

第十、十一周工作总结

张泽宇

2022 年 5 月 8 日

过去两周的主要进行的工作包括有：

- 修改原来的仿真模型，调整了故障持续时间，添加过渡电阻。
- 就改进后的模型调整了灰度图的生成方式，并将三相电流都作为输入生成灰度图，得到了较好的效果。
- 学习了 PyTorch 框架，基于 PyTorch 搭建神经网络模型，模型结构参考 LeNet-5 进行构建，对灰度图进行训练学习，得到较好的结果。
- 学习了 TensorBoard 的使用，可视化了神经网络结构和训练过程。

以下为详细叙述

1 仿真模型的修改

针对上周出现的问题，李教授给出了非常有用的建议。模型的改进主要体现在如下两点。

1.1 故障时间调整

首先回归最简单情况，不考虑故障发生时间的延迟和故障位置，只考虑故障发生的类型，简化问题。考虑到初始状态下电源电压为 0，有一个短暂的建立稳态的过程，因此假设故障在 0.04s 发生，持续到仿真结束，即永久性故障。从定性的角度来讲，这样更为合理的原因可能在于取多个故障周期，可以收集更多的故障特征，反应在灰度图上，即为不同故障类型的图像特征更为明显，可以避免上次多种情况下灰度图都只是一条黑纹的情况。

当永久性故障可以得到较好的目标效果后，再进一步考虑故障结束和故障发生时间延迟，最后考虑不同的故障发生地点。

1.2 过渡电阻设置

在故障发生点增加一个 1000Ω 的电阻。这样的目的是为了控制短路时故障电流不至于太大，同时将电路中的每个负载增大，使正常运行时存在一定的电流。这样可以避免上一周中故障电流与正常电流之间数量级差别太大。

这里选择 1000Ω 是基于试验的结果，当过渡电阻为这个值时，故障电流和故障电压之间的数量级控制在 10^1 之内。理论来讲，过渡电阻的阻值应当依据具体电路而定，没有固定值。

基于上面两种改进，得到了仿真模型的控制部分如下图所示：

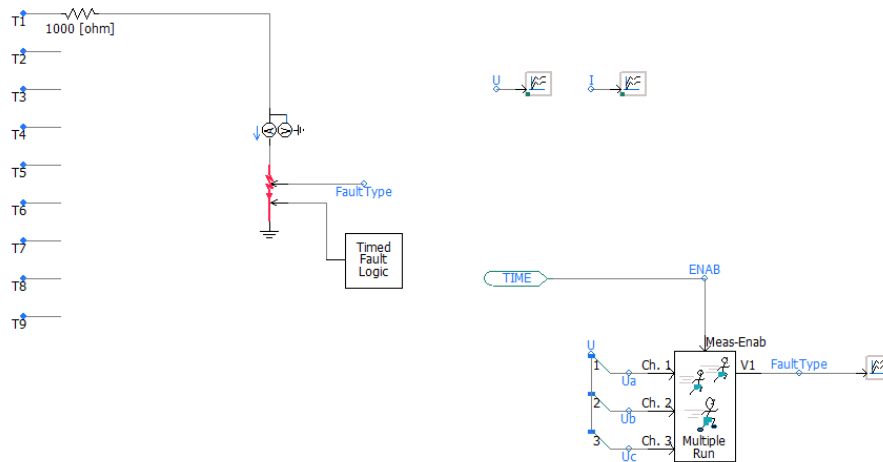


图 1: 改进后的模型控制部分

2 调整灰度图的生成方式

主要修改了两个部分：

2.1 将三相电流整合

听取李教授的建议，将三相电流全部作为输入生成灰度图。经过思考，我认为这样做是非常必要的。例如，如果只是单将一相电流作为输入，在三相对称的前提下，由于 A、B、C 单相短路接地时故障电流电压在时域上波形特征相同，只是发生时间推迟一些，在加上故障延迟的条件，很容易出现特征相似的情况。而如果将三相电流都作为输入，就可以避免这种情况，提高数据集的质量。

