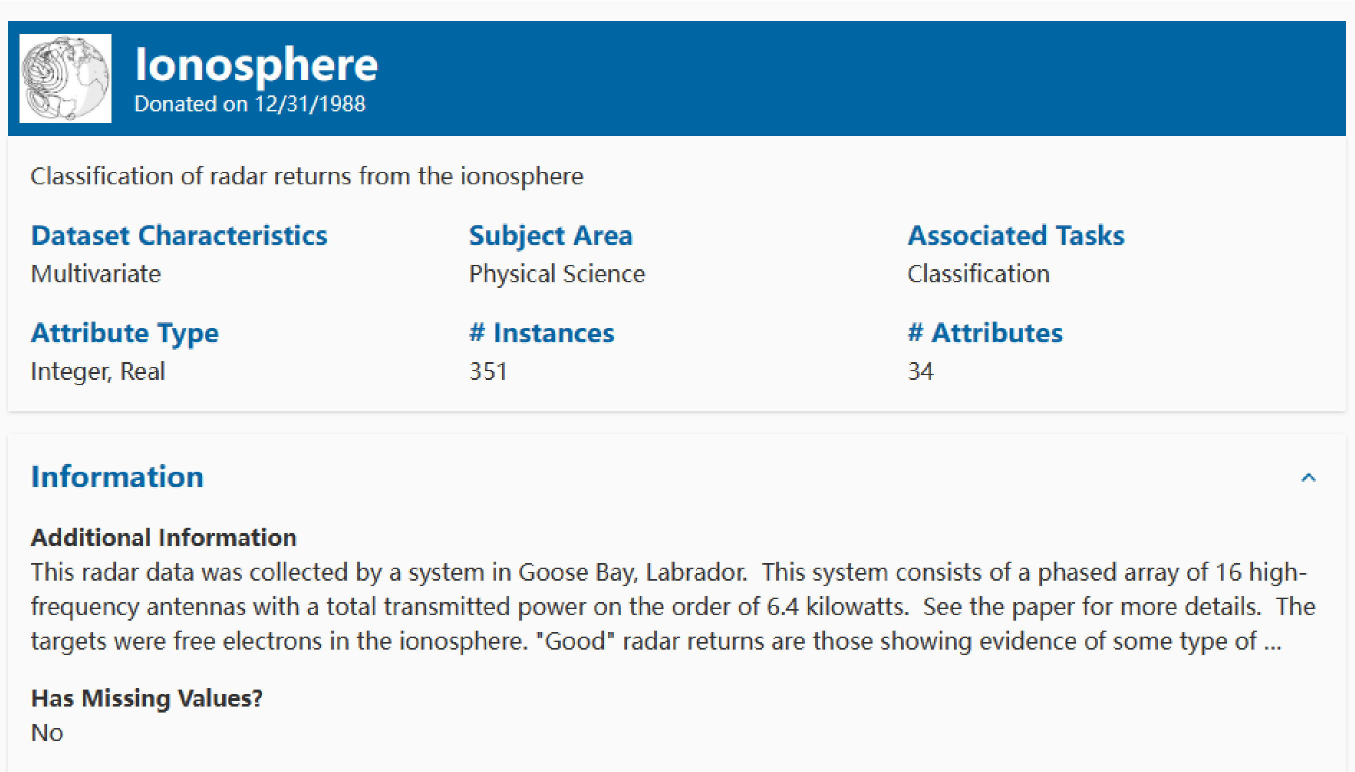
实验一：使用Pytorch构建多层感知机

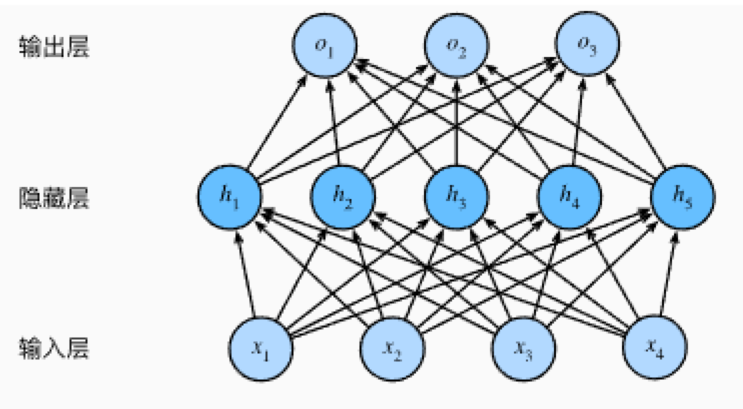
实验内容

本次实验将使用Pytorch构建一个简单的神经网络：多层感知机，并完成一个简单的二分类任务

二分类数据集为ionosphere.csv(电离层数据集)，是UCI机器学习数据集中的经典二分类数据集。它一共有351个观测值，34个自变量，1个因变量（类别），类别取值为g(good)和b(bad)。在ionosphere.csv文件中，共351行，前34列作为自变量（输入的X），最后一列作为类别值（输出的y）。



