深层神经网络

下面我们直接用 mnist 举例，讲一讲深度神经网络

```
import numpy as np
import torch
from torchvision.datasets import mnist # 导入 pytorch 内置的 mnist 数据

from torch import nn
from torch.autograd import Variable
```

```
# 使用内置函数下载 mnist 数据集
train_set = mnist.MNIST('./data', train=True, download=True)
test_set = mnist.MNIST('./data', train=False, download=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
```

我们可以看看其中的一个数据是什么样子的

```
a_data, a_label = train_set[0]
```

```
a_data
```

```
a_label
```

```
5
```

这里的读入的数据是 PIL 库中的格式，我们可以非常方便地将其转换为 numpy array

```
a_data = np.array(a_data, dtype='float32')
print(a_data.shape)
```

```
(28, 28)
```

这里我们可以看到这种图片的大小是 28 x 28

```
print(a_data)
```

```
[[   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.    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.    3.   18.
    18.   18.  126.  136.  175.   26.  166.  255.  247.  127.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.    0.   30.   36.   94.  154.  170.  253.
   253.  253.  253.  253.  225.  172.  253.  242.  195.   64.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.   49.  238.  253.  253.  253.  253.  253.
   253.  253.  253.  251.   93.   82.   82.   56.   39.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.   18.  219.  253.  253.  253.  253.  253.
   198.  182.  247.  241.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.    0.   80.  156.  107.  253.  253.  205.
    11.    0.   43.  154.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.    0.    0.   14.    1.  154.  253.   90.
     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.  139.  253.  190.
     2.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   11.  190.  253.
    70.    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.   35.  241.
   225.  160.  108.    1.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [   0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.   81.
```

```
 240. 253. 253. 119.  25.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  45. 186. 253. 253. 150.  27.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.  16.  93. 252. 253. 187.   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. 249. 253. 249.  64.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  46. 130. 183. 253. 253. 207.   2.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  39. 148.
 229. 253. 253. 253. 250. 182.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  24. 114. 221. 253.
 253. 253. 253. 201.  78.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.  23.  66. 213. 253. 253. 253.
 253. 198.  81.   2.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.  18. 171. 219. 253. 253. 253. 253. 195.
  80.   9.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.  55. 172. 226. 253. 253. 253. 253. 244. 133.  11.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0. 136. 253. 253. 253. 212. 135. 132.  16.   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.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
```

我们可以将数组展示出来，里面的 0 就表示黑色，255 表示白色

对于神经网络，我们第一层的输入就是 28 x 28 = 784，所以必须将得到的数据我们做一个变换，使用 reshape 将他们拉平成一个一维向量

```
def data_tf(x):
    x = np.array(x, dtype='float32') / 255
    x = (x - 0.5) / 0.5 # 标准化，这个技巧之后会讲到
    x = x.reshape((-1,)) # 拉平
    x = torch.from_numpy(x)
    return x

train_set = mnist.MNIST('./data', train=True, transform=data_tf, download=True) # 重新载入数据集，申明定义的数据变换
test_set = mnist.MNIST('./data', train=False, transform=data_tf, download=True)
```

```
a, a_label = train_set[0]
print(a.shape)
print(a_label)
```

```
torch.Size([784])
5
```

```
from torch.utils.data import DataLoader
# 使用 pytorch 自带的 DataLoader 定义一个数据迭代器
train_data = DataLoader(train_set, batch_size=64, shuffle=True)
test_data = DataLoader(test_set, batch_size=128, shuffle=False)
```

使用这样的数据迭代器是非常有必要的，如果数据量太大，就无法一次将他们全部读入内存，所以需要使用 python 迭代器，每次生成一个批次的数据

```
a, a_label = next(iter(train_data))
```

```
# 打印出一个批次的数据大小
print(a.shape)
print(a_label.shape)
```

```
torch.Size([64, 784])
torch.Size([64])
```

```
# 使用 Sequential 定义 4 层神经网络
net = nn.Sequential(
    nn.Linear(784, 400),
    nn.ReLU(),
    nn.Linear(400, 200),
    nn.ReLU(),
    nn.Linear(200, 100),
    nn.ReLU(),
    nn.Linear(100, 10)
)
```

```
net
```

```
Sequential(
  (0): Linear(in_features=784, out_features=400)
  (1): ReLU()
  (2): Linear(in_features=400, out_features=200)
  (3): ReLU()
  (4): Linear(in_features=200, out_features=100)
  (5): ReLU()
  (6): Linear(in_features=100, out_features=10)
)
```

