手写数字分类的简单神经网络 (PyTorch)

PyTorch 与 TensorFlow 一样，提供了深度神经网络的框架，它使得我们能够快速编写深度神经网络程序。PyTorch 的基本操作可参考官方网页或对应的中文翻译网页。

现在，我们使用 PyTorch 构建手写数字分类的简单神经网络。

定义神经网络

PyTorch 使用 `torch.nn` 包来构建神经网络，如下导入该模块

```
1 import torch
2 import torch.nn as nn
```

有了 `nn` 包后，可如下建立手写数字的简单神经网络

```
1  ## Define the network
2  class network(nn.Module):
3      def __init__(self, dim_input, dim_hidden, dim_output):
4          super(network, self).__init__()
5          self.fc1 = nn.Linear(dim_input, dim_hidden)
6          self.fc2 = nn.Linear(dim_hidden, dim_output)
7
8      def forward(self, x):
9          y = self.fc1(x)
10         y = torch.sigmoid(y)
11         y = self.fc2(y)
12         return y
```

- network 是建立的神经网络类名，可以是任何其他名称。
- 在自定义网络的时候，需要继承 `nn.Module` 类，并重新实现构造函数 `__init__` 和 `forward` 这两个方法。注意，`forward` 名称不能修改。 Line 4 是继承构造函数的标准写法。
- 在代码中，nn.Linear 表示的就是线性传递 $wx + b$, 而 torch.sigmoid 是激活函数。
    - forward 方法的意思就是：输入层线性传递到隐藏层，激活后再线性传递到输出层。
    - 为了程序的方便，我们将使用交叉熵函数作为损失函数的标准，因而输出层没有进行额外的激活。后面再具体说明。

  程序也可写为

```
## Define the network
class network(nn.Module):
    def __init__(self, dim_input, dim_hidden, dim_output):
        super(network, self).__init__()
        self.layer = nn.Sequential(
                        nn.Linear(dim_input, dim_hidden),
                        nn.Sigmoid(),
                        nn.Linear(dim_hidden, dim_output)
                        )

    def forward(self, x):
        y = self.layer(x)
        return y
```

- 通常采用前者形式，即把网络中具有可学习参数的层放在构造函数中，而不具有学习参数的层放在 forward 方法中。
- 可以发现，两种形式的 sigmoid 函数调用不同的包。在构造函数中，sigmoid 函数调用 nn 包中的 nn.Sigmoid, 而在 forward 方法中使用 torch.sigmoid。
- forward 中的函数一般使用 `torch.nn.functional`，即一般如下

  ```
  import torch.nn.functional as F
  y = F.sigmoid(y)
  ```

  但 sigmoid 函数现在改为 torch.sigmoid，而不使用 F.sigmoid 了。后者未来将会删除，具体原因未探究。

定义好网络后，就可如下创建具体的对象

```
## Create a Network object
model = network(dim_input, dim_hidden, dim_output)
```

似乎很多程序习惯用 model 作为对象的名称。

导入和加载数据

使用 PyTorch 最困惑的可能就是数据的处理。PyTorch 中定义了一种新的数据类型——张量，我们的数据必须转化为张量才能进行后续操作 (数据操作的难点实际上是用类的方式处理数据)。

PyTorch 中自带 MNIST 数据集，为了导入数据，先要加载图像相关的包

```
import torchvision
import torchvision.transforms as transforms
```

MNIST 数据按下述方式下载

```
# 下载图片和标签
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToTensor())
```

这里的 root 表示数据下载后存储的位置，`train=true` 表示该数据用作训练。

下载后，我们需要加载数据，并把它按 mini-batches 方式分为小批量数据

```
# 数据加载
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=mini_batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=mini_batch_size,
                                          shuffle=False)
```

- train_dataset 中有 60,000 个数据，若 mini_batch_size = 10，则数据分为 6,000 组。shuffle = True 表示数据进行打乱再分批。
- 每组数据的大小为 torch.Size([10, 1, 28, 28])，这里 10 表示图片的个数，1 表示灰度图，28 就是图像的尺寸。
- 在训练过程中，灰度图的信息要拉成 28*28 的向量，可如下操作

  ```
  images = images.reshape(-1, 28*28)
  # 或
  # images = images.view(images.size(0), -1)
  ```

