = mnist_data['x_test']
        self.x = x_train
        self.y = mnist_data['y_train']

        if not train:
            self.x = x_test
            self.y = mnist_data['y_test']
        self.transforms = transforms

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

    def __getitem__(self, index):
        x = self.x[index]
        y = self.y[index]
        return torch.Tensor(x), y
```

executed in 17ms. finished 17:06:17 2023-10-23

使用循环神经网络实现手写数字识别任务

PyTorch

循环神经网络搭建

在此搭建的是最简单的循环神经网络，然后将hn传入到全连接层得到输出。

```python
class RNNNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNNet, self).__init__()
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True, nonlinearity='relu')
        self.fc = nn.Linear(hidden_dim, output_dim)

    # 重写forward方法
    def forward(self, x):
        out, x = self.rnn(x)
        x = self.fc(x)
        return x.squeeze()
```

executed in 18ms, finished 17:06:22 2023-10-23

```python
input_dim = 28
hidden_dim = 64
layer_dim = 1
output_dim = 10
rnn_model = RNNNet(input_dim, hidden_dim, layer_dim, output_dim)
```

executed in 14ms, finished 17:06:23 2023-10-23

```python
summary(rnn_model.to('cuda'), (28, 28))
```

executed in 364ms, finished 17:06:24 2023-10-23

```
----------------------------------------------------------------
      Layer (type)               Output Shape         Param #
================================================================
            RNN-1  [[-1, 28, 64], [-1, 2, 64]]               0
          Linear-2              [-1, 2, 10]             650
================================================================
Total params: 650
Trainable params: 650
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 1.75
Params size (MB): 0.00
Estimated Total Size (MB): 1.76
----------------------------------------------------------------
```

循环神经网络训练

```python
input_dim = 28
hidden_dim = 128
layer_dim = 1
output_dim = 10
epochs = 10
batch_size = 256
torch.manual_seed(2021)

# 查看GPU是否可用，如果可用就用GPU否则用CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 训练集的定义
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
# 测试集的定义
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

# 模型定义并加载至GPU
rnn_model = RNNNet(input_dim, hidden_dim, layer_dim, output_dim).to(device)
# 随机梯度下降
optimizer = torch.optim.SGD(rnn_model.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()
```

```python
min_loss = torch.inf
start_epoch = 0
delta = 1e-4

for epoch in range(start_epoch + 1, start_epoch + epochs+1):
    train(rnn_model, device, train_loader, criterion, optimizer, epoch)
    loss = test(rnn_model, device, test_loader, criterion)
    if loss < min_loss and not torch.isclose(torch.tensor([min_loss]),
torch.tensor([loss]), delta):  # 监测loss，当loss下降时保存模型，敏感度设置为1e-4（默认为1e-5）
        print(f'Loss Reduce {min_loss} to {loss}')
        min_loss = loss
        save_file = f'rnn_checkpoint_best.pt'
        torch.save({
            'epoch': epoch+1,
            'model_state_dict': rnn_model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss
        }, save_file)
        print(f'Save checkpoint to {save_file}')
    print('-------------------------')
```

executed in 14.6s, finished 17:06:41 2023-10-23

模型训练结果

```
Train epoch 1: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0049, accuracy: 5402/10000 (54%)
Loss Reduce inf to 0.00494295916557312
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 2: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0040, accuracy: 6309/10000 (63%)
Loss Reduce 0.00494295916557312 to 0.003991533029079437
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 3: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0028, accuracy: 7532/10000 (75%)
Loss Reduce 0.003991533029079437 to 0.0027526606380939484
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 4: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0020, accuracy: 8298/10000 (83%)
Loss Reduce 0.0027526606380939484 to 0.001968709048628807
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 5: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0014, accuracy: 8915/10000 (89%)
Loss Reduce 0.001968709048628807 to 0.0014009273216128348
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 6: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0012, accuracy: 9072/10000 (91%)
Loss Reduce 0.0014009273216128348 to 0.0012339959993958472
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 7: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0010, accuracy: 9254/10000 (93%)
Loss Reduce 0.0012339959993958472 to 0.0009584869503974914
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 8: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0010, accuracy: 9277/10000 (93%)
-------------------------
Train epoch 9: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0007, accuracy: 9485/10000 (95%)
Loss Reduce 0.0009584869503974914 to 0.0007289471238851547
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 10: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0010, accuracy: 9282/10000 (93%)
```

加载检查点继续训练模型

```python
input_dim = 28
hidden_dim = 128
layer_dim = 1
output_dim = 10
epochs = 10
batch_size = 256
torch.manual_seed(2021)

# 查看GPU是否可用，如果可用就用GPU否则用CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 训练集的定义
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
# 测试集的定义
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

# 模型定义并加载至GPU
rnn_model = RNNNet(input_dim, hidden_dim, layer_dim, output_dim).to(device)
# 随机梯度下降
optimizer = torch.optim.SGD(rnn_model.parameters(), lr=0.001, momentum=0.9)
criterion = nn.CrossEntropyLoss()

min_loss = torch.inf
start_epoch = 0
delta = 1e-4
```

```python
# 指定检查点
resume = f'rnn_checkpoint_best.pt'
if resume:
    print(f'loading from {resume}')
    checkpoint = torch.load(resume, map_location=torch.device("cuda:0"))
    rnn_model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    min_loss = checkpoint['loss']

for epoch in range(start_epoch + 1, start_epoch + epochs+1):
    train(rnn_model, device, train_loader, criterion, optimizer, epoch)
    loss = test(rnn_model, device, test_loader, criterion)
    if loss < min_loss and not torch.isclose(torch.tensor([min_loss]),
torch.tensor([loss]), delta): # 监测loss，当loss下降时保存模型，敏感度设置为
1e-4（默认为1e-5）
        print(f'Loss Reduce {min_loss} to {loss}')
        min_loss = loss
        save_file = f'rnn_checkpoint_best.pt'
        torch.save({
            'epoch': epoch+1,
            'model_state_dict': rnn_model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss
        }, save_file)
        print(f'Save checkpoint to {save_file}')
    print('-------------------------')
```

模型训练结果

```
loading from rnn_checkpoint_best.pt
Train epoch 11: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0007, accuracy: 9477/10000 (95%)
-------------------------
Train epoch 12: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0008, accuracy: 9425/10000 (94%)
-------------------------
Train epoch 13: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0006, accuracy: 9520/10000 (95%)
Loss Reduce 0.0007289471238851547 to 0.0006436171770095825
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 14: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0006, accuracy: 9575/10000 (96%)
Loss Reduce 0.0006436171770095825 to 0.0005800850979983806
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 15: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0007, accuracy: 9504/10000 (95%)
-------------------------
Train epoch 16: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0006, accuracy: 9533/10000 (95%)
-------------------------
Train epoch 17: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0006, accuracy: 9593/10000 (96%)
Loss Reduce 0.0005800850979983806 to 0.0005519135936163365
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 18: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0005, accuracy: 9660/10000 (97%)
Loss Reduce 0.0005519135936163365 to 0.0004750970598310232 3
Save checkpoint to rnn_checkpoint_best.pt
-------------------------
Train epoch 19: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0005, accuracy: 9629/10000 (96%)
-------------------------
Train epoch 20: 60000/60000, [-------------------------------------------------->] 100%
Test: average loss: 0.0005, accuracy: 9639/10000 (96%)
```

谢谢