# 研究生《深度学习》课程
# 实验报告

| | |
|---|---|
| 实验名称： | PyTorch 基本操作实验 |
| 姓　　名： | 郑楚彬 |
| 学　　号： | 21140129 |
| 上课类型： | 专业课 |
| 日　　期： | 2021.08.07 |

## 一、实验内容

## 二、实验设计

若实验内容皆为指定内容，则此部分则可省略；若实验内容包括自主设计模型等内容，则需要在此部分写明设计思路、流程，并画出模型图并使用相应的文字进行描述。

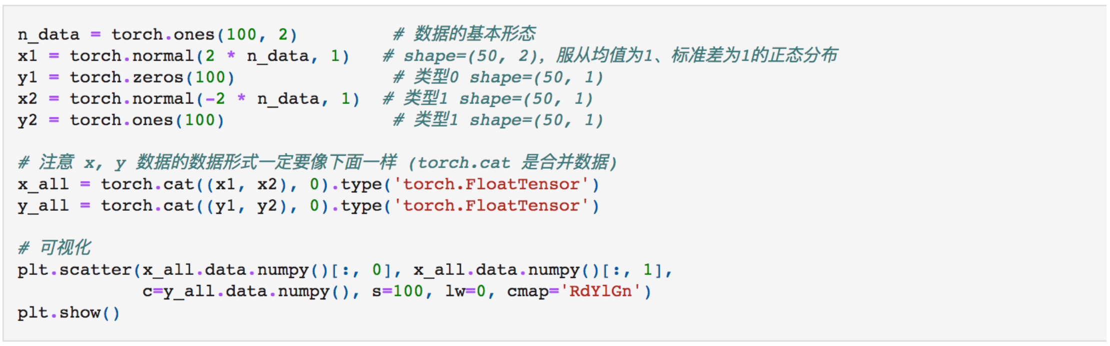
## 三、实验环境及实验数据集

实验环境：

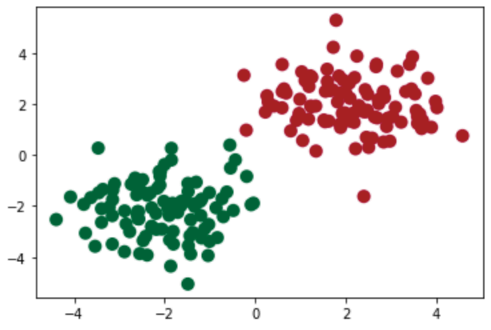
macOs10.13.6、Pytorch 1.5.0、Jupyter Notebook

数据集：

（1）人工构造的数据集（200，2）



```
n_data = torch.ones(100, 2)         # 数据的基本形态
x1 = torch.normal(2 * n_data, 1)   # shape=(50, 2), 服从均值为1、标准差为1的正态分布
y1 = torch.zeros(100)               # 类型0 shape=(50, 1)
x2 = torch.normal(-2 * n_data, 1)  # 类型1 shape=(50, 1)
y2 = torch.ones(100)                # 类型1 shape=(50, 1)

# 注意 x, y 数据的数据形式一定要像下面一样 (torch.cat 是合并数据)
x_all = torch.cat((x1, x2), 0).type('torch.FloatTensor')
y_all = torch.cat((y1, y2), 0).type('torch.FloatTensor')

# 可视化
plt.scatter(x_all.data.numpy()[:, 0], x_all.data.numpy()[:, 1],
            c=y_all.data.numpy(), s=100, lw=0, cmap='RdYlGn')
plt.show()
```



（2）Fashion-MNIST 数据集(一个多类图像分类数据集)

训练集:60,000

测试集:10,000

每个样本的数据格式为: 28 × 28 × 1 (高,宽,通道)