交叉熵在 pytorch 中已经内置了，交叉熵的数值稳定性更差，所以内置的函数已经帮我们解决了这个问题

```
# 定义 loss 函数
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), 1e-1) # 使用随机梯度下降，学习率 0.1
```

```
# 开始训练
losses = []
acces = []
eval_losses = []
eval_acces = []

for e in range(20):
    train_loss = 0
    train_acc = 0
    net.train()
    for im, label in train_data:
        im = Variable(im)
        label = Variable(label)
        # 前向传播
        out = net(im)
        loss = criterion(out, label)
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # 记录误差
        train_loss += loss.data[0]
        # 计算分类的准确率
        _, pred = out.max(1)
        num_correct = (pred == label).sum().data[0]
        acc = num_correct / im.shape[0]
```

```
        train_acc += acc

    losses.append(train_loss / len(train_data))
    acces.append(train_acc / len(train_data))
    # 在测试集上检验效果
    eval_loss = 0
    eval_acc = 0
    net.eval() # 将模型改为预测模式
    for im, label in test_data:
        im = Variable(im)
        label = Variable(label)
        out = net(im)
        loss = criterion(out, label)
        # 记录误差
        eval_loss += loss.data[0]
        # 记录准确率
        _, pred = out.max(1)
        num_correct = (pred == label).sum().data[0]
        acc = num_correct / im.shape[0]
        eval_acc += acc

    eval_losses.append(eval_loss / len(test_data))
    eval_acces.append(eval_acc / len(test_data))
    print('epoch: {}, Train Loss: {:.6f}, Train Acc: {:.6f}, Eval Loss: {:.6f}, Eval
Acc: {:.6f}'
          .format(e, train_loss / len(train_data), train_acc / len(train_data),
                     eval_loss / len(test_data), eval_acc / len(test_data)))
```

```
epoch: 0, Train Loss: 0.525527, Train Acc: 0.830690, Eval Loss: 0.214004, Eval Acc:
0.929292
epoch: 1, Train Loss: 0.169223, Train Acc: 0.948527, Eval Loss: 0.156571, Eval Acc:
0.951048
epoch: 2, Train Loss: 0.119509, Train Acc: 0.962537, Eval Loss: 0.141246, Eval Acc:
0.955301
epoch: 3, Train Loss: 0.093633, Train Acc: 0.970349, Eval Loss: 0.096926, Eval Acc:
0.970036
epoch: 4, Train Loss: 0.077827, Train Acc: 0.975413, Eval Loss: 0.088236, Eval Acc:
0.971025
epoch: 5, Train Loss: 0.062835, Train Acc: 0.980211, Eval Loss: 0.090155, Eval Acc:
0.973200
epoch: 6, Train Loss: 0.053678, Train Acc: 0.983109, Eval Loss: 0.084136, Eval Acc:
0.974189
epoch: 7, Train Loss: 0.056607, Train Acc: 0.982343, Eval Loss: 0.075727, Eval Acc:
0.976562
```

