# 一、安装与准备数据集

## 1.1 安装Pytorch 和 Torchvision

### 1.1.1 什么是Pytorch？

PyTorch是一个[开源](https://baike.baidu.com/item/开源/246339?fromModule=lemma_inlink)的[Python](https://baike.baidu.com/item/Python?fromModule=lemma_inlink)机器学习库，基于Torch，用于自然语言处理等应用程序。

### 1.1.2 什么是Torchvision？

torchvision 包含了目前流行的数据集，模型结构和常用的图片转换工具。

### 1.1.3 安装命令

pip install torch==1.11.0  
pip install torchvision==0.12.0

## 1.2 引入库

import torch  
import torchvision  
from torch.utils.data import DataLoader

# 二、准备数据集

## 2.1 定义预训练超参数

导入就绪后，我们可以继续准备将要使用的数据。但在那之前，我们将定义超参数，我们将使用的实验。在这里，epoch的数量定义了我们将循环整个训练数据集的次数，而learning\_rate和momentum是我们稍后将使用的优化器的超参数。对于可重复的实验，我们必须为任何使用随机数产生的东西设置随机种子——如numpy和random！

n\_epochs = 3 # epoch的数量定义了我们将循环整个训练数据集的次数  
  
# 我们将使用batch\_size=64进行训练，并使用size=1000对这个数据集进行测试。  
batch\_size\_train = 64 # 训练批次，即每个批次共有64条数据  
batch\_size\_test = 1000 # 测试批次，即每个批次共有1000条数据  
  
# learning\_rate和momentum是我们稍后将使用的优化器的超参数  
learning\_rate = 0.01 # 学习率  
momentum = 0.5 # 趋势  
  
log\_interval = 10 # 每10批次训练打印一次当前训练进度  
random\_seed = 1 # 随机数种子  
torch.manual\_seed(random\_seed) # 对于可重复的实验，我们必须为任何使用随机数产生的东西设置随机种子

## 2.2 加载训练数据集和测试数据集

现在我们还需要数据集的dataloader。这就是TorchVision发挥作用的地方。它让我们用一种方便的方式来加载MNIST数据集。我们将使用batch\_size=64进行训练，并使用size=1000对这个数据集进行测试。下面的Normalize()转换使用的值0.1307和0.3081是MNIST数据集的全局平均值和标准偏差，这里我们将它们作为给定值。TorchVision提供了许多方便的转换，比如裁剪或标准化。

### 2.2.1 加载数据集

运行下面的程序后，会自动将数据集下载到目录下的data文件夹。

# 加载训练数据集  
train\_loader = torch.utils.data.DataLoader(  
 torchvision.datasets.MNIST('./data/', train=True, download=True,  
 transform=torchvision.transforms.Compose([  
 torchvision.transforms.ToTensor(),  
 torchvision.transforms.Normalize(  
 (0.1307,), (0.3081,))  
 ])),  
 batch\_size=batch\_size\_train, shuffle=True)  
  
# 加载测试数据集  
test\_loader = torch.utils.data.DataLoader(  
 torchvision.datasets.MNIST('./data/', train=False, download=True,  
 transform=torchvision.transforms.Compose([  
 torchvision.transforms.ToTensor(),  
 torchvision.transforms.Normalize(  
 (0.1307,), (0.3081,))  
 ])),  
 batch\_size=batch\_size\_test, shuffle=True)

除了数据集和批处理大小之外，PyTorch的DataLoader还包含一些有趣的选项。例如，我们可以使用num\_workers > 1来使用子进程异步加载数据，或者使用固定RAM(通过pin\_memory)来加速RAM到GPU的传输。但是因为这些在我们使用GPU时很重要，我们可以在这里省略它们。