类别(10 类): dress(连衣衣裙)、coat(外套)、sandal(凉鞋)、shirt(衬衫)、sneaker(运动鞋)、bag(包)和 ankle boot(短靴)

```python
import torch
import torchvision
import torchvision.transforms as transforms

dataset_dir = '/Users/zhengchubin/PycharmProjects/learn/data/'

# 下载不了数据集
# mnist_train = torchvision.datasets.FashionMNIST(root=dataset_dir, train=True, download=True, transform=transforms.ToTensor())
# mnist_test = torchvision.datasets.FashionMNIST(root=dataset_dir, train=False, download=True, transform=transforms.ToTensor())

# 手动下载数据集并读取
mnist_train = torchvision.datasets.FashionMNIST(root=dataset_dir, train=True, transform=transforms.ToTensor())
mnist_test = torchvision.datasets.FashionMNIST(root=dataset_dir, train=False, transform=transforms.ToTensor())

mnist_train, mnist_train.data.shape, mnist_train.targets.shape
```

```
(Dataset FashionMNIST
     Number of datapoints: 60000
     Root location: /Users/zhengchubin/PycharmProjects/learn/data/
     Split: Train
     StandardTransform
 Transform: ToTensor(),
 torch.Size([60000, 28, 28]),
 torch.Size([60000]))
```

# 四、实验过程

## 1.1 PyTorch 基本操作实验

（1）减法操作

前两种形式的减法操作不改变原有矩阵的值，由于广播机制，两个操作的形状可以不同；但原地操作的减法不允许张量由于广播而改变形状，因而会抛出异常。

```
M = torch.tensor([[4, 5, 6]])
N = torch.tensor([[1], [2]])
M, N
```

```
(tensor([[4, 5, 6]]),
 tensor([[1],
         [2]]))
```

```
M - N
```

```
tensor([[3, 4, 5],
        [2, 3, 4]])
```

```
torch.sub(M, N)
```

```
tensor([[3, 4, 5],
        [2, 3, 4]])
```

```
# 非原地操作，减法操作后返回新矩阵，M不变
M.sub(N), M
```

```
(tensor([[3, 4, 5],
         [2, 3, 4]]),
 tensor([[4, 5, 6]]))
```

```
# 由于广播机制，计算后的矩阵形状为(2,3)，而 M 为(1,3)，无法进行原地操作，因而报错
M.sub_(N)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-6-ebcaa8933493> in <module>
      1 # 由于广播机制，计算后的矩阵形状为(2,3)，而 M 为(1,3)，无法进行原地操作，因而报错
----> 2 M.sub_(N)
      3 

RuntimeError: output with shape [1, 3] doesn't match the broadcast shape [2, 3]
```

（2）矩阵求内积

经查相关资料，矩阵内积的定义：两个行数和列数均相同的矩阵，对应元素相乘再求和。即两个矩阵乘积得到的矩阵的迹。但本题两个矩阵维度不同，不知道咋弄，权当矩阵乘法计算。

```
P = torch.normal(mean=0.0, std=0.01, size=(3, 2))
Q = torch.normal(mean=0.0, std=0.01, size=(4, 2))
P, Q
```

```
(tensor([[-0.0009, -0.0095],
         [-0.0035,  0.0057],
         [-0.0013, -0.0051]]),
 tensor([[-0.0138, -0.0030],
         [-0.0037, -0.0032],
         [ 0.0116, -0.0154],
         [ 0.0029,  0.0062]]))
```

```
Q_T = Q.T
Q_T
```

```
tensor([[-0.0138, -0.0037,  0.0116,  0.0029],
        [-0.0030, -0.0032, -0.0154,  0.0062]])
```

```
# 相当于矩阵乘积
torch.mm(P, Q_T)
```

```
tensor([[ 4.0993e-05,  3.3273e-05,  1.3523e-04, -6.1042e-05],
        [ 3.0718e-05, -4.9176e-06, -1.2764e-04,  2.5079e-05],
        [ 3.2510e-05,  2.0637e-05,  6.3133e-05, -3.4767e-05]])
```

（3）梯度计算

由于梯度的计算是累加的，若中断了梯度的追踪，则默认丢弃了这部分的值，从而计算出的梯度也是不正确的。

```
x = torch.ones(1,requires_grad=True)
y1 = x**2

with torch.no_grad():
    y2 = x**3

y3 = y1 + y2
y3.backward()
x.grad
```

```
tensor([2.])
```