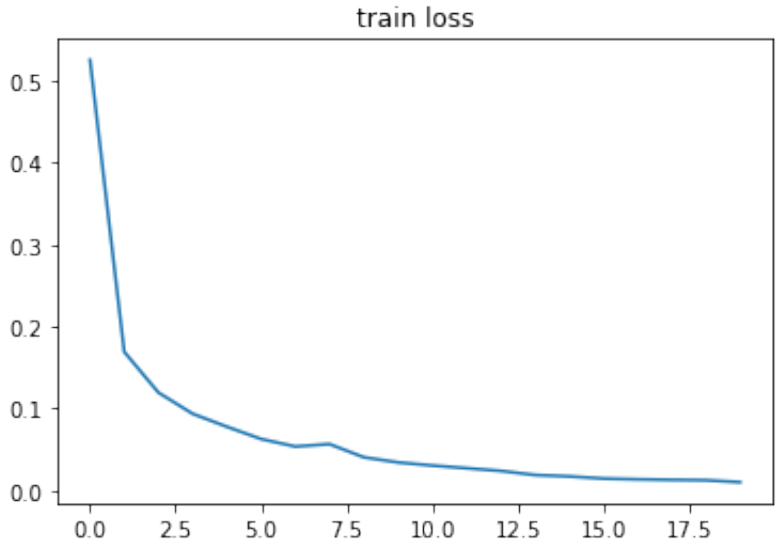
```
epoch: 8, Train Loss: 0.040552, Train Acc: 0.986774, Eval Loss: 0.065600, Eval Acc:
0.980024
epoch: 9, Train Loss: 0.034272, Train Acc: 0.989272, Eval Loss: 0.121962, Eval Acc:
0.963212
epoch: 10, Train Loss: 0.030490, Train Acc: 0.990005, Eval Loss: 0.067141, Eval Acc:
0.979233
epoch: 11, Train Loss: 0.027200, Train Acc: 0.991188, Eval Loss: 0.160441, Eval Acc:
0.953521
epoch: 12, Train Loss: 0.023948, Train Acc: 0.991904, Eval Loss: 0.076049, Eval Acc:
0.980123
epoch: 13, Train Loss: 0.018909, Train Acc: 0.993503, Eval Loss: 0.065272, Eval Acc:
0.980518
epoch: 14, Train Loss: 0.017229, Train Acc: 0.994386, Eval Loss: 0.067790, Eval Acc:
0.981309
epoch: 15, Train Loss: 0.014564, Train Acc: 0.995253, Eval Loss: 0.067104, Eval Acc:
0.981804
epoch: 16, Train Loss: 0.013621, Train Acc: 0.995819, Eval Loss: 0.076764, Eval Acc:
0.980716
epoch: 17, Train Loss: 0.012969, Train Acc: 0.995836, Eval Loss: 0.154731, Eval Acc:
0.963805
epoch: 18, Train Loss: 0.012531, Train Acc: 0.996202, Eval Loss: 0.098053, Eval Acc:
0.975574
epoch: 19, Train Loss: 0.010139, Train Acc: 0.996635, Eval Loss: 0.072089, Eval Acc:
0.982002
```

画出 loss 曲线和 准确率曲线

```
import matplotlib.pyplot as plt
%matplotlib inline
```

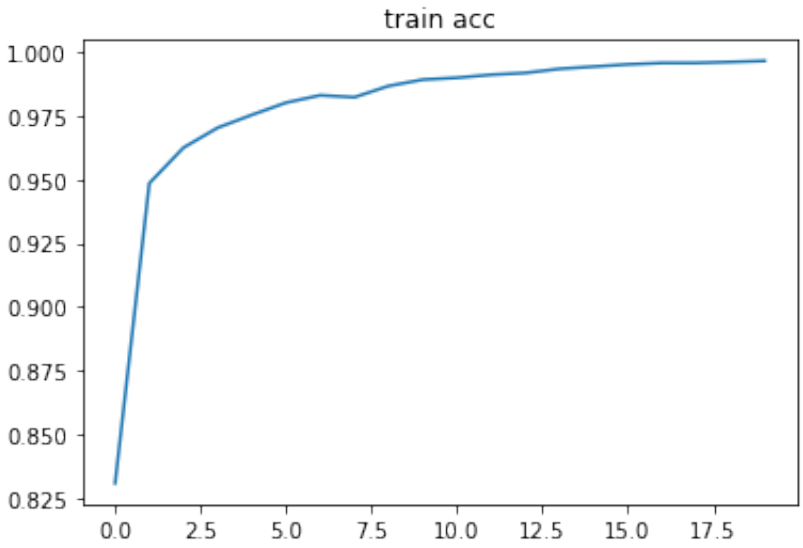
```
plt.title('train loss')
plt.plot(np.arange(len(losses)), losses)
```

```
[<matplotlib.lines.Line2D at 0x1132fb390>]
```