同时，三相电流必须以 $(i_A|_{t=t1}, i_B|_{t=t1}, i_C|_{t=t1}, i_A|_{t=t2}, i_B|_{t=t2}, i_C|_{t=t2} \dots)$ 的形式进行组织。因此添加了 Mix_data 函数，用 Flatten 的 F 模式合并读取到的数据。

```
1 def Mix_data(data):
2     dic = {'FaultType': data['FaultType']}
3     temp_U = np.array((data['U_1'], data['U_2'], data['U_3']))
4     dic['U'] = temp_U.flatten('F')
5     temp_I = np.array((data['I_1'], data['I_2'], data['I_3']))
6     dic['I'] = temp_I.flatten('F')
7     return dic
```

2.2 调整了灰度图像的大小

这部分改进是基于每次仿真时间为 0.2s，channel 输出步长为 $10\mu s$ ，因此可以得到 20000 个数据点。对此，采用 $128 \times 128 = 16384$ 的灰度图更为合适。而且由于 0.04s 开始发生故障，因此随机截取的灰度矩阵中可以包含故障发生的大部分数据和故障前的小部分数据。

基于上述两处改动，并对程序的一些细节部分进行优化，得到了最终的灰度图像，如图 3 至图 6 所示：

可以看出，相比于上周的结果，灰度图之间有较为明显的差异性。每种故障先生成 500 张灰度图，得到共 2000 张灰度图的数据集。

3 基于 PyTorch 的神经网络搭建

之所以转用 PyTorch 平台，是因为在之间的学习中，我发现 Matlab 构建神经网络太考验计算机性能，Matlab 对计算机内存要求有点高，这个和它的程序工作原理有一些关系，导致一些改动和想法都无法实现，同时每次神经网络的训练经历的时间比较长，所以用三周左右的时间学习了 PyTorch 框架，利用 Python 进行深度学习更容易实现，同时对模型的计算转到 GPU 上运行，计算速度得到了极大的提升。现将整个过程复述如下：

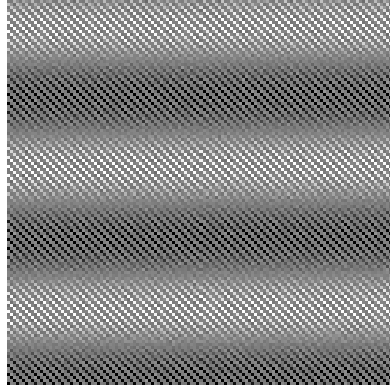


图 3: A 相短路接地

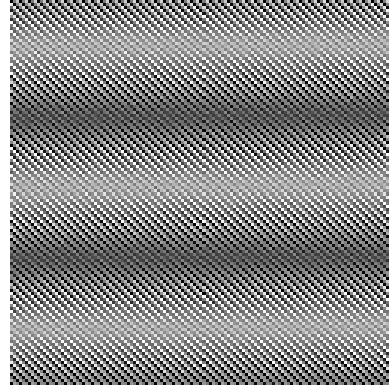


图 4: AB 相短路接地

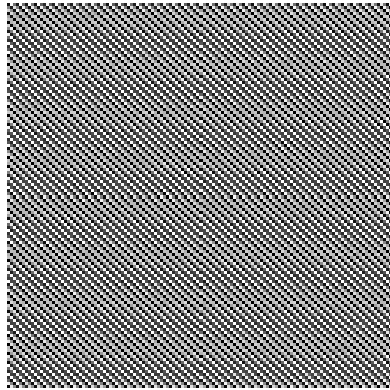


图 5: ABC 三相短路接地

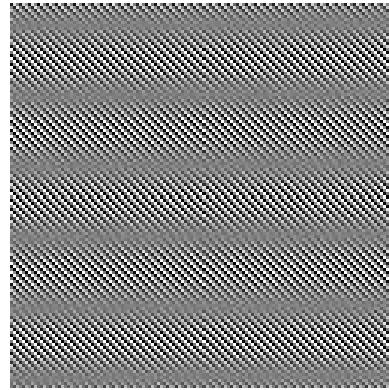


图 6: AB 相短路

3.1 神经网络搭建

利用 PyTorch 的现有函数，可以快速搭建神经网络。这部分程序如下：

```

1  import torch
    from torch import nn
3  from torch.utils.tensorboard import SummaryWriter

5

    class Net(nn.Module):
6
7        def __init__(self):
8            super(Net, self).__init__()
9            self.module = nn.Sequential(
10                nn.Conv2d(in_channels=1, out_channels=16, kernel_size=(5, 5)),
11                nn.ReLU(),
12                nn.MaxPool2d(kernel_size=(8, 8)),
13
14                nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(5, 5)),
15                nn.ReLU(),
16                nn.MaxPool2d(kernel_size=(2, 2)),
17

