机器学习 hw2

史睿 518030910397

理论题 1.

现假设样本来自三个类，某次训练中的一个 batch 包含 3 个训练样本 x1, x2, x3, 分别来自第 1, 2, 3 类。

a) 试推导采用单热向量编码时该 batch 交叉熵损失函数表达式。

b) 如果网络输出为 y1 =(0.65, 0.43, 0.11), y2=(0.05, 0.51, 0.18), y2 =(0.33,0.21, 0. 72)，计算交叉熵损失函数值。

Sol. (a) 训练样本 $\mathcal{X}=\{x_1, x_2, x_3\}$

其中 $x_1=[1\ 0\ 0]^T$, $x_2=[0\ 1\ 1]^T$, $x_3=[0\ 1\ 1]^T$

网络输出 $y=\{y_1, y_2, y_3\}$

由 Softmax 函数，$P(j|y_i)=\dfrac{e^{y_{ij}}}{\sum_{k=1}^{3} e^{y_{ik}}}$ ， $j=1,2,3$

故网络输出 $y_{ij}=\dfrac{e^{y_{ij}}}{\sum_{k=1}^{3} e^{y_{ik}}}$

Cross Entropy Loss

$$J(x,y)=\sum_{i=1}^{3} H(x_i, p(y_i))=\sum_{i=1}^{3}\left(-\sum_{j=1}^{3} x_{ij}\log p(y_{ij})\right)$$

$$=-\log p(y_{11})-\log p(y_{22})-\log p(y_{33})$$

$$=-\log\frac{e^{y_{11}}}{\sum_{k=1}^{3} e^{y_{1k}}}-\log\frac{e^{y_{22}}}{\sum_{k=1}^{3} e^{y_{2k}}}-\log\frac{e^{y_{33}}}{\sum_{k=1}^{3} e^{y_{3k}}}$$

(b) $$J=-\log\frac{e^{0.65}}{e^{0.65}+e^{0.43}+e^{0.11}}-\log\frac{e^{0.51}}{e^{0.05}+e^{0.51}+e^{0.18}}-\log\frac{e^{0.72}}{e^{0.33}+e^{0.21}+e^{0.72}}$$

$=2.55$

理论题 2.

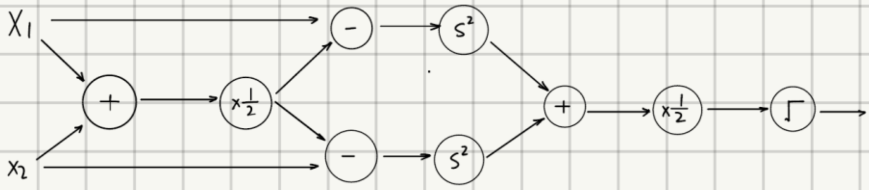
假设输入有 2 个样本 x1, x2

(1)请画计算 x1, x2 标准差的详细计算图;

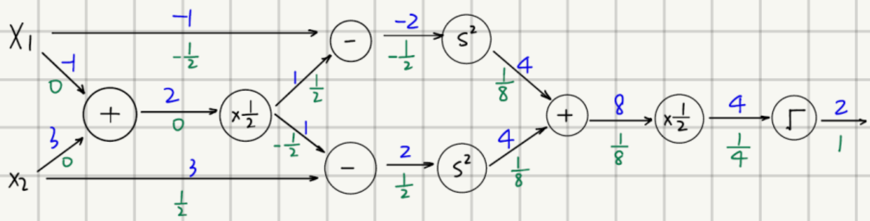
(2) 标出当 x1 =-1, x2 =3 时输出对图中每个节点输入变量的梯度值，并求出 x1, x2 总的梯度值。

Sol. (1) 标准差公式 $\sigma = \sqrt{\dfrac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$

故 x_1, x_2 的标准差为 $\sigma = \sqrt{\dfrac{\left(x_1 - \frac{x_1+x_2}{2}\right)^2 + \left(x_2 - \frac{x_1+x_2}{2}\right)^2}{2}}$



(2)