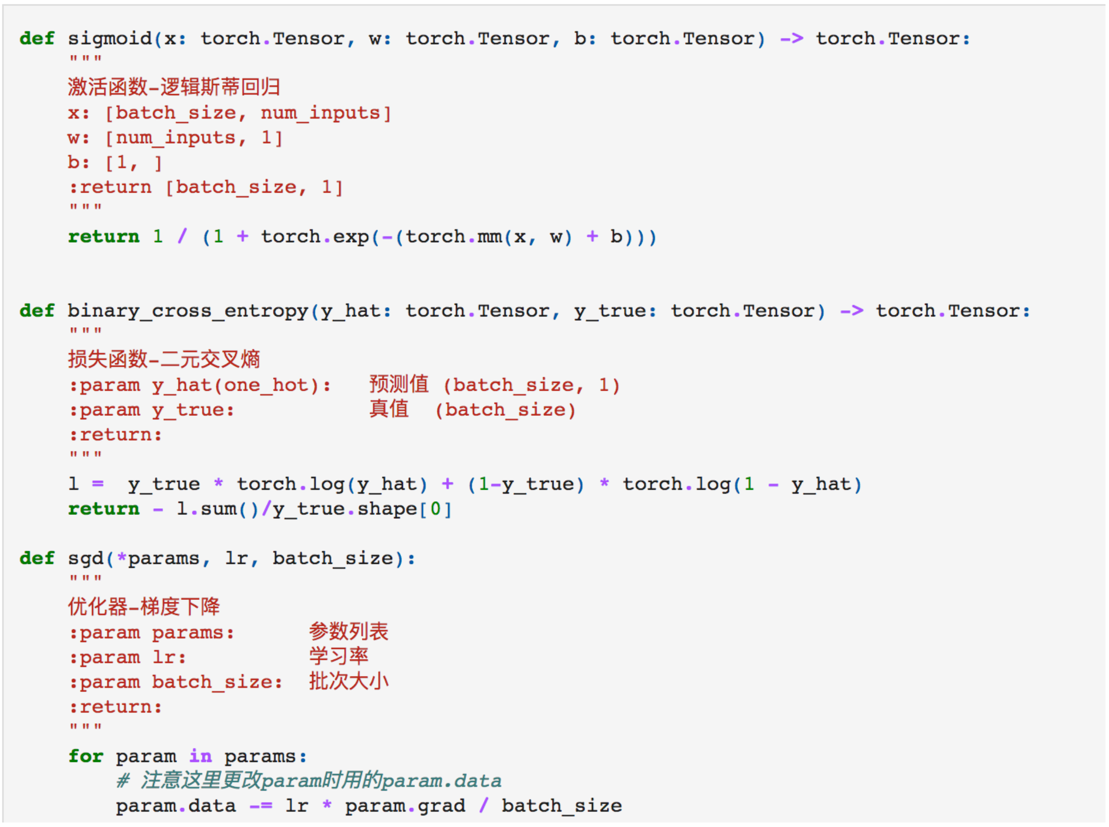
```
x = torch.ones(1,requires_grad=True)
y1 = x**2
y2 = x**3

y3 = y1 + y2
y3.backward()
x.grad
```

```
tensor([5.])
```