```

```

nn.Flatten(),
19
nn.Linear(in_features=32*5*5, out_features=256),
21 nn.Linear(in_features=256, out_features=36),
nn.Linear(in_features=36, out_features=4)
23 )

25 def forward(self, x):
    x = self.module(x)
27     return x

29
if __name__ == '__main__':
31     net = Net()
    input = torch.ones((32, 1, 128, 128))
33     output = net(input)
    print(output.shape)
35     writer = SummaryWriter('log_net')
    writer.add_graph(net, input)
37     writer.close()

```

需要指明，这次的神经网络是在现有的 LeNet-5 模型上搭建的，与以往自己的模型不同。LeNet-5 模型是一个 CNN 的经典结构¹，主要是用于手写字体的识别。它的输入为 32 像素图像，用来进行 10 分类。由于 LeNet-5 模型较为简易而且在简单图像识别上表现较优，所以借鉴其结构。但是由于输入、输出都不相同，所以在结构的基础上对参数进行修改。

在代码中，使用 TensorBoard 记录模型的信号流向，但是 Flow 图对网络结构表现得不是很清晰，所以又使用一种工具绘制了模型的结构，如下图 6 所示（由于 svg 图像无法在 latex 中引入，转成 png 图像时有一定的矢量损失，所以图像变模糊，后期探索有无解决方法）：

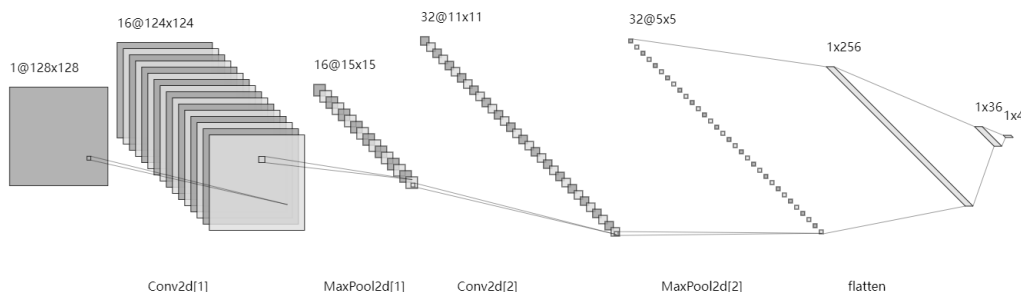


图 6: 模型结构

直观来讲，网络在第一层池化时 kernel_size 参数设置有些过大，可以进行优化。

¹<http://yann.lecun.com/exdb/lenet/index.html>

3.2 数据集准备

采用 PyTorch 提供的 Dataset 类制作数据集，格式为文件夹名为标签名，这里用 1、4、7、8 代表单相短路接地、两相短路接地、三相短路接地、两相短路。将总体数据随机选择 10% 作为验证集。这部分代码如下：

```
1     import torch
      from torch.utils.data import Dataset, random_split
3     import os
      from PIL import Image
5     from torchvision import transforms

7
      class ReadData(Dataset):
9         def __init__(self, root_dir, label_dir):
              self.root_dir = root_dir
11             self.label_dir = label_dir
              self.path = os.path.join(self.root_dir, self.label_dir)
13             self.img_path = os.listdir(self.path)
              self.transforms = transforms.ToTensor()

15
            def __getitem__(self, idx):
17                img_name = self.img_path[idx]
                img_item_path = os.path.join(self.root_dir, self.label_dir, img_name)
19                img = Image.open(img_item_path)
                img = self.transforms(img)
21                label = eval(self.label_dir)
                if label == 1:
23                    label = 0
                elif label == 4:
25                    label = 1
                elif label == 7:
27                    label = 2
                elif label == 8:
29                    label = 3
                else:
31                    label = 'wrong'
                label = torch.tensor(label)
33                return img, label

35
            def __len__(self):
37                return len(self.img_path)

39
            data_dir = ''
            type1_label_dir = '1'
41            type2_label_dir = '4'
            type3_label_dir = '7'
43            type4_label_dir = '8'

45
            type1_dataset = ReadData(data_dir, type1_label_dir)
            type2_dataset = ReadData(data_dir, type2_label_dir)
47            type3_dataset = ReadData(data_dir, type3_label_dir)
```

```

type4_dataset = ReadData(data_dir, type4_label_dir)
49
    len_train = 450
51    len_test = 50
    train_type1_dataset, test_type1_dataset = random_split(
53    dataset=type1_dataset,
    lengths=[len_train, len_test],
55    generator=torch.Generator().manual_seed(0)
    )
57    train_type2_dataset, test_type2_dataset = random_split(
    dataset=type2_dataset,
59    lengths=[len_train, len_test],
    generator=torch.Generator().manual_seed(0)
61    )
    train_type3_dataset, test_type3_dataset = random_split(
63    dataset=type3_dataset,
    lengths=[len_train, len_test],
65    generator=torch.Generator().manual_seed(0)
    )
67    train_type4_dataset, test_type4_dataset = random_split(
    dataset=type4_dataset,
69    lengths=[len_train, len_test],
    generator=torch.Generator().manual_seed(0)
71    )

73    train_dataset = train_type1_dataset + train_type2_dataset + train_type3_dataset + ...
    train_type4_dataset
    test_dataset = test_type1_dataset + test_type2_dataset + test_type3_dataset + ...
    test_type4_dataset
75