Green 为反向传播梯度值

x_1 总梯度为 $-\frac{1}{2}$

x_2 总梯度为 $\frac{1}{2}$

实践题一

1）实现一个三层神经网络，并使用 iris 数据集前 80%训练、后 20%测试，要求测试错误率小于 5%，分析至少三种非线性激活函数的影响。

源代码 iris_pytorch.ipynb

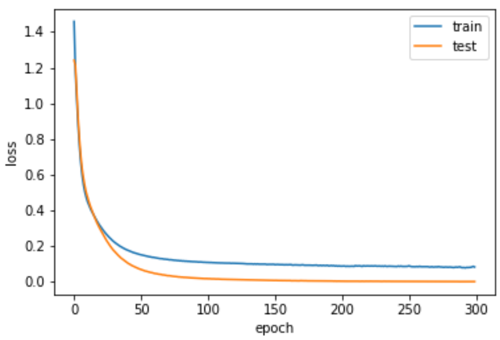
```
In  [21]: model

Out[21]: ThreeLayerNet(
            (fc1): Linear(in_features=4, out_features=40, bias=True)
            (fc2): Linear(in_features=40, out_features=3, bias=True)
            (relu): ReLU()
            (sigmoid): Sigmoid()
            (tanh): Tanh()
            (prelu): PReLU(num_parameters=1)
          )
```

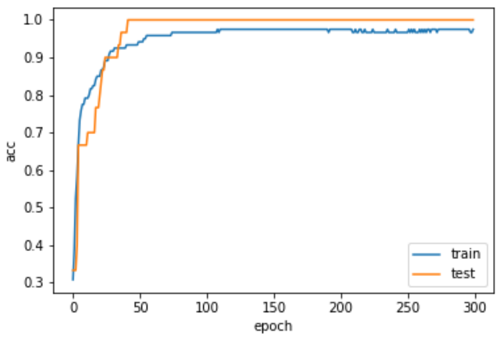
实验采用了简单的三层网络结构，全连接隐藏层有 40 个神经元，激活函数对比了 ReLU, Sigmoid, Tanh, PReLU 四种，损失函数为 CrossEntropyLoss

①Relu 激活函数

Relu

```
In  [32]: main(activation='relu')

relu
Eopch: 49  Loss: 0.15428  ACC: 0.94167
Test  Loss: 0.07261  ACC: 1.00000
Eopch: 99  Loss: 0.10944  ACC: 0.96667
Test  Loss: 0.01939  ACC: 1.00000
Eopch: 149  Loss: 0.09951  ACC: 0.97500
Test  Loss: 0.00899  ACC: 1.00000
Eopch: 199  Loss: 0.08869  ACC: 0.97500
Test  Loss: 0.00483  ACC: 1.00000
Eopch: 249  Loss: 0.08793  ACC: 0.96667
Test  Loss: 0.00390  ACC: 1.00000
Eopch: 299  Loss: 0.08414  ACC: 0.97500
Test  Loss: 0.00267  ACC: 1.00000
```



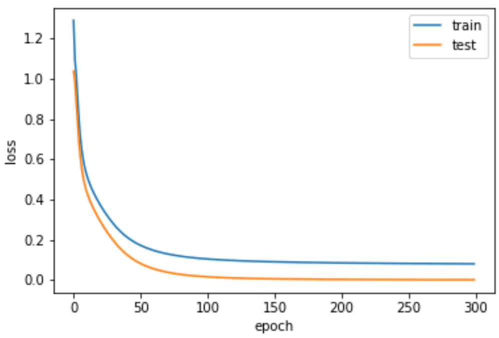


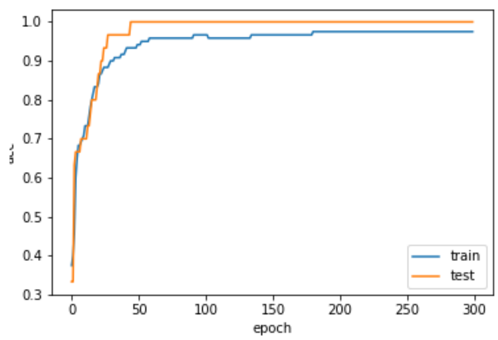
②Sigmoid 激活函数

Sigmoid

In [33]:
```
main(activation='sigmoid')
```

```
sigmoid
Eopch: 49  Loss: 0.17654  ACC: 0.94167
Test  Loss: 0.08647  ACC: 1.00000
Eopch: 99  Loss: 0.10605  ACC: 0.96667
Test  Loss: 0.01740  ACC: 1.00000
Eopch: 149  Loss: 0.09164  ACC: 0.96667
Test  Loss: 0.00698  ACC: 1.00000
Eopch: 199  Loss: 0.08612  ACC: 0.97500
Test  Loss: 0.00404  ACC: 1.00000
Eopch: 249  Loss: 0.08328  ACC: 0.97500
Test  Loss: 0.00288  ACC: 1.00000
Eopch: 299  Loss: 0.08156  ACC: 0.97500
Test  Loss: 0.00230  ACC: 1.00000
```