多层感知器（英语：Multilayer Perceptron，缩写：MLP）是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。MLP可以被看作是一个有向图，由多个的节点层所组成，每一层都全连接到下一层。除了输入节点，每个节点都是一个带有非线性激活函数的神经元。下图展示了一个多层感知机的结构



实验要求

1. 下载、预处理ionosphere数据集、

2. 使用Pytorch实现一个多层（包含输入输出层在内>3层）感知机，并完成对MNIST数据集的训练和测试。多层感知机层数，隐藏层维度，batch size和epoch等可以按需设置
3. 在测试集上的准确率>90%

实验考察能力

1. 数据采集和预处理
2. 特征提取、选择/学习
3. 神经网络构建和模型评估，学习深度学习库Pytorch的基本使用

实验指导

数据集处理

可以使用`sklearn`和`pandas`库来导入和处理数据集

加载数据的具体代码如下：

```
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path, header=None)
        # store the inputs and outputs
        self.X = df.values[:, :-1]
        self.y = df.values[:, -1]
        # ensure input data is floats
        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
        self.y = LabelEncoder().fit_transform(self.y)
        self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.3):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])

def prepare_data(path):
    # load the dataset
```

```
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=64, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl
```

其中，batch_size大小可以按需更改

多层感知机构建

可以通过继承`troch.nn.Module`并重写`__init__`和`forward`函数的方式创建一个多层感知机。在这里，我们使用Pytorch自带的`nn.Linear`完成每个线形层的实现，并使用`torch.nn.functional.relu`和`torch.nn.functional.Sigmoid`分别作为输入层隐藏层和输出层的激活函数。具体代码如下所示

```
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        # second hidden layer
        self.hidden2 = Linear(10, 8)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # third hidden layer and output
        self.hidden3 = Linear(8, 1)
        xavier_uniform_(self.hidden3.weight)
        self.act3 = Sigmoid()

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
        # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # third hidden layer and output
        X = self.hidden3(X)
        X = self.act3(X)
        return X
```

此处，隐藏层的输入维度和输出维度可以更改，但需要保证输入维度与上一层的输出维度匹配，第一层的维度与数据的输入维度匹配。

损失函数，优化器和准确率计算与模型训练

损失函数使用BCE函数，使用numpy计算准确率，优化器使用SGD。损失函数和优化器可以按需修改，优化器中的具体超参数也可按需修改。 训练网络的步骤分为以下几步：

1. 载入模型
2. 初始化，清空网络内上一次训练得到的梯度
3. 载入数据，送入网络进行前向传播
4. 计算损失函数，并进行反向传播计算梯度
5. 调用优化器进行优化

具体代码如下

```
# loss func and optim
optimizer = torch.optim.SGD(model.parameters(),lr=0.01,momentum=0.9)
lossfunc = torch.nn.NLLLoss().cuda()

# accuarcy
def train_model(train_dl, model):
    # define the optimization
    criterion = BCELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    for epoch in range(100):
        # enumerate mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # credit assignment
            loss.backward()
            print("epoch: {}, batch: {}, loss: {}".format(epoch, i, loss.data))
            # update model weights
            optimizer.step()
```

测试网络

使用使用测试集训练网络，直接计算结果并将计算准确率即可

```
def evaluate_model(test_dl, model):
    predictions, actuals = [], []
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
```

```
            yhat = yhat.detach().numpy()
            actual = targets.numpy()
            actual = actual.reshape((len(actual), 1))
            # round to class values
            yhat = yhat.round()
            # store
            predictions.append(yhat)
            actuals.append(actual)
        predictions, actuals = vstack(predictions), vstack(actuals)
        # calculate accuracy
        acc = accuracy_score(actuals, predictions)
        return acc


    # make a class prediction for one row of data
    def predict(row, model):
        # convert row to data
        row = Tensor([row])
        # make prediction
        yhat = model(row)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        return yhat
```