```

在这个程序中，打标签的过程（22 至 32 行）存在一些需要优化的地方。因为 Tensor 没有字符串的变量类型，针对这个问题可以把 1、4、7、8 由 str 类转为 int 类参与后面的运算，但是如果故障类型增加，由于损失函数计算是按照预测输出序列最大值的索引进行，那么第五类故障的索引 4 就会和标签 4 冲突，导致计算产生问题。所以这里选择把 1、4、7、8 重新打上 1、2、3、4 的标签，既不影响损失的计算，也可以避免发生冲突。但这并不是一个常用普遍的方法。

3.3 训练过程

训练过程分为：数据集导入、设置训练参数、设置训练、设置验证、数据记录五个模块。

3.3.1 数据集导入

```

train_data_set = Readdata_V2.train_dataset
2    test_data_set = Readdata_V2.test_dataset

4    train_data_size = len(train_data_set)

```

```

        test_data_size = len(test_data_set)

6
    print('=====')
8    print('训练集的长度为: {}'.format(train_data_size))
    print('测试集的长度为: {}'.format(test_data_size))
10   print('=====')

12   train_data_loader = DataLoader(train_data_set, batch_size=1, shuffle=True)
    test_data_loader = DataLoader(test_data_set, batch_size=1, shuffle=True)
14

```

这部分代码中有一个很关键的参数，`shuffle=True`，因为在数据集的制作过程中，（上个代码的 73、74 行）只是将不同故障类别的灰度图叠加，没有做打乱顺序处理，因此在数据排布上是按照四种故障依次出现的顺序。如果 `shuffle` 设置为 `False`，每批次取出图片前不打乱顺序，就会导致模型先对 1 故障训练，再对 4 故障训练，以此类推，最后的损失函数波动较大，虽然也能收敛，但是训练过程不很完善。

3.3.2 设置训练参数

```

1    learning_rate = 0.01
    epochs = 10
3    total_train_step = 0
    total_test_step = 0

5

    net = Net_V2.Net()
7    net.cuda()

9    loss_fc = nn.CrossEntropyLoss()
    loss_fc.cuda()
11   optimizer = torch.optim.SGD(params=net.parameters(), lr=learning_rate)
    writer = SummaryWriter('logs')
13

```

学习率取 0.01，训练 10 轮，损失函数采用交叉熵损失函数，优化方法选用随机梯度下降法，并利用 Tensorboard 记录训练过程。同时把网络和损失的计算放在 GPU 上进行，提升运算速率。

3.3.3 设置训练和验证

```

    for epoch in range(epochs):
2        print("-----第{}轮训练开始-----".format(epoch + 1))

4        for data in train_data_loader:
            imgs, labels = data