### 2.2.2 查看数据集结构

让我们看看一批测试数据由什么组成。

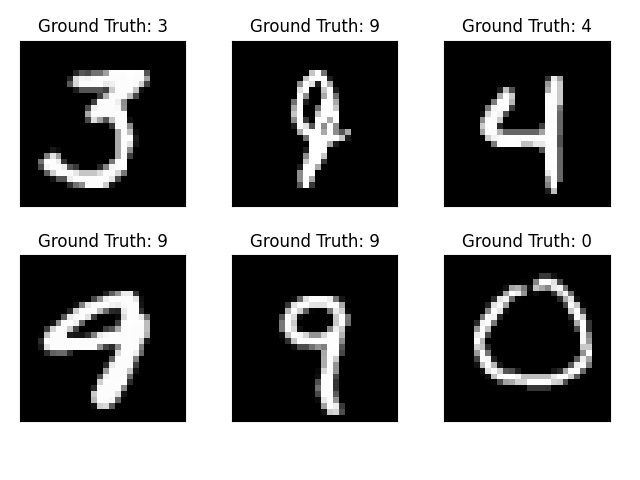
# enumerate 返回一个索引对应一个数据  
examples = enumerate(test\_loader) # 枚举测试数据集  
batch\_idx, (example\_data, example\_targets) = next(examples) # 拿取第一批的测试数据集  
# print(example\_targets) # 1000个图片的标签，标签范围0~9，对应图片中的数字。  
# print(example\_data.shape) # torch.Size([1000, 1, 28, 28])  
# example\_data.shape 1000条28x28像素的手写数字图片的灰度(即没有rgb通道)

### 2.2.3 绘制一部分图片数据

我们可以使用matplotlib来绘制其中的一些数字图片，并输出它的真实标记

# 我们可以使用matplotlib来绘制其中的一些图片  
fig = plt.figure() # 创建图像  
for i in range(6): # 取出头六条数据  
 plt.subplot(2, 3, i + 1) # 将图像分成 2x3 个子图对6条数据进行绘制，子图编号为1~N，所以i+1  
 plt.tight\_layout() # 调整子图之间和周围的填充。  
 # imshow 展示图片  
 plt.imshow(  
 example\_data[i][0], # 表示第i+1个28x28的像素数据  
 cmap='gray', # cmap='gray'指定该图为灰度图  
 interpolation='none' # 不插值  
 )  
 plt.title("Ground Truth: {}".format(example\_targets[i])) # 设置标题  
 plt.xticks([]) # 不显示x轴刻度  
 plt.yticks([]) # 不显示y轴刻度  
plt.show() # 展示所有子图

输出结果如下图所示：



# 二、创建模型

## 3.1 定义模型

现在让我们开始建立我们的网络。

1. 我们将使用两个2d卷积层，
2. 然后是两个全连接(或线性)层。
3. 作为激活函数，我们将选择整流线性单元(简称ReLUs)，
4. 作为正则化的手段，我们将使用两个dropout层。

在PyTorch中，构建网络的一个好方法是为我们希望构建的网络创建一个新类。让我们在这里导入一些子模块，以获得更具可读性的代码。

# 在PyTorch中，构建网络的一个好方法是为我们希望构建的网络创建一个新类。让我们在这里导入一些子模块，以获得更具可读性的代码。  
class Net(nn.Module):  
 def \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 # 我们将使用两个2d卷积层，然后是两个全连接(或线性)层。  
 self.conv1 = nn.Conv2d(1, 10, kernel\_size=5)  
 self.conv2 = nn.Conv2d(10, 20, kernel\_size=5)  
 self.conv2\_drop = nn.Dropout2d()  
 self.fc1 = nn.Linear(320, 50)  
 self.fc2 = nn.Linear(50, 10)  
  
 def forward(self, x):  
 # 作为激活函数，我们将选择整流线性单元(简称ReLUs)  
 x = F.relu(F.max\_pool2d(self.conv1(x), 2))  
 x = F.relu(F.max\_pool2d(self.conv2\_drop(self.conv2(x)), 2))  
 x = x.view(-1, 320)  
 x = F.relu(self.fc1(x))  
 # 作为正则化的手段，我们将使用两个dropout层。  
 x = F.dropout(x, training=self.training)  
 x = self.fc2(x)  
 return F.log\_softmax(x, dim=1)