  此时, images 的尺寸为 `torch.Size([10, 784])`，每行对应一张图片。注意，images.size(0) 是每组图像的个数，即 mini_batch_size (当然，最后一组的数据可能小于它)。

torch.utils.data.DataLoader

功能：产生可迭代的数据。
我们将专门介绍其用法，以封装个人数据。

训练神经网络

训练过程如下

```
## Train the network
# loss function
criterion = nn.CrossEntropyLoss()
# optimzer
optimizer = torch.optim.Adam(model.parameters(), lr=eta)
# loop of epochs
```

```
for ep in range(epochs):
    for images, labels in train_loader:

        # feedforward
        images = images.reshape(-1, 28*28)
        outputs = model(images)
        loss = criterion(outputs, labels)

        # backpropagation
        optimizer.zero_grad()
        loss.backward()  # compute the gradients

        ## train network with given optimizer
        optimizer.step() # update the parameters
```

- 上面的过程基本上是标准模式，其他问题稍加修改即可 (主要是数据的处理)。
- 在程序中，损失函数基于交叉熵标准，下一节说明。
- 优化器指的就是参数更新的方法。前面介绍的是随机梯度下降法，其实还可以进行额外的一些操作。一般认为 Adam 算法最好。

现在考察循环中的语句。

- 数据迭代
  - train_loader 是可迭代的数据，即小批量数据。
  - 对 mini_batch_size = 10，images 中有 10 张图片，labels 对应手写数字 (不是 10 维向量)。
- 前向传播
  - model 是神经网络对象，它接收 forward 方法的参数，见 Line 12.
  - 计算出输出后，就可以确定损失函数，即 Line 13.
  - 注意，此时损失函数是关于权重和偏置的函数。
- 反向传播
  - 确定好损失函数后，我们实施反向传播算法，计算出梯度等信息，即 Line 17。
  - PyTorch 使用计算图计算梯度，必须将梯度清零，才能用新的 min-batch 计算。否则梯度会累加。
  - 要弄清楚原因，可能必须了解计算图的过程。一个说明梯度确实累加的例子见知乎网页。
- 梯度更新
  - Line 20 对梯度进行了一次更新，使用的是 Adam 算法。

对每个 epoch 计算出的参数，可进行测试数据的预测。

```
    with torch.no_grad(): # 不构建计算图
        n_correct = 0
        n_test = len(test_dataset)
        for images, labels in test_loader:
            images = images.reshape(-1, 28*28)
            outputs = model(images)
            pred = torch.argmax(outputs.data, 1)
            n_correct += (pred == labels).sum().item()
        print("Epoch {:2d} : {} / {}".format(ep+1,n_correct,n_test))
```

- 这里，Line 1 表明不构建计算图，从而不会重新计算梯度。
- pred 是预测出的数字，其类型为 tensor，而 .item 用于获取 tensor 的元素。

损失函数的交叉熵标准

PyTorch 中预设了许多损失函数，它们用来衡量真实输出和神经网络输出的差距，如均方误差 `torch.nn.MSELoss()` 。

交叉熵的概念来自信息论，它的一个比较好的解释可参见网页 。交叉熵损失函数常用语分类问题中，其表达式如下

$$H(p,q) = -\sum_{i=1}^{n} p(x_i)\log(q(x_i)).$$

式中，

- n 表示所有可能的个数，即分类的个数。对手写数字分类，x_1 表示标签 0，x_2 表示标签 1，... , 即有如下的概率分布表

	x_1 (数字 0)	x_2 (数字 1)	.....	x_{10} (数字 9)
p				
q				

- $p(x)$ 是真实数据的概率分布，$q(x)$ 是预测数据的概率分布，而 log 表示自然对数。交叉熵度量这两种概率分布的差异。

下面详细说明 PyTorch 中的 `torch.nn.CrossEntropyLoss()` 。为了方便，这里假设分为三类，比如猫、狗和兔的分类。

Step 1: 将输出变为概率分布

- 对不同激活函数，每个输出神经元的值未必介于 0 到 1。随机生成输出

```
out = torch.randn(4,3)
# out = tensor([[ 1.6691,  0.7157, -1.2144],
#               [-0.1696,  0.0193,  1.0981],
#               [ 0.9182,  0.0048, -0.7070],
#               [-1.4975,  0.4232, -0.1497]])
```

这里有 4 个样本。

- 交叉熵标准首先使用柔性最大值 `nn.Softmax` 将每个输出神经元的值变为 0 到 1 的数，且和为 1。

```
softmax = nn.Softmax(dim=1) # 对行进行运算
sout = softmax(out)
# 结果为
# sout = tensor([[0.6938, 0.2674, 0.0388],
#                [0.1736, 0.2097, 0.6167],
#                [0.6258, 0.2510, 0.1232],
#                [0.0857, 0.5847, 0.3297]])
```