```



```

6         imgs = imgs.cuda()
        labels = labels.cuda()
8         outs = net(imgs)
        loss = loss_fc(outs, labels)
10
        optimizer.zero_grad()
12        loss.backward()
        optimizer.step()
14        total_train_step += 1
        writer.add_scalar('train_loss', loss, total_train_step)
16        print("训练次数:{}, loss:{}".format(total_train_step, loss.item()))

18    total_test_loss = 0
    total_accuracy = 0
20    with torch.no_grad():
        for data in test_data_loader:
22            imgs, labels = data
            imgs = imgs.cuda()
24            labels = labels.cuda()
            outs = net(imgs)
26            loss = loss_fc(outs, labels)
            total_test_loss += loss.item()
28            accuracy = (outs.argmax(1) == labels).sum()
            total_accuracy += accuracy
30            total_test_step += 1

32    print("第{}轮训练测试集loss:{}".format(epoch + 1, total_test_loss))
    print("第{}轮训练测试集正确率:{}".format(epoch + 1, total_accuracy / ...
test_data_size))
34    writer.add_scalar('test_loss', total_test_loss, total_test_step)
    writer.add_scalar('accuracy', total_accuracy / test_data_size, total_test_step)
36

```

每轮训练进行 450 次迭代，每结束一轮训练进行一次验证，分别计算训练损失、验证损失、正确率。

这部分代码中第 11 行和第 20 行是关键内容，可以保证优化的正确运算。

3.3.4 数据记录

分为两个部分：

- 保存模型，主要用于模型的验证。这里用 dict 即字典形式保存了模型的参数，再导入时需要 import 一下模型结构。

```

1        torch.save(net.state_dict(), 'Net_result.pth')
        print("模型已保存")
3        writer.close()

```

- 在 cmd 中打开 TensorBoard 查看训练结果。采用命令

```
1 TensorBoard --logdir=path
```

在主机端口打开 TensorBoard，查看训练过程。这一部分内容在下一节中展示。

3.4 训练结果

3.4.1 损失

在 TensorBoard 中查看训练损失、验证损失，如下图所示：

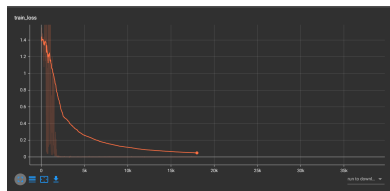


图 8: 训练损失

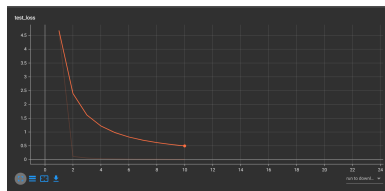


图 9: 验证损失

可以看出损失函数最终收敛，同时，在运行面板上导出的数据也表明训练的结果非常好。第 18000 次迭代的训练损失值为 $7.27174e-06$ ，第十轮的验证损失为 0.01166687。

3.4.2 分类测试

编写测试代码。

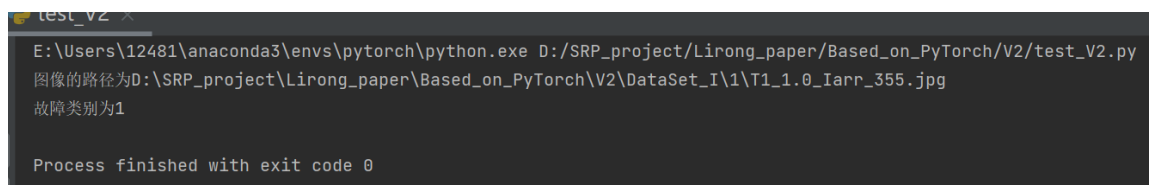
```
1 from PIL import Image
  from torchvision import transforms
3 import Net_V2
  import torch
5
  net_test = Net_V2.Net()
7 net_test.load_state_dict(torch.load('path'))
9
  img = Image.open('path')
  transform = transforms.ToTensor()
11 img = transform(img)
  img = torch.reshape(img, [-1, 1, 128, 128])
13 outs = net_test(img)
  out_index = outs.argmax(1)
15 if out_index == 0:
    print("故障类别为1")
17 elif out_index == 1:
    print("故障类别为4")
```

```

19     elif out_index == 2:
        print("故障类别为7")
21     elif out_index == 3:
        print("故障类别为8")
23     else:
        print("无法判断")
25