广义地说，我们可以想到torch.nn层中包含可训练的参数，而torch.nn.functional就是纯粹的功能性。forward()传递定义了使用给定的层和函数计算输出的方式。为了便于调试，在前向传递中打印出张量是完全可以的。在试验更复杂的模型时，这就派上用场了。请注意，前向传递可以使用成员变量甚至数据本身来确定执行路径——它还可以使用多个参数！

现在让我们初始化网络和优化器。注意:如果我们使用GPU进行训练，我们也应该使用例如network.cuda()将网络参数发送给GPU。在将网络参数传递给优化器之前，将它们传输到适当的设备是很重要的，否则优化器将无法以正确的方式跟踪它们。

# 实例化网络  
network = Net()  
  
if torch.cuda.is\_available():  
 network.cuda() # 将网络转移到GPU上  
  
optimizer = optim.SGD(  
 network.parameters(), # 返回网络中的所有参数  
 lr=learning\_rate, # 指定学习率  
 momentum=momentum # 指定动量  
)

## 3.2 模型训练

是时候建立我们的训练循环了。

1. 我们要确保我们的网络处于训练模式。
2. 每个epoch对所有训练数据进行一次迭代。加载单独批次由DataLoader处理。
3. 我们需要使用optimizer.zero\_grad()手动将梯度设置为零，因为PyTorch在默认情况下会累积梯度。
4. 我们生成网络的输出(前向传递)，并计算输出与真值标签之间的负对数概率损失。
5. 现在，我们收集一组新的梯度，并使用optimizer.step()将其传播回每个网络参数。
6. 我们还将使用一些打印输出来跟踪进度。
7. 为了在以后创建一个良好的培训曲线，我们还创建了两个列表来节省培训和测试损失。
8. 在x轴上，我们希望显示网络在培训期间看到的培训示例的数量。

有关PyTorch自动渐变系统内部工作方式的详细信息，请参阅autograd的官方文档(强烈推荐)。

# 在x轴上，我们希望显示网络在培训期间看到的培训示例的数量。  
train\_losses = [] # 训练损失记录  
train\_counter = []  
  
# 我们还创建了两个列表来节省培训和测试损失。  
test\_losses = [] # 测试损失记录  
test\_counter = [i \* len(train\_loader.dataset) for i in range(n\_epochs + 1)]

在开始训练之前，我们将运行一次测试循环，看看仅使用随机初始化的网络参数可以获得多大的精度/损失。

def train(epoch):  
 # epoch 表示当前训练的轮数  
 network.train() # 将网络设置为训练模式  
 for batch\_idx, (data, target) in enumerate(train\_loader):  
 # batch\_idx 表示当前训练的批次  
 # data 表示当前批次的数据  
 # target 表示当前批次的标签  
  
 # 首先，我们需要使用optimizer.zero\_grad()手动将梯度设置为零，因为PyTorch在默认情况下会累积梯度。  
 optimizer.zero\_grad() # 清空梯度  
 output = network(data) # 将数据输入网络，得到输出  
  
 # 然后，我们生成网络的输出(前向传递)，并计算输出与真值标签之间的负对数概率损失。  
 loss = F.nll\_loss(output, target) # 计算损失  
 loss.backward() # 反向传播  
  
 # 现在，我们收集一组新的梯度，并使用optimizer.step()将其传播回每个网络参数。  
 optimizer.step() # 更新参数  
 if batch\_idx % log\_interval == 0:  
 print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(  
 epoch, # 当前训练的轮数  
 batch\_idx \* len(data), # 当前已训练的样本数  
 len(train\_loader.dataset), # 训练集总样本数  
 100. \* batch\_idx / len(train\_loader), # 当前训练进度  
 loss.item() # 当前损失  
 ))  
 train\_losses.append(loss.item()) # 每十次记录一次损失  
 train\_counter.append((batch\_idx \* 64) + ((epoch - 1) \* len(train\_loader.dataset))) # 每十次记录一次训练样本数  
 # .state\_dict() 返回一个包含模块整体状态的字典。  
 # 参数和持久性缓冲区（例如，运行平均数）都是包括在内。键是相应的参数和缓冲区名称。设置为 "无 "的参数和缓冲区不包括在内。  
 torch.save(network.state\_dict(), './model.pth') # 每十次保存一次模型  
 torch.save(optimizer.state\_dict(), './optimizer.pth') # 每十次保存一次优化器  
  
train(1)

