数据读取

In [1]:

```
# 数据读取

import torch
import torchvision
import torchvision.transforms as transforms

# 设置对数据的变换
transform = transforms.Compose([
    transforms.Resize((32, 32)), # 将图片压缩至32x32
    transforms.ToTensor(),  # 将图片参数变成张量
    transforms.Normalize((0.1307, ), (0.3081, )) # 对张量进行标准化
])

# 下载训练集
train_set = torchvision.datasets.MNIST(root='../dataset', train=True, download=True, transform=transform)
# 下载验证集
test_set = torchvision.datasets.MNIST(root='../dataset', train=False, download=True, transform=transform)
```

生成数据生成器

In [2]:

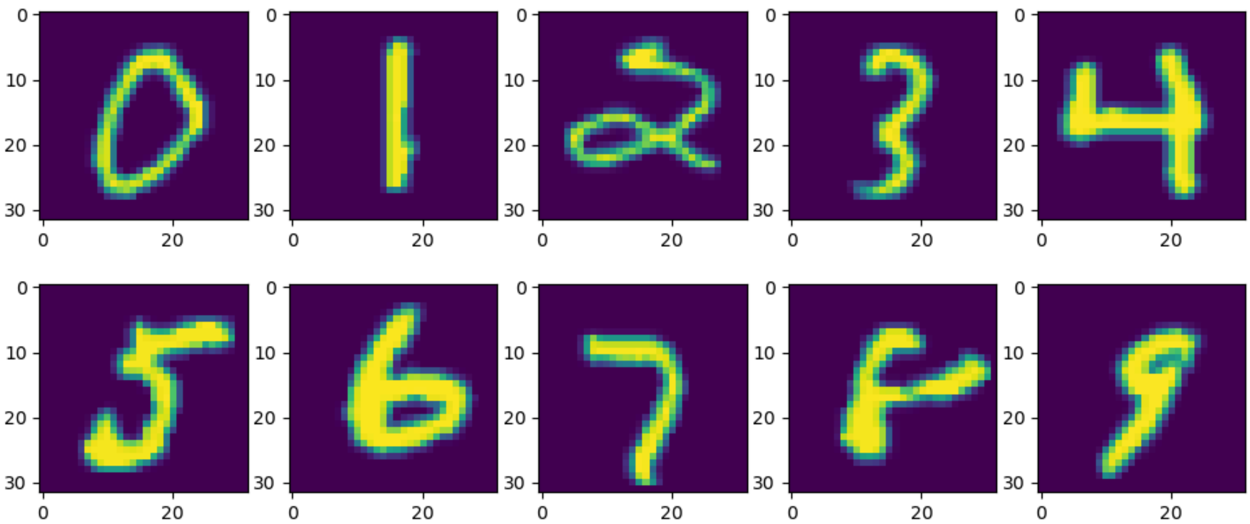
```
# 生成数据生成器

batch_size = 128  # batch size设置为128
train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_size, shuffle = True)  # 生成训练集数据生成器
test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_size, shuffle = False)  # 生成验证集数据生成器
```

数据可视化

In [18]:

```
# 数据可视化

from matplotlib import pyplot as plt
from torchvision.utils import make_grid
images, labels = next(iter(train_loader))  # 获取训练集第一个批次中的图片及相应标签
print(images.shape)
print(labels.shape)

fig,ax = plt.subplots(2, 5, figsize=(12, 5)) # 设置画布大小
ax = ax.flatten()  # 将ax压缩为一维数组, 方便存储图像数据
for i in range(10):
    im = images[labels==i][0].reshape(32, 32)  # 提取每个label为i的图像中的第一个照片, 原为(1, 32, 32)转换为(32, 32)
    ax[i].imshow(im)  # 插入这个画布第i个位置
plt.show()
```

```
torch.Size([128, 1, 32, 32])
torch.Size([128])
```