训练与测试脚本

最终，通过调用上述模块可以组装出完整的训练与测试脚本，具体如下：

```
from numpy import vstack
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from torch import Tensor
from torch.optim import SGD
from torch.utils.data import Dataset, DataLoader, random_split
from torch.nn import Linear, ReLU, Sigmoid, Module, BCELoss
from torch.nn.init import kaiming_uniform_, xavier_uniform_


# dataset definition
class CSVDataset(Dataset):
    # load the dataset
    def __init__(self, path):
        # load the csv file as a dataframe
        df = read_csv(path, header=None)
        # store the inputs and outputs
        self.X = df.values[:, :-1]
        self.y = df.values[:, -1]
        # ensure input data is floats
        self.X = self.X.astype('float32')
        # label encode target and ensure the values are floats
```

```
        self.y = LabelEncoder().fit_transform(self.y)
        self.y = self.y.astype('float32')
        self.y = self.y.reshape((len(self.y), 1))

    # number of rows in the dataset
    def __len__(self):
        return len(self.X)

    # get a row at an index
    def __getitem__(self, idx):
        return [self.X[idx], self.y[idx]]

    # get indexes for train and test rows
    def get_splits(self, n_test=0.3):
        # determine sizes
        test_size = round(n_test * len(self.X))
        train_size = len(self.X) - test_size
        # calculate the split
        return random_split(self, [train_size, test_size])


# model definition
class MLP(Module):
    # define model elements
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # input to first hidden layer
        self.hidden1 = Linear(n_inputs, 10)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu')
        self.act1 = ReLU()
        # second hidden layer
        self.hidden2 = Linear(10, 8)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # third hidden layer and output
        self.hidden3 = Linear(8, 1)
        xavier_uniform_(self.hidden3.weight)
        self.act3 = Sigmoid()

    # forward propagate input
    def forward(self, X):
        # input to first hidden layer
        X = self.hidden1(X)
        X = self.act1(X)
        # second hidden layer
        X = self.hidden2(X)
        X = self.act2(X)
        # third hidden layer and output
        X = self.hidden3(X)
        X = self.act3(X)
        return X


# prepare the dataset
```

```python
def prepare_data(path):
    # load the dataset
    dataset = CSVDataset(path)
    # calculate split
    train, test = dataset.get_splits()
    # prepare data loaders
    train_dl = DataLoader(train, batch_size=32, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl


# train the model
def train_model(train_dl, model):
    # define the optimization
    criterion = BCELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # enumerate epochs
    for epoch in range(100):
        # enumerate mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            # clear the gradients
            optimizer.zero_grad()
            # compute the model output
            yhat = model(inputs)
            # calculate loss
            loss = criterion(yhat, targets)
            # credit assignment
            loss.backward()
            print("epoch: {}, batch: {}, loss: {}".format(epoch, i, loss.data))
            # update model weights
            optimizer.step()


# evaluate the model
def evaluate_model(test_dl, model):
    predictions, actuals = [], []
    for i, (inputs, targets) in enumerate(test_dl):
        # evaluate the model on the test set
        yhat = model(inputs)
        # retrieve numpy array
        yhat = yhat.detach().numpy()
        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # round to class values
        yhat = yhat.round()
        # store
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # calculate accuracy
    acc = accuracy_score(actuals, predictions)
    return acc
```

```python
# make a class prediction for one row of data
def predict(row, model):
    # convert row to data
    row = Tensor([row])
    # make prediction
    yhat = model(row)
    # retrieve numpy array
    yhat = yhat.detach().numpy()
    return yhat


# prepare the data
path = './data/ionosphere.csv'
train_dl, test_dl = prepare_data(path)
print(len(train_dl.dataset), len(test_dl.dataset))
# define the network
model = MLP(34)
print(model)
# train the model
train_model(train_dl, model)
# evaluate the model
acc = evaluate_model(test_dl, model)
print('Accuracy: %.3f' % acc)
# make a single prediction (expect class=1)
row = [1, 0, 0.99539, -0.05889, 0.85243, 0.02306, 0.83398, -0.37708, 1, 0.03760,
0.85243, -0.17755, 0.59755, -0.44945,
       0.60536, -0.38223, 0.84356, -0.38542, 0.58212, -0.32192, 0.56971, -0.29674,
0.36946, -0.47357, 0.56811, -0.51171,
       0.41078, -0.46168, 0.21266, -0.34090, 0.42267, -0.54487, 0.18641, -0.45300]
yhat = predict(row, model)
print('Predicted: %.3f (class=%d)' % (yhat, yhat.round()))
```

实验二：使用卷积神经网络进行手写数字识别

实验内容

卷积神经网络（英语：Convolutional Neural Network，缩写：CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。

卷积神经网络由一个或多个卷积层和顶端的全连通层（对应经典的神经网络）组成，同时也包括关联权重和池化层（pooling layer）。这一结构使得卷积神经网络能够利用输入数据的二维结构。与其他深度学习结构相比，卷积神经网络在图像和语音识别方面能够给出更好的结果。