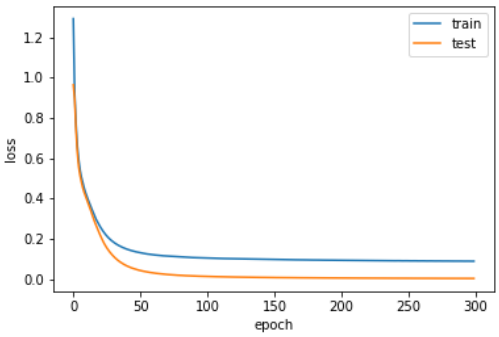


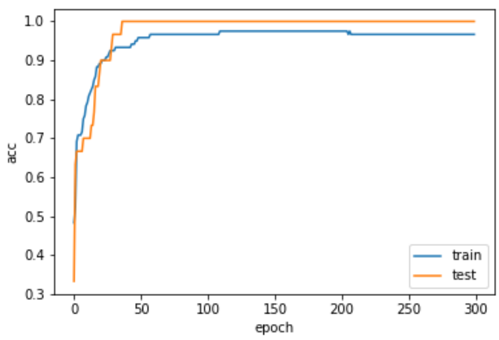
③Tanh 激活函数

Tanh

In [34]:
```
main(activation='tanh')
```

```
tanh
Eopch: 49  Loss: 0.13242  ACC: 0.95833
Test  Loss: 0.04444  ACC: 1.00000
Eopch: 99  Loss: 0.10482  ACC: 0.96667
Test  Loss: 0.01381  ACC: 1.00000
Eopch: 149  Loss: 0.09775  ACC: 0.97500
Test  Loss: 0.00809  ACC: 1.00000
Eopch: 199  Loss: 0.09370  ACC: 0.97500
Test  Loss: 0.00568  ACC: 1.00000
Eopch: 249  Loss: 0.09114  ACC: 0.96667
Test  Loss: 0.00439  ACC: 1.00000
Eopch: 299  Loss: 0.08918  ACC: 0.96667
Test  Loss: 0.00355  ACC: 1.00000
```





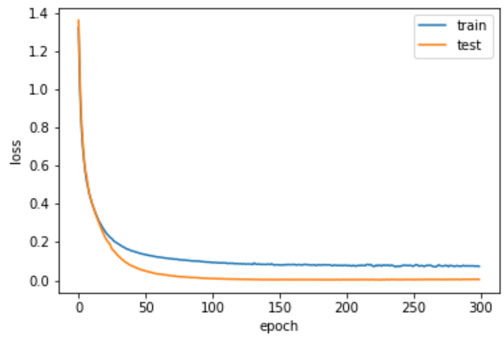
④Prelu 激活函数

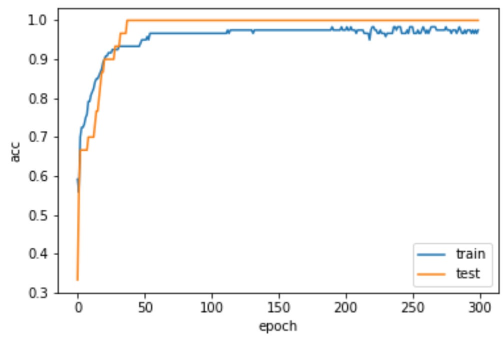
比较发现，PReLU 的效果优于 ReLU，PReLU 收敛速度表现最好。Sigmoid 在训练集 loss 上的收敛速度最慢，Tanh 的效果位于最优与最差之间。