建立LeNet模型

In [4]:

```
# 建立模型

import torch.nn as nn
import torch.nn.functional as F

class LeNet(nn.Module):  # 建立LeNet类, 父类为nn.Module
    def __init__(self):
```

```
        super().__init__()  # 继承父类中的__init__属性
        # 建立卷积层
        self.conv = nn.Sequential(
            # 6个有1个通道的5x5卷积核，大小变为(-1, 6, 28, 28)
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1, padding=0),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 最大值池化，大小变为(-1, 6, 14, 14)
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5), # 16个有6个通道的5x5卷积核，大小变为(-1, 16, 10, 10)
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2) # 最大值池化，大小变为(-1, 16, 5, 5)
        )
        # 建立全连接层
        self.fc = nn.Sequential(
            nn.Linear(16*5*5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10),
        )

    def forward(self, x):
        feature = self.conv(x) # 先经过卷积层
        output = self.fc(feature.view(x.size()[0], -1)) # 将卷积层结果拉直，后经过全连接层
        return output


device = "cuda"  # 设置训练设备为GPU
model = LeNet()  # 建立LeNet的一个实例
model = model.to(device)  # 将model的参数放到GPU上
```

LeNet模型总结

In [5]:

```
# 模型总结

from torchsummary import summary
IMSIZE = 32
summary(model, (1, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 28, 28]             156
              ReLU-2            [-1, 6, 28, 28]               0
         MaxPool2d-3            [-1, 6, 14, 14]               0
            Conv2d-4           [-1, 16, 10, 10]           2,416
              ReLU-5           [-1, 16, 10, 10]               0
         MaxPool2d-6             [-1, 16, 5, 5]               0
            Linear-7                  [-1, 120]          48,120
              ReLU-8                  [-1, 120]               0
            Linear-9                   [-1, 84]          10,164
             ReLU-10                   [-1, 84]               0
           Linear-11                   [-1, 10]             850
================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.11
Params size (MB): 0.24
Estimated Total Size (MB): 0.35
----------------------------------------------------------------
```

LeNet模型参数量如何计算

第一个卷积层，共有六个卷积核，每个卷积核为5x5大小，且只有一个channel，故参数量为(5x5 + 1)x6 = 156 第二个卷积层，共有16个卷积核，每个仍为5x5大小，有6个channel，故参数量为(5x5x6 + 1)x16 = 2416 全连接层中，第一层由16x5x5降维至120，故对应矩阵参数量为(16x5x5 + 1)x120 = 48120 第二层由120降至84，参数量为(120+1)x84 = 10164 第三层由84降至10，参数量为(84+1)x10 = 850