现在进入测试循环。在这里，我们总结了测试损失，并跟踪正确分类的数字来计算网络的精度。

def test():  
 network.eval() # 将网络设置为评估模式  
 test\_loss = 0 # 测试损失  
 correct = 0 # 正确预测的样本数  
  
 # 使用上下文管理器no\_grad()，我们可以避免将生成网络输出的计算结果存储在计算图中。  
 # 禁用梯度计算的上下文管理器。在评估模型时，这是非常有用的，因为它可以减少内存使用，并加快计算。  
 # 禁用梯度计算对于推理很有用，当你确定你不会调用: meth:`Tensor.backward()`。它将减少内存计算的内存消耗  
 with torch.no\_grad():  
 for data, target in test\_loader: # 遍历测试数据集  
 output = network(data) # 将数据输入网络，得到输出  
 test\_loss += F.nll\_loss(output, target, reduction='sum').item() # sum 计算损失  
 pred = output.data.max(1, keepdim=True)[1]  
 correct += pred.eq(target.data.view\_as(pred)).sum() # 累加预测正确数  
 test\_loss /= len(test\_loader.dataset) # 计算平均损失  
 test\_losses.append(test\_loss) # 记录当前平均损失  
 print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(  
 test\_loss, # 打印当前平均损失  
 correct, # 打印预测正确数  
 len(test\_loader.dataset), # 打印测试集总样本数  
 100. \* correct / len(test\_loader.dataset) # 计算准确率  
 ))  
  
test() # 不加这个，后面画图就会报错：x and y must be the same size

使用上下文管理器no\_grad()，我们可以避免将生成网络输出的计算结果存储在计算图中。

Tips：小插曲，这里函数命名为**test**，pycharm会自动识别为**单元测试模式**，从而进入测试模式。

# 三、训练模型

## 4.1 开始训练

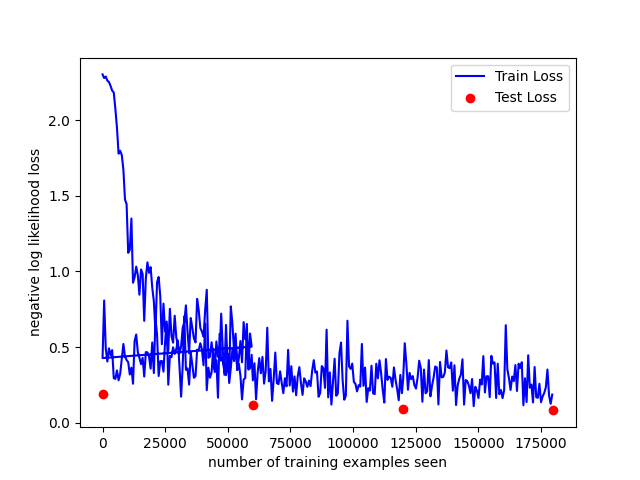
是时候开始训练了! 我们将在循环遍历n\_epochs之前手动添加test()调用，以使用随机初始化的参数来评估我们的模型。

for epoch in range(1, n\_epochs + 1):  
 train(epoch) # 训练 epoch 表示当前是第几轮训练  
 test() # 测试一下当前模型的效果

## 4.2 绘画训练曲线图

fig = plt.figure() # 创建图像  
plt.plot(train\_counter, train\_losses, color='blue') # 画出训练损失曲线  
plt.scatter(test\_counter, test\_losses, color='red') # 画出测试损失散点图  
plt.legend(['Train Loss', 'Test Loss'], loc='upper right') # 图例标识  
plt.xlabel('number of training examples seen') # x轴标签  
plt.ylabel('negative log likelihood loss') # y轴标签  
plt.show() # 显示图像