Prelu

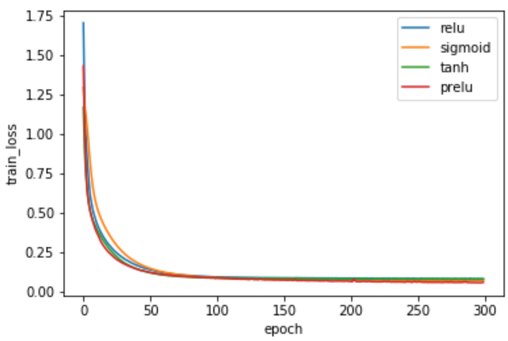
```
In [35]: main(activation='prelu')
```

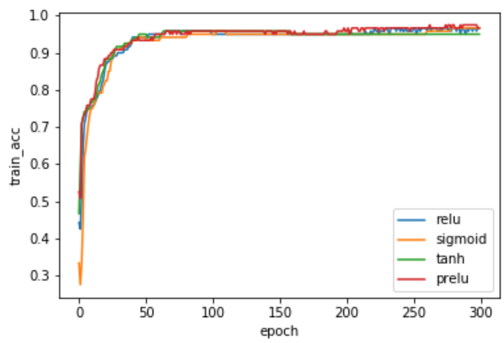
```
prelu
Eopch: 49  Loss: 0.13603  ACC: 0.95000
Test  Loss: 0.05043  ACC: 1.00000
Eopch: 99  Loss: 0.09408  ACC: 0.96667
Test  Loss: 0.00955  ACC: 1.00000
Eopch: 149  Loss: 0.08165  ACC: 0.97500
Test  Loss: 0.00369  ACC: 1.00000
Eopch: 199  Loss: 0.07883  ACC: 0.97500
Test  Loss: 0.00364  ACC: 1.00000
Eopch: 249  Loss: 0.07481  ACC: 0.98333
Test  Loss: 0.00432  ACC: 1.00000
Eopch: 299  Loss: 0.07269  ACC: 0.97500
Test  Loss: 0.00493  ACC: 1.00000
```

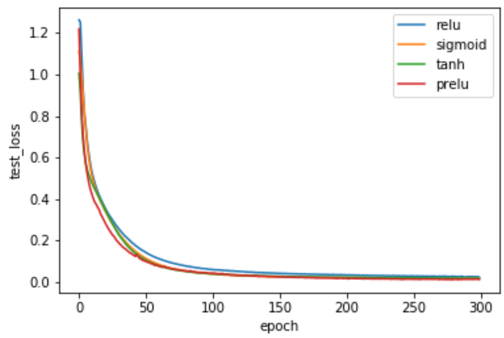


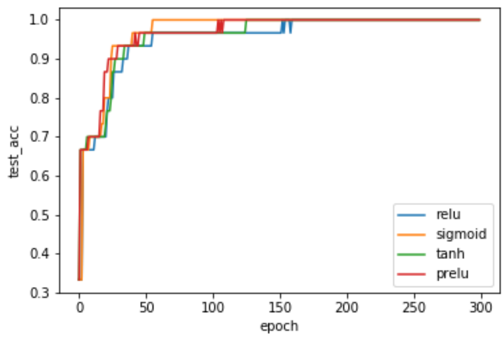


比较四种激活函数的影响：





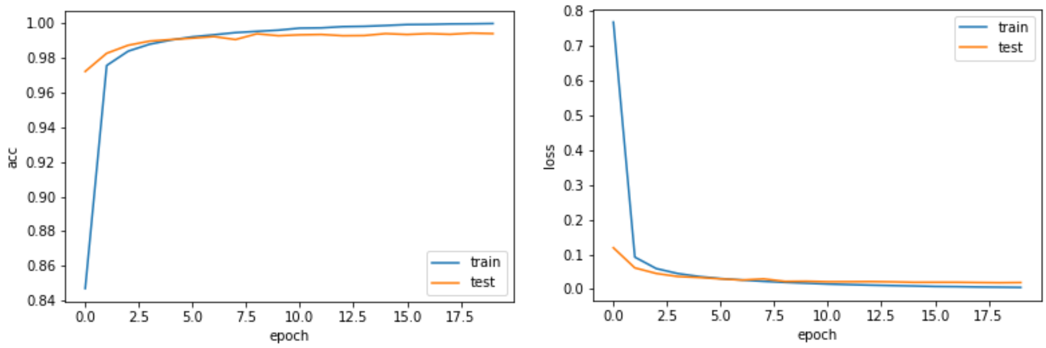




四种激活函数最终都能得到较好的结果，测试集上的准确率都稳定在 1.00
比较发现，PReLU 的效果优于 ReLU，PReLU 收敛速度表现最好。Sigmoid 在训练集 loss 上的收敛速度最慢，Tanh 的效果位于最优与最差之间。