LeNet模型训练

In [7]:

```
# 模型训练
import time

def accuracy(output, label):
    pred = torch.max(output, dim=1)[1]  # output的每一行中找出该行的最大值的索引
    return torch.sum(pred==label).item() / len(pred) # 计算pred与label匹配精度

# 对模型进行验证
def test(model, test_loader):
    # 初始化损失与精度变量
    test_loss = 0; test_acc = 0
    # 标记模型进行预测
    model.eval()
```

```
    for inputs, labels in test_loader:
        inputs, labels= inputs.to(device), labels.to(device) # 将输入和标签放至GPU
        optimizer.zero_grad() # 清楚优化器梯度
        outputs = model(inputs) # 计算输出
        loss = F.cross_entropy(outputs, labels) # 计算损失
        acc = accuracy(outputs, labels) # 计算精度
        test_loss += loss.item()
        test_acc += acc
    # 计算验证集上的损失和预测精度
    test_loss /= len(test_loader)
    test_acc /= len(test_loader)
    return test_loss, test_acc

# 打印训练日志
def printlog(epoch, train_time, train_loss, train_acc, test_loss, test_acc, epochs):
    print(f"Epoch [{epoch}/{epochs}], time: {train_time:.2f}s, train_loss: {train_loss:.4f}, train_accuracy = {train_acc:.4f}, t

# 对模型进行训练
def train(model, optimizer, train_loader, test_loader, epochs=1):
    # 初始化损失与精度列表
    train_losses = []; train_accs = []
    test_losses = []; test_accs = []
    # 标记模型进行训练
    model.train()
    # 重复epochs次
    for epoch in range(epochs):
        # 初始化损失与精度变量
        train_loss = 0; train_acc = 0
        test_loss = 0; test_acc = 0
        # 记录训练开始时间
        start = time.time()
        for inputs, labels in train_loader:
            # 将输入和标签放至GPU
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad() # 清空优化器梯度
            outputs = model(inputs) # 计算输出
            loss = F.cross_entropy(outputs, labels) # 计算损失
            acc = accuracy(outputs, labels) # 计算精度
            train_loss += loss.item()
            train_acc += acc
            loss.backward() # 反向传播
            optimizer.step() # 优化器优化

        end = time.time() # 记录一个epoch结束时间
        train_time = end - start # 计算一个epoch用时
        train_loss /= len(train_loader)
        train_acc /= len(train_loader)
        test_loss, test_acc = test(model, test_loader) # 将训练结果在验证集上检验一下
        # 将损失与精度变量计入进分别的列表中
        train_losses.append(train_loss); train_accs.append(train_acc)
        test_losses.append(test_loss); test_accs.append(test_acc)
        printlog(epoch+1, train_time, train_loss, train_acc, test_loss, test_acc, epochs) # 打印一个epoch的训练结果

    return train_losses, train_accs, test_losses, test_accs

epochs = 20 # epoch设置为20个
lr = 1e-3 # 学习率
optimizer = torch.optim.Adam(model.parameters(), lr=lr) # optimizer设置为Adam
train_losses, train_accs, test_losses, test_accs = train(model, optimizer, train_loader, test_loader, epochs=epochs) # 开启训练
```

```
Epoch [1/20], time: 30.85s, train_loss: 0.2699, train_accuracy = 0.9213, test_loss: 0.0925, test_accuracy: 0.9716
Epoch [2/20], time: 23.42s, train_loss: 0.0745, train_accuracy = 0.9769, test_loss: 0.0490, test_accuracy: 0.9834
Epoch [3/20], time: 22.19s, train_loss: 0.0519, train_accuracy = 0.9843, test_loss: 0.0416, test_accuracy: 0.9864
Epoch [4/20], time: 22.19s, train_loss: 0.0403, train_accuracy = 0.9879, test_loss: 0.0379, test_accuracy: 0.9867
Epoch [5/20], time: 22.48s, train_loss: 0.0330, train_accuracy = 0.9896, test_loss: 0.0317, test_accuracy: 0.9886
Epoch [6/20], time: 22.39s, train_loss: 0.0281, train_accuracy = 0.9913, test_loss: 0.0290, test_accuracy: 0.9895
Epoch [7/20], time: 22.33s, train_loss: 0.0226, train_accuracy = 0.9925, test_loss: 0.0283, test_accuracy: 0.9900
Epoch [8/20], time: 22.08s, train_loss: 0.0194, train_accuracy = 0.9937, test_loss: 0.0322, test_accuracy: 0.9905
Epoch [9/20], time: 22.27s, train_loss: 0.0167, train_accuracy = 0.9945, test_loss: 0.0387, test_accuracy: 0.9887
Epoch [10/20], time: 22.67s, train_loss: 0.0157, train_accuracy = 0.9952, test_loss: 0.0366, test_accuracy: 0.9905
Epoch [11/20], time: 23.19s, train_loss: 0.0138, train_accuracy = 0.9956, test_loss: 0.0358, test_accuracy: 0.9905
Epoch [12/20], time: 22.56s, train_loss: 0.0118, train_accuracy = 0.9959, test_loss: 0.0318, test_accuracy: 0.9910
Epoch [13/20], time: 21.41s, train_loss: 0.0118, train_accuracy = 0.9961, test_loss: 0.0313, test_accuracy: 0.9918
Epoch [14/20], time: 21.55s, train_loss: 0.0084, train_accuracy = 0.9974, test_loss: 0.0387, test_accuracy: 0.9896
Epoch [15/20], time: 22.55s, train_loss: 0.0113, train_accuracy = 0.9964, test_loss: 0.0387, test_accuracy: 0.9896
Epoch [16/20], time: 22.88s, train_loss: 0.0081, train_accuracy = 0.9974, test_loss: 0.0411, test_accuracy: 0.9895
Epoch [17/20], time: 22.50s, train_loss: 0.0071, train_accuracy = 0.9976, test_loss: 0.0523, test_accuracy: 0.9880
Epoch [18/20], time: 22.29s, train_loss: 0.0069, train_accuracy = 0.9977, test_loss: 0.0410, test_accuracy: 0.9906
Epoch [19/20], time: 23.04s, train_loss: 0.0067, train_accuracy = 0.9976, test_loss: 0.0482, test_accuracy: 0.9888
Epoch [20/20], time: 22.72s, train_loss: 0.0071, train_accuracy = 0.9977, test_loss: 0.0383, test_accuracy: 0.9905
```