生成图像如下：



## 4.3 检验当前预测效果

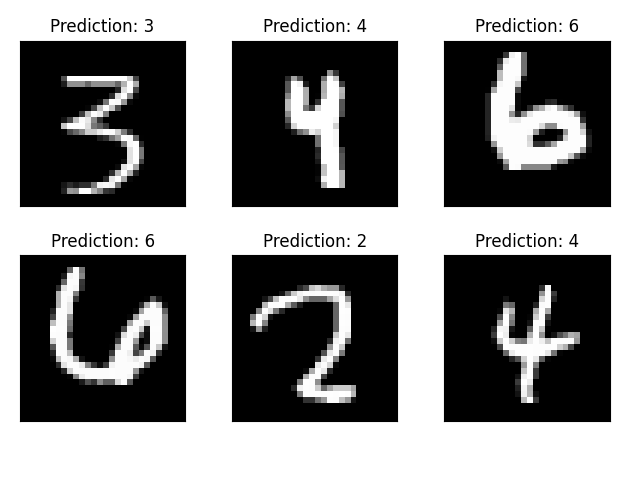
### 4.3.1 输出测试数据集中第一批数据的预测结果

examples = enumerate(test\_loader) # 枚举测试集  
batch\_idx, (example\_data, example\_targets) = next(examples) # 获取第一个批次的数据  
with torch.no\_grad(): # 不计算梯度，节省内存  
 output = network(example\_data) # 将数据输入网络，得到输出

### 4.3.2 输出预测结果

fig = plt.figure() # 创建图像  
for i in range(6): # 画出前6个样本的预测结果  
 plt.subplot(2, 3, i + 1) # 2行3列，第 i+1 个子图  
 plt.tight\_layout() # 自动适配子图参数，使之填充整个图像区域  
 plt.imshow(example\_data[i][0], cmap='gray', interpolation='none') # 画出第 i 个样本  
 plt.title("Prediction: {}".format(output.data.max(1, keepdim=True)[1][i].item())) # 标题为预测结果  
 plt.xticks([]) # 不显示 x 轴刻度  
 plt.yticks([]) # 不显示 y 轴刻度  
plt.show() # 显示图像

输出结果如图：



# 四、加载模型继续训练

这里展示如何从本地加载模型，然后继续训练。

## 5.1 创建网络

现在让我们继续对网络进行训练，或者看看如何从第一次培训运行时保存的state\_dicts中继续进行训练。

我们将初始化一组新的网络和优化器。

continued\_network = Net() # 创建网络  
continued\_optimizer = optim.SGD( # 创建优化器  
 network.parameters(), # 网络参数  
 lr=learning\_rate, # 学习率  
 momentum=momentum # 动量  
)

## 5.2 读取本地模型

使用.load\_state\_dict()，我们现在可以加载网络的内部状态，并在最后一次保存它们时优化它们。

# 加载模型参数  
network\_state\_dict = torch.load('model.pth')  
continued\_network.load\_state\_dict(network\_state\_dict)  
# 加载优化器参数  
optimizer\_state\_dict = torch.load('optimizer.pth')  
continued\_optimizer.load\_state\_dict(optimizer\_state\_dict)

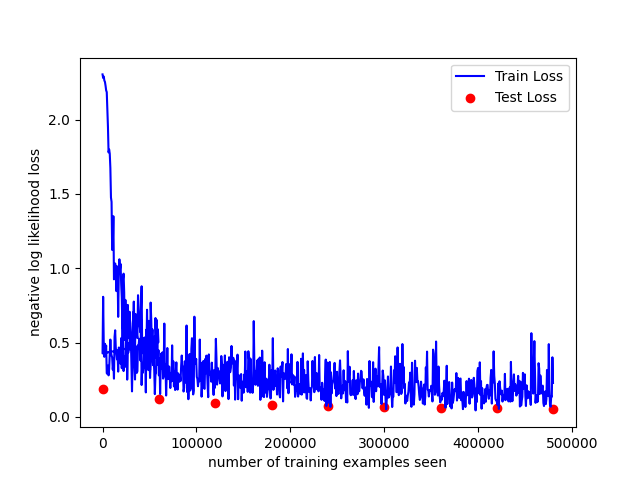
## 5.3 继续训练

同样，运行一个训练循环应该立即恢复我们之前的训练。为了检查这一点，我们只需使用与前面相同的列表来跟踪损失值。由于我们为所看到的训练示例的数量构建测试计数器的方式，我们必须在这里手动添加它。