2）设计并实现一个深度学习网络结构，能够在 MNIST 数据集上(前 6 万个训练，后 1 万个测试)获得至少 99%的测试精度

源代码 mnist_pytorch.ipynb



左图为 20 个 epoch 上训练数据和测试数据的测试精度曲线
右图为 20 个 epoch 上训练数据和测试数据的损失函数值曲线

```
In [12]: test_model(model, train_loader)
         Loss: 0.00450  ACC: 0.99970

In [13]: test_model(model, test_loader)
         Loss: 0.01848  ACC: 0.99399
```

上图为训练集和测试集在测试 acc 最高的 model 上的损失值与精度

```
Eopch: 10  Loss: 0.01469  ACC: 0.99684
Test  Loss: 0.02128  ACC: 0.99299
Eopch: 11  Loss: 0.01301  ACC: 0.99700
Test  Loss: 0.02137  ACC: 0.99319
Eopch: 12  Loss: 0.01147  ACC: 0.99770
Test  Loss: 0.02173  ACC: 0.99249
Eopch: 13  Loss: 0.01022  ACC: 0.99788
Test  Loss: 0.02090  ACC: 0.99259
Eopch: 14  Loss: 0.00908  ACC: 0.99835
Test  Loss: 0.01962  ACC: 0.99369
Eopch: 15  Loss: 0.00766  ACC: 0.99890
Test  Loss: 0.01971  ACC: 0.99319
Eopch: 16  Loss: 0.00712  ACC: 0.99900
Test  Loss: 0.01968  ACC: 0.99369
Eopch: 17  Loss: 0.00621  ACC: 0.99927
Test  Loss: 0.01897  ACC: 0.99329
Eopch: 18  Loss: 0.00564  ACC: 0.99933
Test  Loss: 0.01848  ACC: 0.99399
Eopch: 19  Loss: 0.00495  ACC: 0.99950
Test  Loss: 0.01897  ACC: 0.99369
```

上图为 epoch10~19 训练集和测试集上的准确率，可见准确率稳定在 99.3%左右

```
Out[4]: Net(
          (layer1): Sequential(
            (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
            (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU()
          )
          (layer2): Sequential(
            (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
            (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU()
            (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
          )
          (layer3): Sequential(
            (0): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1))
            (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU()
          )
          (layer4): Sequential(
            (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
            (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU()
          )
          (layer5): Sequential(
            (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU()
            (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
          )
          (fc): Sequential(
            (0): Linear(in_features=1024, out_features=128, bias=True)
            (1): ReLU()
            (2): Linear(in_features=128, out_features=128, bias=True)
            (3): ReLU()
            (4): Linear(in_features=128, out_features=10, bias=True)
          )
        )
```

上图为使用的网络结构

该网络结构启发于 VGG Net，不同的是 VGG 是用于224×224的图片，而对于 mnist 这样28×28的小尺寸图片，需要减少卷积层的数量。

首先两个卷积层$Conv2d(1,16,kernel_size = 3)$，$(1,28,28) \rightarrow (16,26,26)$

$Conv2d(16,16,kernel\_size = 3)$，$(16,26,26) \rightarrow (16,24,24)$

池化层$MaxPool2d(kernel_size = 2, stride = 2)$，$(16,24,24) \rightarrow (16,12,12)$

接着三个卷积层$Conv2d(16,64,kernel_size = 3)$，$(16,12,12) \rightarrow (64,12,12)$

$Conv2d(64,64,kernel\_size = 3)$，$(64,12,12) \rightarrow (64,10,10)$

$Conv2d(64,64,kernel\_size = 3)$，$(64,10,10) \rightarrow (64,8,8)$

池化层$MaxPool2d(kernel_size = 2, stride = 2)$，$(64,8,8) \rightarrow (64,4,4)$

最后为三个全连接层$64 \times 4 \times 4 \rightarrow 128 \rightarrow 128 \rightarrow 10$

实践题二

1）仅使用 numpy 实现三层神经网络 BP 训练算法(输入 d 维向量，中间 h 个隐含神经元， 输出 c>1 类单热向量编码，隐含层使用 sigmoid 激活函数，输入输出层使用线性激活函数)，损失函数用均方误差或者交叉熵。