建立自定义神经网络模型

In [9]:

```
# 自定义神经网络

class Customized_NN(nn.Module): # 建立自定义神经网络的类，其父类为nn.Module
    def __init__(self):
        super().__init__() # 继承父类的__init__属性
        # 卷积层
```

```
        self.conv = nn.Sequential(
            # 8个有1个通道的5x5卷积核，有1个padding，输出张量大小为(-1, 8, 30, 30)
            nn.Conv2d(in_channels=1, out_channels=8, kernel_size=5, stride=1, padding=1),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2, stride=2), # 平均值池化，输出张量大小为(-1, 8, 15, 15)
            # 20个有8个通道的5x5卷积核，有1个padding，输出张量大小为(-1, 20, 13, 13)
            nn.Conv2d(in_channels=8, out_channels=20, kernel_size=5, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2) # 最大值池化，输出张量大小为(-1, 20, 6, 6)
        )
        # 全连接层
        self.fc = nn.Sequential(
            nn.Linear(20*6*6, 200),
            nn.ReLU(),
            nn.Linear(200, 120),
            nn.ReLU(),
            nn.Linear(120, 56),
            nn.ReLU(),
            nn.Linear(56, 10)
        )

    def forward(self, x):
        feature = self.conv(x) # 先经过卷积层
        output = self.fc(feature.view(x.size()[0], -1)) # 将卷积层结果拉直，后经过全连接层
        return output

my_model = Customized_NN().cuda() # 建立此自定义神经网络的一个实例，并将参数放至GPU

IMSIZE = 32
summary(my_model, (1, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 8, 30, 30]             208
              ReLU-2            [-1, 8, 30, 30]               0
         AvgPool2d-3            [-1, 8, 15, 15]               0
            Conv2d-4           [-1, 20, 13, 13]           4,020
              ReLU-5           [-1, 20, 13, 13]               0
         MaxPool2d-6             [-1, 20, 6, 6]               0
            Linear-7                  [-1, 200]         144,200
              ReLU-8                  [-1, 200]               0
            Linear-9                  [-1, 120]          24,120
             ReLU-10                  [-1, 120]               0
           Linear-11                   [-1, 56]           6,776
             ReLU-12                   [-1, 56]               0
           Linear-13                   [-1, 10]             570
================================================================
Total params: 179,894
Trainable params: 179,894
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.19
Params size (MB): 0.69
Estimated Total Size (MB): 0.88
----------------------------------------------------------------
```

自定义神经网络模型参数量如何计算

第一个卷积层，共有8个卷积核，每个卷积核为5x5大小，且只有一个channel，故参数量为(5x5 + 1)x8 = 208 第二个卷积层，共有20个卷积核，每个仍为5x5大小，有8个channel，故参数量为(5x5x8 + 1)x20 = 4020 全连接层中，第一层由20x6x6降维至200，故对应矩阵参数量为(20x6x6 + 1)x200 = 144200 第二层由200降至120，参数量为(200+1)x120 = 24120 第三层由120降至56，参数量为(120+1)x56 = 6776 第四层由56降至10，参数量(56+1)x10 = 570

自定义神经网络模型训练

In [13]:
```
# 自定义模型训练
epochs = 10 # epochs设置为10
lr = 1e-4 # 学习率
optimizer = torch.optim.Adam(my_model.parameters(), lr=lr) # optimizer选定为Adam
train_losses, train_accs, test_losses, test_accs = train(my_model, optimizer, train_loader, test_loader, epochs=epochs) # 开始训
```