```

这里的关键代码在 1 行中的-1，即 Tensor 变量中的批次值让计算机自动选取。
修改 img 的图像路径，运行程序，发现取得非常好的分类效果。



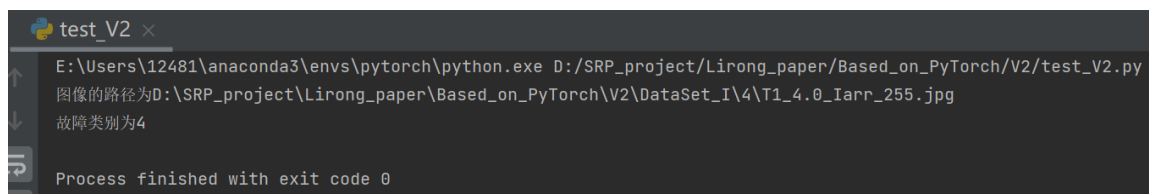
```

test_V2 x
E:\Users\12481\anaconda3\envs\pytorch\python.exe D:/SRP_project/Lirong_paper/Based_on_PyTorch/V2/test_V2.py
图像的路径为D:\SRP_project\Lirong_paper\Based_on_PyTorch\V2\DataSet_I\1\T1_1.0_Iarr_355.jpg
故障类别为1

Process finished with exit code 0

```

图 10: 故障 1 测试



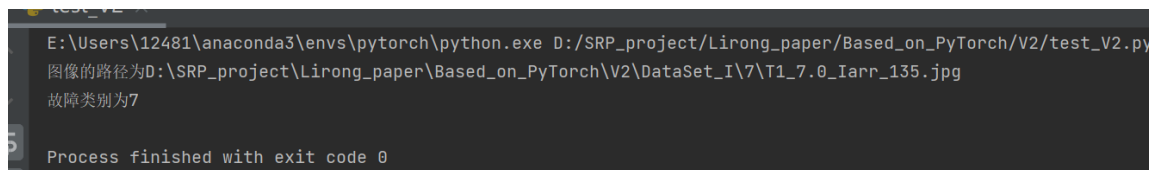
```

test_V2 x
E:\Users\12481\anaconda3\envs\pytorch\python.exe D:/SRP_project/Lirong_paper/Based_on_PyTorch/V2/test_V2.py
图像的路径为D:\SRP_project\Lirong_paper\Based_on_PyTorch\V2\DataSet_I\4\T1_4.0_Iarr_255.jpg
故障类别为4

Process finished with exit code 0

```

图 11: 故障 4 测试



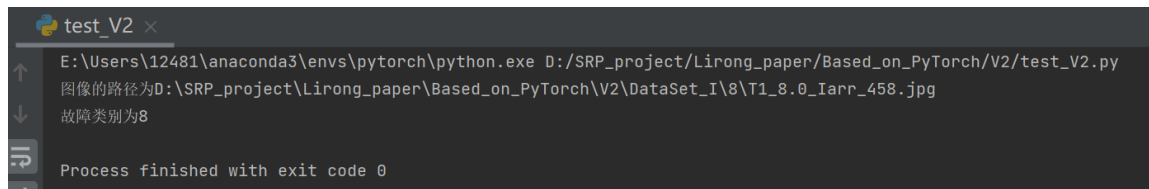
```

test_V2 x
E:\Users\12481\anaconda3\envs\pytorch\python.exe D:/SRP_project/Lirong_paper/Based_on_PyTorch/V2/test_V2.py
图像的路径为D:\SRP_project\Lirong_paper\Based_on_PyTorch\V2\DataSet_I\7\T1_7.0_Iarr_135.jpg
故障类别为7

Process finished with exit code 0

```

图 12: 故障 7 测试



```

test_V2 x
E:\Users\12481\anaconda3\envs\pytorch\python.exe D:/SRP_project/Lirong_paper/Based_on_PyTorch/V2/test_V2.py
图像的路径为D:\SRP_project\Lirong_paper\Based_on_PyTorch\V2\DataSet_I\8\T1_8.0_Iarr_458.jpg
故障类别为8

Process finished with exit code 0

```

图 13: 故障 8 测试

至此，达到了最初的目的，实现了对故障类型的分类处理。

整个流程看下来，存在一些需要改进的地方，比如模型结构上，第一层池化的池化核大小可以改小一些；数据集打标签的部分需要完善程序。下一周计划对这些做出改进，同时在这有的基础上实现对故障定位的功能。