源代码$threelayernet.py$

初始化网络输入$input_size, hidden_size, output_size$，分别为输入的维数、隐含层神经元的个数以及输出的维度

```
self.params = {}
self.params['W1'] = std * np.random.randn(input_size, hidden_size)
self.params['b1'] = np.zeros(hidden_size)
self.params['W2'] = std * np.random.randn(hidden_size, output_size)
self.params['b2'] = np.zeros(output_size)
```

权值和偏移量参数存储在$params$字典中，初始化网络时权值随机设置，偏移量取 0

$W1$的 shape 为(D,H)，$b1$的 shape 为$(H,)$，$W2$的 shape 为(H,C)，$b2$的 shape 为$(C,)$

核心部分为$\boldsymbol{loss}$**函数**，函数中实现了前向传播和反向传播，函数返回损失值以及求得的梯度

使用的损失函数为$MSELoss$

```
def MSELoss(self, predict, ground_truth):
    return np.sum(0.5*(predict-ground_truth)**2)
```

前向传播

前向传播通过一个全连接层，再通过一个$sigmoid$激活函数，最后通过一个全连接层得到输出

```
# Forward
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']

a1 = X
z1 = np.dot(a1, W1) + b1
a2 = self.sigmoid(z1)
z2 = np.dot(a2, W2) + b2

loss = self.MSELoss(z2, y)
```

反向传播

根据链式求导法则，$下游梯度 = 上游梯度 \times 本地梯度$，计算出$W1, b1, W2, b2$的梯度值，并存储在$grads$字典中，便于后续梯度下降时读取

```
# Backward pass: compute gradients
```

```
grads = {}
dW2 = np.dot(a2.T, z2 - y)
dW2 += reg * dW2
grads['W2'] = dW2
grads['b2'] = np.sum(z2 - y, axis=0)

da2 = np.dot((z2 - y), W2.T)
dz1 = np.dot(da2, self.derivative_sigmoid(z1))
dW1 = np.dot(a1.T, dz1)
dW1 += reg * dW1
grads['W1'] = dW1
grads['b1'] = np.sum(dz1, axis=0)
```

***train*函数**将训练数据按照 batch_size 分好进行训练，从$loss$中获取梯度信息进行梯度下降修改参数

```
# Gradient Descent

dW1, db1 = grads['W1'], grads['b1']
dW2, db2 = grads['W2'], grads['b2']

self.params['W1'] -= learning_rate * dW1
self.params['b1'] -= learning_rate * db1
self.params['W2'] -= learning_rate * dW2
self.params['b2'] -= learning_rate * db2
```

***predict*函数**进行预测，从$params$中获取参数，进行前向传播得到预测的标签值，再将预测值转成$one-hot$编码，返回预测值$pred$

```
# one-hot
pred = np.zeros([y_pred.shape[0],self.output_size])
for i in range(y_pred.shape[0]):
    pred[i][int(y_pred[i])] = 1
```

2）在 iris 数据集上对 1)中实现的算法测试，并与实践题一的结果进行比较

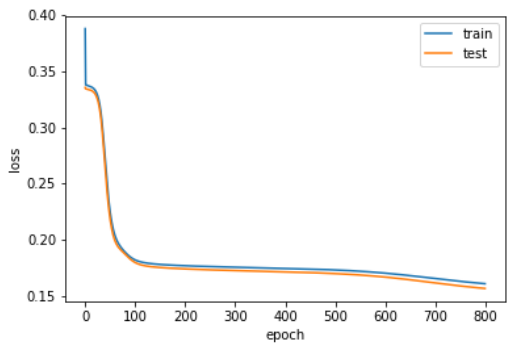
源代码 iris_numpy.ipynb

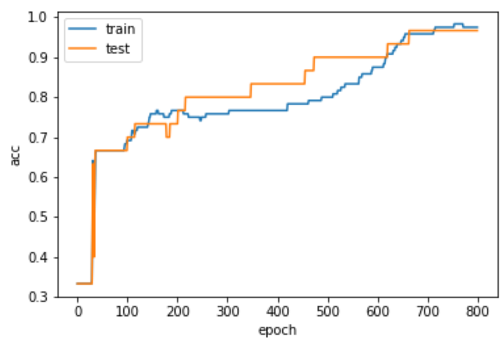
```
model = ThreeLayerNet(input_size, 40, output_size)
```