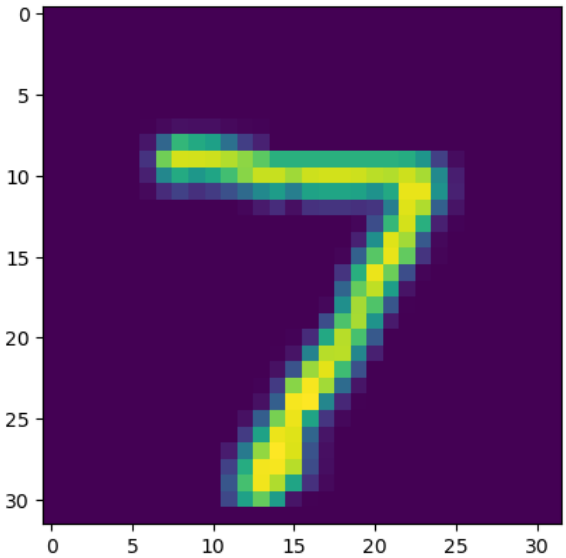
```
Epoch [1/10], time: 20.82s, train_loss: 0.0014, train_accuracy = 0.9995, test_loss: 0.0410, test_accuracy: 0.9915
Epoch [2/10], time: 24.40s, train_loss: 0.0004, train_accuracy = 0.9999, test_loss: 0.0416, test_accuracy: 0.9920
Epoch [3/10], time: 23.80s, train_loss: 0.0001, train_accuracy = 1.0000, test_loss: 0.0424, test_accuracy: 0.9922
Epoch [4/10], time: 21.96s, train_loss: 0.0001, train_accuracy = 1.0000, test_loss: 0.0435, test_accuracy: 0.9923
Epoch [5/10], time: 20.51s, train_loss: 0.0000, train_accuracy = 1.0000, test_loss: 0.0448, test_accuracy: 0.9923
Epoch [6/10], time: 20.40s, train_loss: 0.0000, train_accuracy = 1.0000, test_loss: 0.0458, test_accuracy: 0.9922
Epoch [7/10], time: 23.05s, train_loss: 0.0000, train_accuracy = 1.0000, test_loss: 0.0466, test_accuracy: 0.9924
Epoch [8/10], time: 22.59s, train_loss: 0.0000, train_accuracy = 1.0000, test_loss: 0.0476, test_accuracy: 0.9926
Epoch [9/10], time: 21.99s, train_loss: 0.0000, train_accuracy = 1.0000, test_loss: 0.0484, test_accuracy: 0.9927
Epoch [10/10], time: 22.05s, train_loss: 0.0000, train_accuracy = 1.0000, test_loss: 0.0492, test_accuracy: 0.9927
```

验证集预测精度最后为99.27%

LeNet模型预测

In [37]:
```
# 模型预测

images, labels = next(iter(test_loader)) # 从test_loader中获取一个batch的数据

im_1 = images[0].unsqueeze(0) # 提取这个batch中的第一张图片
im_copy1 = im_1
x_1 = labels[0].item() # 提取这个batch中的第一个标签
im_copy1 = im_copy1.reshape(32, 32)
plt.imshow(im_copy1) # 展示图片

# 预测im_1的数字
im_1 = im_1.to(device)
model.eval()
with torch.no_grad():
    pred_1 = model(im_1)

pred_1 = torch.max(pred_1, dim=1)[1].item() # 提取预测概率最高值的索引

print(f"LeNet模型对第一张照片数字的预测为：{pred_1}, 真实值为：{x_1}")
```

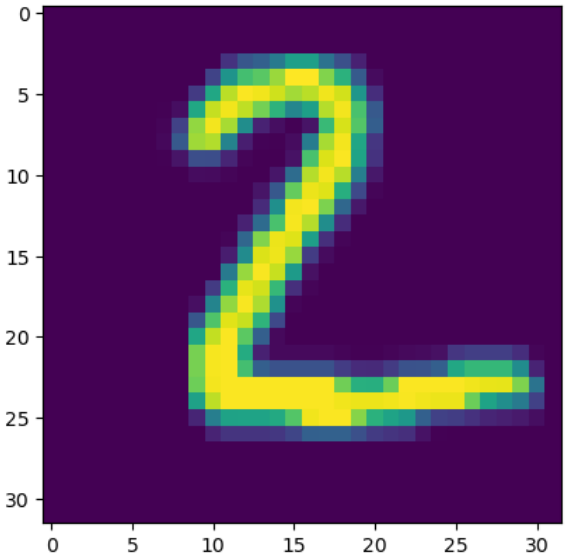
LeNet模型对第一张照片数字的预测为：7，真实值为：7



自定义神经网络模型预测

In [39]:
```
im_2 = images[1].unsqueeze(0) # 提取这个batch中的第二张图片
im_copy2 = im_2
x_2 = labels[1].item() # 提取这个batch中的第二个标签
im_copy2 = im_copy2.reshape(32, 32)
plt.imshow(im_copy2) # 展示图片

# 预测im_2的数字
im_2 = im_2.to(device)
my_model.eval()
with torch.no_grad():
    pred_2 = my_model(im_2)

pred_2 = torch.max(pred_2, dim=1)[1].item() # 提取预测概率最高值的索引

print(f"自定义神经网络模型对第二张照片数字的预测为：{pred_2}, 真实值为：{x_2}")
```

自定义神经网络模型对第二张照片数字的预测为：2，真实值为：2


0
5
10
15
20
25
30
0
5
10
15
20
25
30