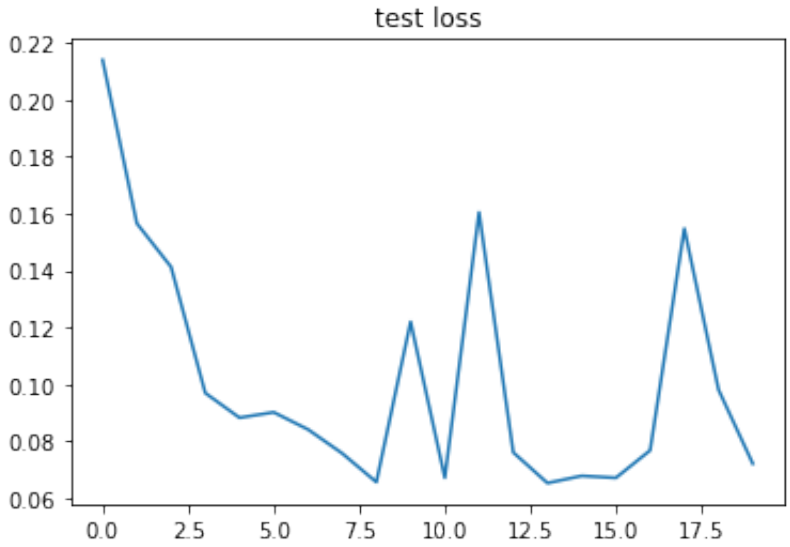

```
plt.plot(np.arange(len(acces)), acces)
plt.title('train acc')
```

```
<matplotlib.text.Text at 0x1134fad68>
```



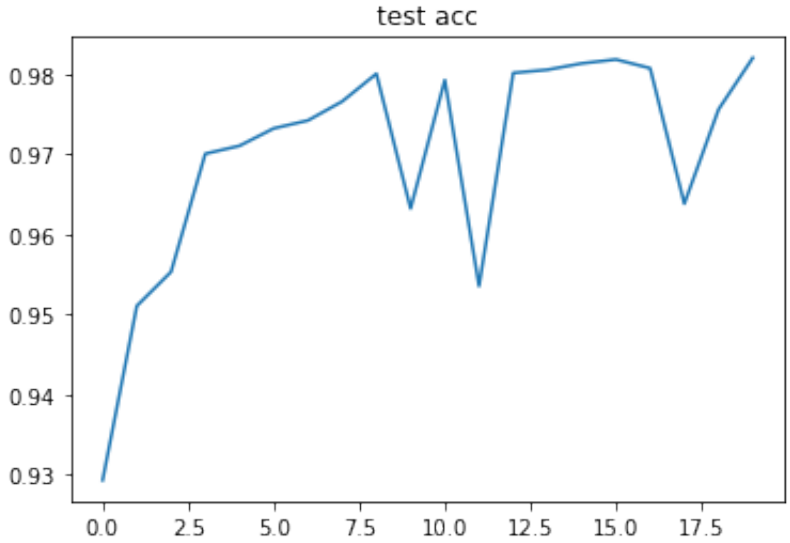

```
plt.plot(np.arange(len(eval_losses)), eval_losses)
plt.title('test loss')
```

```
<matplotlib.text.Text at 0x1136d2860>
```




```
plt.plot(np.arange(len(eval_acces)), eval_acces)
plt.title('test acc')
```

```
<matplotlib.text.Text at 0x1137a9828>
```



可以看到我们的三层网络在训练集上能够达到 99.9% 的准确率，测试集上能够达到 98.20% 的准确率

小练习：看一看上面的训练过程，看一下准确率是怎么计算出来的，特别注意 max 这个函数

自己重新实现一个新的网络，试试改变隐藏层的数目和激活函数，看看有什么新的结果