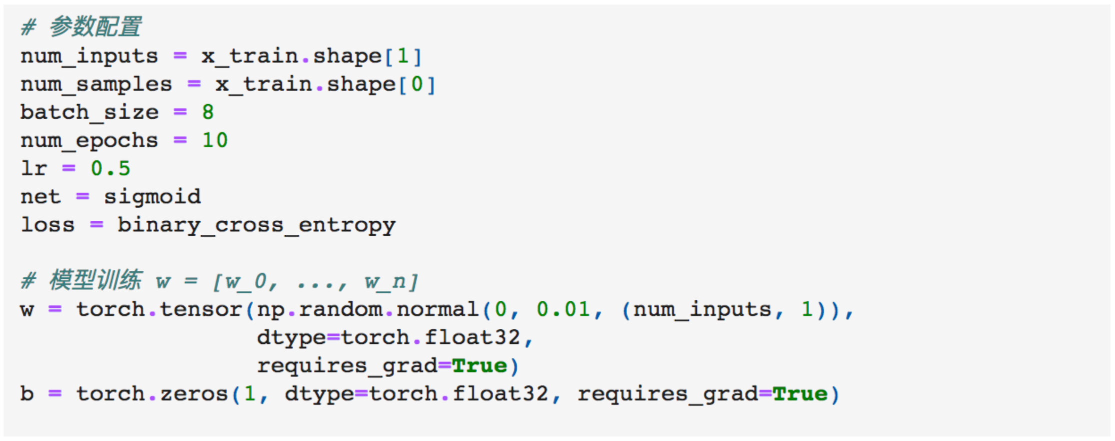
## 1.2 Logistic 回归实验

（1）动手从 0 实现 logistic 回归

　　人工构造的数据集服从正态分布，标签之间的区分较为明显，从而模型学习的效果非常好，迭代次数为 10 时 loss 就降到 0.03 左右，测试集的准确率高达 100%。



```
def sigmoid(x: torch.Tensor, w: torch.Tensor, b: torch.Tensor) -> torch.Tensor:
    """
    激活函数-逻辑斯蒂回归
    x: [batch_size, num_inputs]
    w: [num_inputs, 1]
    b: [1, ]
    :return [batch_size, 1]
    """
    return 1 / (1 + torch.exp(-(torch.mm(x, w) + b)))


def binary_cross_entropy(y_hat: torch.Tensor, y_true: torch.Tensor) -> torch.Tensor:
    """
    损失函数-二元交叉熵
    :param y_hat(one_hot):   预测值 (batch_size, 1)
    :param y_true:           真值  (batch_size)
    :return:
    """
    l =  y_true * torch.log(y_hat) + (1-y_true) * torch.log(1 - y_hat)
    return - l.sum()/y_true.shape[0]

def sgd(*params, lr, batch_size):
    """
    优化器-梯度下降
    :param params:      参数列表
    :param lr:          学习率
    :param batch_size:  批次大小
    :return:
    """
    for param in params:
        # 注意这里更改param时用的param.data
        param.data -= lr * param.grad / batch_size
```



```
# 参数配置
num_inputs = x_train.shape[1]
num_samples = x_train.shape[0]
batch_size = 8
num_epochs = 10
lr = 0.5
net = sigmoid
loss = binary_cross_entropy

# 模型训练 w = [w_0, ..., w_n]
w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, 1)),
                 dtype=torch.float32,
                 requires_grad=True)
b = torch.zeros(1, dtype=torch.float32, requires_grad=True)
```

```python
for epoch in range(num_epochs):

    # 读取数据集
    iter_train = data_iter(x_train, y_train, batch_size=batch_size)

    # 一批次的训练数据
    for X, y_true in iter_train:

        # 模型预测值
        y_hat = net(X, w, b)

        # 损失值
        # (batch_size, 1) => (batch_size, )
        y_hat = y_hat.view(y_true.size())
        l = loss(y_hat, y_true)

        # 反向传播
        l.backward()

        # 随机梯度下降
        sgd(w, b, lr=lr, batch_size=batch_size)

        # 梯度置零
        w.grad.data.zero_()
        b.grad.data.zero_()

    y_hat = net(x_train, w, b)
    y_hat = y_hat.view(y_train.size())
    train_l = loss(y_hat, y_train)
    print('epoch %d, loss %f' % (epoch + 1, train_l))

w, b
```

```
epoch 1, loss 0.147422
epoch 2, loss 0.090928
epoch 3, loss 0.068785
epoch 4, loss 0.056685
epoch 5, loss 0.048948
epoch 6, loss 0.043523
epoch 7, loss 0.039480
epoch 8, loss 0.036333
epoch 9, loss 0.033803
epoch 10, loss 0.031717
```

```python
# 评估-准确率
def accuracy_count(y_hat: torch.Tensor, y: torch.Tensor):
    mask = y_hat > 0.5
    y_hat[mask] = 1
    y_hat[~mask] = 0
    return (y_hat == y).sum()

iter_test = data_iter(x_test, y_test, batch_size=batch_size)
acc_sum, n = 0.0, 0
for X, y in iter_test:

    y_hat = net(X, w, b)
    y_hat = y_hat.view(y.size())

    acc_sum += accuracy_count(y_hat, y)
    n += y.shape[0]
acc_sum = acc_sum / n

print('accuracy =', acc_sum)
```

```
accuracy = tensor(1.)
```