# 注意不要注释前面的“for epoch in range(1, n\_epochs + 1):”部分，  
# 不然报错：x and y must be the same size  
# 为什么是“4”开始呢，因为n\_epochs=3，上面用了[1, n\_epochs + 1)  
for i in range(4, 9):  
 test\_counter.append(i \* len(train\_loader.dataset))  
 train(i) # 训练 epoch 表示当前是第几轮训练  
 test() # 测试一下当前模型的效果

## 5.4 绘画训练-测试曲线图

fig = plt.figure() # 创建图像  
plt.plot(train\_counter, train\_losses, color='blue') # 画出训练损失曲线  
plt.scatter(test\_counter, test\_losses, color='red') # 画出测试损失散点图  
plt.legend(['Train Loss', 'Test Loss'], loc='upper right') # 图例标识  
plt.xlabel('number of training examples seen') # x轴标签  
plt.ylabel('negative log likelihood loss') # y轴标签  
plt.show() # 显示图像



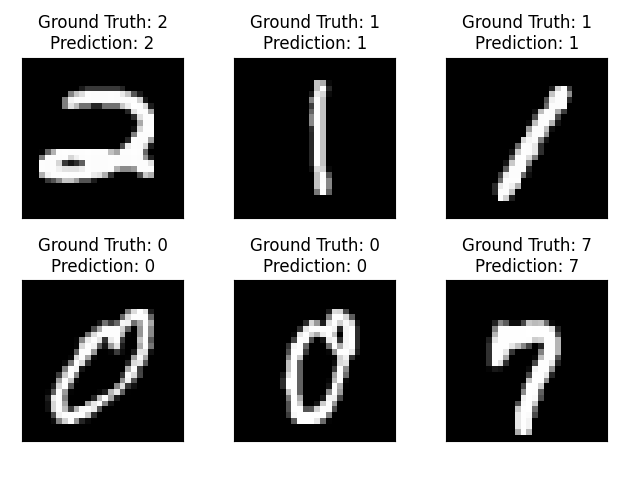
# 五、模型预测结果展示

定义一个函数，用于测试第一批测试数据集的头六个样本的预测效果。

同时展示预测标签与真实标签，便于对比预测正确与否。

def showDemoDigit():  
 examples = enumerate(test\_loader) # 枚举测试集  
 batch\_idx, (example\_data, example\_targets) = next(examples) # 获取第一个批次的数据  
 with torch.no\_grad(): # 不计算梯度，节省内存  
 output = network(example\_data) # 将数据输入网络，得到输出  
 fig = plt.figure() # 创建图像  
 for i in range(6): # 画出前6个样本的预测结果  
 plt.subplot(2, 3, i + 1) # 2行3列，第 i+1 个子图  
 plt.tight\_layout() # 自动适配子图参数，使之填充整个图像区域  
 plt.imshow(example\_data[i][0], cmap='gray', interpolation='none') # 画出第 i 个样本  
 plt.title(f"Ground Truth: {example\_targets[i]}\nPrediction: {output.data.max(1, keepdim=True)[1][i].item()}") # 标题为预测结果  
 plt.xticks([]) # 不显示 x 轴刻度  
 plt.yticks([]) # 不显示 y 轴刻度  
 plt.show() # 显示图像  
  
  
showDemoDigit()

展示结果如下：



# 六、总结与心得

通过学习使用Pytorch实现数字分类识别案例，我学会了：

1. 调用torchvision下载数据集并保存到本地。
2. 定义网络模型类，设计模型网络结构。
3. 使用matplotlib.pyplot子库进行绘图。
4. 读取本地模型，在上一次最新的训练进度下继续训练。

未来我将继续深入学习，实现数字的目标检测算法。