model 的隐藏层为 40 个神经元，训练学习率设置为0.001，$batch_size$大小为 5，NUM_EPOCH为 800，训练结果如下
最终准确率稳定在 0.96 左右

In [17]:
```python
NUM_EPOCH = 800
learning_rate = 1e-3
batch_size = 5

train_loss = []
train_acc = []
test_loss = []
test_acc = []
for idx in range(NUM_EPOCH):
    train_res = model.train(train_data, train_label, learning_rate, batch_size, reg=1e-3)
    train_loss.append(train_res['loss'])
    train_acc.append(train_res['train_acc'])

    test_res = model.test(test_data, test_label)
    test_loss.append(test_res['loss'])
    test_acc.append(test_res['test_acc'])

    if (idx + 1) % 50 == 0:
        print('Epoch: %d  Train loss: %.3f  acc: %.3f  Test loss: %.3f acc: %.3f' % (idx, train_res['loss'], train_res['train_acc'], test_
```

```
Epoch: 49  Train loss: 0.228  acc: 0.667  Test loss: 0.221 acc: 0.667
Epoch: 99  Train loss: 0.182  acc: 0.692  Test loss: 0.180 acc: 0.667
Epoch: 149  Train loss: 0.178  acc: 0.758  Test loss: 0.176 acc: 0.733
Epoch: 199  Train loss: 0.177  acc: 0.767  Test loss: 0.174 acc: 0.733
Epoch: 249  Train loss: 0.176  acc: 0.750  Test loss: 0.173 acc: 0.800
Epoch: 299  Train loss: 0.176  acc: 0.758  Test loss: 0.173 acc: 0.800
Epoch: 349  Train loss: 0.175  acc: 0.767  Test loss: 0.172 acc: 0.833
Epoch: 399  Train loss: 0.175  acc: 0.767  Test loss: 0.172 acc: 0.833
Epoch: 449  Train loss: 0.174  acc: 0.783  Test loss: 0.171 acc: 0.833
Epoch: 499  Train loss: 0.173  acc: 0.800  Test loss: 0.170 acc: 0.900
Epoch: 549  Train loss: 0.172  acc: 0.833  Test loss: 0.169 acc: 0.900
Epoch: 599  Train loss: 0.171  acc: 0.875  Test loss: 0.167 acc: 0.900
Epoch: 649  Train loss: 0.168  acc: 0.942  Test loss: 0.165 acc: 0.933
Epoch: 699  Train loss: 0.166  acc: 0.958  Test loss: 0.162 acc: 0.967
Epoch: 749  Train loss: 0.163  acc: 0.975  Test loss: 0.159 acc: 0.967
Epoch: 799  Train loss: 0.161  acc: 0.975  Test loss: 0.157 acc: 0.967
```





左图为 800 个 epoch 的 loss，右图为对应的准确率，最终模型训练可达到 0.96 左右的准确率，与第一题中用 pytorch 实现的准确率相比较低，此外收敛速度也慢了不少，准确率曲线也没有 pytorch 实现的平滑。分析发现，在本题中用 numpy 实现的三层网络中，由于使用的是梯度下降法更新参数，与 pytorch 使用的 SGD 和 Adam 等 optimizer 相比，收敛速度慢，且容易在局部最小值处*stuck*，因此效果不及 pytorch 实现的网络。

总结

这次作业第一题用现有的深度学习库 pytorch 搭了网络并在 iris 和 MNIST 数据集上进行了训练，通过第一题的实践学习了深度网络结构的构建、调参等，使用不同的激活函数使得对激活函数的作用有了深入的理解。第二题用 numpy 实现三层网络，关键之处在于前向传播、反向传播以及损失函数，通过实践加深了对课堂上学到的这些理论的印象和理解。