（2）torch.nn 实现 logistic 回归

```python
class LogisticRegressionModel(torch.nn.Module):
    """ 逻辑斯蒂回归模型 """
    def __init__(self, num_features):
        super(LogisticRegressionModel, self).__init__()
        self.linear = torch.nn.Linear(num_features, 1)

    def forward(self, X):
        return torch.sigmoid(self.linear(X))

# 参数配置
num_samples = x_train.shape[0]
num_features = x_train.shape[1]
batch_size = 8
num_epochs = 10
lr = 0.03

net = LogisticRegressionModel(num_features)
loss = torch.nn.BCELoss()
optimizer = optim.SGD(net.parameters(), lr=lr)

for epoch in range(num_epochs):

    # 读取数据集
    iter_train = data_iter(x_train, y_train, batch_size=batch_size)

    # 一批次的训练数据
    for X, y_true in iter_train:

        # 模型预测值
        y_hat = net(X)
        y_hat = y_hat.view(y_true.size())

        # 损失值
        l = loss(y_hat, y_true)

        # 反向传播
        l.backward()

        optimizer.step()

    y_hat = net(x_train)
    y_hat = y_hat.view(y_train.size())
    train_l = loss(y_hat, y_train)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().item()))
```

```
epoch 1, loss 0.033113
epoch 2, loss 0.012771
epoch 3, loss 0.009272
epoch 4, loss 0.007705
epoch 5, loss 0.006359
epoch 6, loss 0.005003
epoch 7, loss 0.003706
epoch 8, loss 0.002586
epoch 9, loss 0.001717
epoch 10, loss 0.001103
tensor(1.)
```

```python
# 评估：准确率
iter_test = data_iter(x_test, y_test, batch_size=batch_size)
acc_sum, n = 0.0, 0
for X, y in iter_test:
    y_hat = net(X)
    y_hat = y_hat.view(y.size())

    acc_sum += accuracy_count(y_hat, y)
    n += y.shape[0]

print(acc_sum/n)
```

## 1.3 Softmax 回归实验

（1）动手从 0 实现 Softmax 回归

整体而言，softmax 在 Fashion-MNIST 数据集上的表现不错，迭代次数为 5 时，测试集的准确率就能达到 80%以上，若加大迭代次数使模型收敛，估计准确率在 90%以上也不是问题。

```python
def softmax(x: torch.Tensor, w: torch.Tensor, b: torch.Tensor):
    """
    激活函数
    x: [batch_size, num_features]
    w: [num_features, num_classes]
    b: [num_classes, ]
    :return [batch_size, num_classes]
    """
    exp_ = torch.exp(torch.mm(x, w) + b)
    exp_sum = exp_.sum(dim=1, keepdim=True)
    return exp_ / exp_sum


def cross_entropy(y_hat: torch.Tensor, y_true: torch.Tensor) -> torch.Tensor:
    """
    损失函数-交叉熵
    :param y_hat(one_hot):   预测值 (batch_size, class_num)
    :param y_true:           真值  (batch_size)
    :return:
    """
    return -torch.log(y_hat.gather(dim=1, index=y_true.view(-1, 1)))

def sgd(*params, lr, batch_size):
    """
    优化器-梯度下降
    :param params:      参数列表
    :param lr:          学习率
    :param batch_size:  批次大小
    :return:
    """
    for param in params:
        # 注意这里更改param时用的param.data
        param.data -= lr * param.grad / batch_size

# 参数配置
num_features = 28 * 28
num_classes = 10
batch_size = 256
num_epochs = 5
lr = 0.1
net = softmax
loss = cross_entropy

# 模型训练 w = [w_0, ..., w_n]
w = torch.tensor(np.random.normal(0, 0.01, (num_features, num_classes)),
                 dtype=torch.float32,
                 requires_grad=True)
b = torch.zeros(num_classes, dtype=torch.float32, requires_grad=True)
```

```python
for epoch in range(num_epochs):

    # 读取数据集
    iter_train = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True)

    # 一批次的训练数据
    l_sum, n = 0.0, 0
    for X, y_true in iter_train:

        # 转换
        X = X.view(-1, num_features)

        # 模型预测值
        y_hat = net(X, w, b)

        # 损失值
        l = loss(y_hat, y_true).sum()

        # 反向传播
        l.backward()

        # 随机梯度下降
        sgd(w, b, lr=lr, batch_size=batch_size)

        # 梯度置零
        w.grad.data.zero_()
        b.grad.data.zero_()

        l_sum += l
        n += y_true.shape[0]

    print('epoch %d, loss %f' % (epoch + 1, l_sum/n))
```

```
epoch 1, loss 0.784691
epoch 2, loss 0.571183
epoch 3, loss 0.525277
epoch 4, loss 0.501656
epoch 5, loss 0.484789
```

```python
# 评估：准确率
iter_test = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=False)

acc_sum, n = 0.0, 0
for X, y in iter_test:
    X = X.view(-1, num_features)
    y_hat: torch.Tensor = net(X, w, b)
    y_hat = y_hat.argmax(dim=1)
    acc_sum += (y_hat == y).float().sum().item()
    n += y.shape[0]

acc_avg = acc_sum / n
acc_avg
```

```
0.8243
```