正因为如此，柔性最大值可视为概率分布。

Step 2: 取对数似然，即取负对数

```
lnout = -torch.log(sout)
# 结果为
# lnout = tensor([[0.3656, 1.3190, 3.2491],
#                 [1.7510, 1.5622, 0.4833],
#                 [0.4688, 1.3822, 2.0940],
#                 [2.4575, 0.5367, 1.1096]])
```

Step 3: 计算交叉熵

- 设猫、狗、兔的标签分别为 0, 1, 2 (注意从 0 开始编号)，若第一行对应的是狗，则相应的 3 维向量表示为 [0,1,0]。
- 这个 3 维向量就是样本的真实概率分布 $p = [0, 1, 0]$。
- 显然，$-\sum_{i=1}^{3} p_i \log(q_i)$ 就是取 $-\log(q)$ 对应真实标签位置的值，即取第 1 个位置的值 (从 0 开始)。
- 这样，交叉熵就是取出 lnout 对应标签位置的值再求和 (实际上 PyTorch 中还进行了平均)。假设标签是 [0,2,1,1]，即 4 张图分别为猫、兔、狗、狗。交叉熵的结果为

  ```
  (0.3656+0.4833+1.3822+0.5367)/4 = 0.6919500000000001
  ```

- 上面一步使用 `torch.nn.NLLLoss()` 实现，但它本身取了负号

  ```
  labels = torch.tensor([0,2,1,1])
  loss = torch.nn.NLLLoss()
  loss(lnout,labels)
  # 结果为 tensor(-0.6920)
  ```

损失函数的交叉熵标准

- 交叉熵损失函数 CrossEntropyLoss 就是把 Softmax–log–NLLLoss 合并成一步。
- 如果视 NLLLoss 为损失函数标准，那么 Softmax-log 可视为输出层的激活函数。实际上 `nn` 中有 `nn.LogSoftmax(dim=1)` 。
- NLLLoss 的第二个输入只需为标签 (从 0 开始)，因而 labels 不用转化为二进制向量。
- 图片必须与行对应。

程序整理

```
## Libraries
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

## Parameters
dim_input, dim_hidden, dim_output = 28*28, 15, 10
epochs = 10
mini_batch_size = 10
eta = 1e-3

## Load MNIST
```

```
# 下载图片和标签
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),

                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToTensor())
# 数据加载
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=mini_batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=mini_batch_size,
                                          shuffle=False)

## Define the network
class network(nn.Module):
    def __init__(self, dim_input, dim_hidden, dim_output):
        super(network, self).__init__()
        self.fc1 = nn.Linear(dim_input, dim_hidden)
        self.fc2 = nn.Linear(dim_hidden, dim_output)

    def forward(self, x):
        y = self.fc1(x)
        y = torch.sigmoid(y)
        y = self.fc2(y)
        return y

## Create a Network object
model = network(dim_input, dim_hidden, dim_output)


## Train the network
# loss function
criterion = nn.CrossEntropyLoss() #nn.MSELoss()
# optimzer
optimizer = torch.optim.Adam(model.parameters(), lr=eta)
# loop of epochs
for ep in range(epochs):
    for images, labels in train_loader:

        # feedforward
        images = images.reshape(-1, 28*28) # 有可能后面部分不是 10 个
        outputs = model(images)
        loss = criterion(outputs, labels)

        # backpropagation
        optimizer.zero_grad()
        loss.backward()  # compute the gradients

        ## train network with given optimizer
        optimizer.step() # update the parameters


```

```
    with torch.no_grad(): # 不构建计算图
        n_correct = 0
        n_test = len(test_dataset)
        for images, labels in test_loader:
            images = images.reshape(-1, 28*28)
            outputs = model(images)
            pred = torch.argmax(outputs.data, 1)
            n_correct += (pred == labels).sum().item()
        print("Epoch {:2d} : {} / {}".format(ep+1,n_correct,n_test))
```

计算结果如下

```
Epoch  1 : 9152 / 10000
Epoch  2 : 9265 / 10000
Epoch  3 : 9305 / 10000
Epoch  4 : 9348 / 10000
Epoch  5 : 9391 / 10000
Epoch  6 : 9416 / 10000
Epoch  7 : 9426 / 10000
Epoch  8 : 9418 / 10000
Epoch  9 : 9440 / 10000
Epoch 10 : 9436 / 10000
```

- 隐藏层的激活函数也可使用 relu 函数，即把 `torch.sigmoid(y)` 换成 `torch.relu(y)`，结果差不多。
- Python 的文件组织非常方便，函数和其他操作可在同一个文件中进行。

损失函数的均方差标准

下一个 Note 用均方差标准实现。