（2）torch.nn 实现 softmax 回归

一开始分开定义 **softmax** 运算和交叉熵损失函数，但模型在测试集上的准确率始终不高且不稳定，仅 60%~70%之间，后经查阅相关资料，才明白分开定义可能会造成数值不稳定。因此，PyTorch 提供了一个包括 **softmax** 运算和交叉熵损失计算的函数。它的数值稳定性更好。所以在输出层，我们不需要再进行 **softmax** 操作，从而最后的模型也较为稳定，准确率也在 80%以上。

```python
class SoftmaxRegressionModel(torch.nn.Module):
    """ SoftMax回归模型 """
    def __init__(self, num_features, num_classes):
        super(SoftmaxRegressionModel, self).__init__()
        self.num_features = num_features
        self.linear = torch.nn.Linear(num_features, num_classes)
        # 初始化
        nn.init.normal_(self.linear.weight, mean=0, std=0.01)
        nn.init.constant_(self.linear.bias, val=0)

    def forward(self, X):
        """
        :param X: [batch_size, num_features]
        :return:  [batch_size, num_classes]
        """
        # (,1,28,28) => [,784]
        X = X.view(X.shape[0], -1)
        # return torch.softmax(self.linear(X), dim=1)
        return self.linear(X)
```

```python
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

# 参数配置
num_features = 28 * 28
num_classes = 10
batch_size = 256
num_epochs = 5
lr = 0.1

net = SoftmaxRegressionModel(num_features, num_classes)
# net = torch.nn.Sequential(Flatten(),
#                           nn.Linear(num_features, num_classes))
# net.apply(init_weights)

loss = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=lr)
```

```python
for epoch in range(num_epochs):

    # 读取数据集
    iter_train = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True)

    # 一批次的训练数据
    l_sum, n = 0.0, 0
    for X, y_true in iter_train:

        # 模型预测值
        y_hat = net(X)

        # 损失值
        l = loss(y_hat, y_true)

        # 反向传播
        optimizer.zero_grad() #清空梯度
        l.backward()
        optimizer.step()


        l_sum += l
        n += y_true.shape[0]

    print('epoch %d, loss %f' % (epoch + 1, l_sum/n))
```

```
epoch 1, loss 0.003076
epoch 2, loss 0.002234
epoch 3, loss 0.002054
epoch 4, loss 0.001961
epoch 5, loss 0.001899
```

```
# 评估：准确率
iter_test = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=False)

acc_sum, n = 0.0, 0
for X, y in iter_test:
    y_hat: torch.Tensor = net(X)
    y_hat = y_hat.argmax(dim=1)
    acc_sum += (y_hat == y).float().sum().item()
    n += y.shape[0]

acc_avg = acc_sum / n
acc_avg
```

```
0.8181
```

## 五、实验结果

## 六、实验心得体会

本次实验加深了对深度学相关模型的理解，动手从 0 实现 logistic 回归、softmax 回归，从而熟悉和了解深度学习模型从准备数据集、训练、评估的全流程，收获非常大。

## 七、参考文献

## 八、附录

需要补充说明的内容，如无可略。

# 实验报告编写要求

1. 正文要求小四号宋体，行间距 1.5 倍；

2. 英文要求小四号 Times New Roman；

3. 在实验内容、实验过程、实验结果三部分需要针对当次实验不同的实验内容分别填写（模版以实验一为例），实验设计中如有必要也可以分开填写；

4. 实验报告配图的每幅图应有编号和标题，编号和标题应位于图下方处，居中，中文用五号宋体；

5. 表格应为三线表，每个表格应有编号和标题，编号和标题应写在表格上方正中，距正文段前 0.5 倍行距。表格中量与单位之间用“/”分隔，编号与标题中的中文用五号宋体；

6. 图、表、公式、算式等，一律用阿拉伯数字分别依序连续编排序号。其标注形式应便于互相区别，可分别为：图 1、表 2、公式(5)等。