

《人工智能软件开发与实践》

(2023 学年 秋季 学期)

作 业 报 告

学 号： _
姓 名： ____
班 级： ____
任课教师： _____

作业报告

实验名称： 使用 PYTORCH 构建多层感知机

成绩：

实验类别： 验证/综合型实验 实验要求： 1 人 1 组 时间： 2023 年 9 月 5 日

一、 实验目的


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、实验内容

1. 下载、预处理 ionosphere 数据集

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）。

<https://archive.ics.uci.edu/dataset/52/ionosphere>



Ionosphere
Donated on 12/31/1988

Classification of radar returns from the ionosphere

Dataset Characteristics Multivariate	Subject Area Physical Science	Associated Tasks Classification
Attribute Type Integer, Real	# Instances 351	# Attributes 34

Information

Additional Information
This radar data was collected by a system in Goose Bay, Labrador. This system consists of a phased array of 16 high-frequency antennas with a total transmitted power on the order of 6.4 kilowatts. See the paper for more details. The targets were free electrons in the ionosphere. "Good" radar returns are those showing evidence of some type of ...
[SHOW MORE](#)

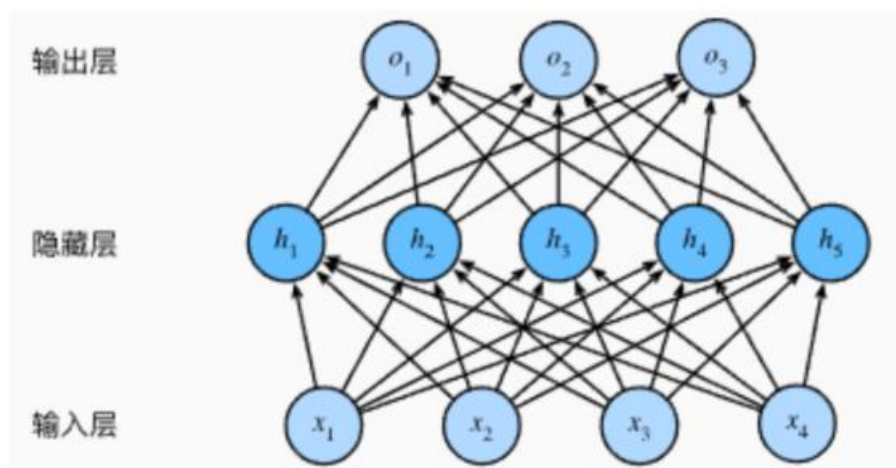
Has Missing Values?
No

Attribute Information

Additional Information
-- All 34 are continuous
-- The 35th attribute is either "good" or "bad" according to the definition summarized above. This is a binary classification task.

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。

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。



3. 在测试集上的准确率

4. 为程序按照功能块添加注释（将带有注释的程序，粘贴至此处）

```
from numpy import vstack
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
import torch
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_

batchNumber = 64
epochNumber = 100
hiddenDim = 30

class CSVDataset(Dataset):
    # 加载数据集
    def __init__(self, path):
```

```

# 将 csv 文件读取为 dataframe
df = read_csv(path, header=None)
# 存储输入与输出，每行末尾列为因变量，其余为自变量
self.X = df.values[:, :-1]
self.y = df.values[:, -1]
# 输入为浮点型
self.X = self.X.astype('float32')
# 标记编码，确保因变量为浮点数
self.y = LabelEncoder().fit_transform(self.y)
self.y = self.y.astype('float32')
self.y = self.y.reshape((len(self.y), 1))

# 获取数据集行数
def __len__(self):
    return len(self.X)

# 通过索引获取行
def __getitem__(self, idx):
    return [self.X[idx], self.y[idx]]

# 获取训练、测试集行的索引
def get_splits(self, n_test=0.3):
    # 根据比例确定训练集和测试集大小
    test_size = round(n_test * len(self.X))
    train_size = len(self.X) - test_size
    # 计算分裂
    return random_split(self, [train_size, test_size])

def prepare_data(path):
    # 加载数据集
    dataset = CSVDataset(path)
    # 划分训练集和测试集
    train, test = dataset.get_splits()
    # 准备数据加载器
    train_dl = DataLoader(train, batch_size=batchNumber, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl

# class MLP(Module):    # 隐藏层数 1
#     # 定义模型元素
#     def __init__(self, n_inputs):
#         super(MLP, self).__init__()
#         # 输入层→隐藏层
#         self.hidden = Linear(n_inputs, hiddenDim)

```

```

#         kaiming_uniform_(self.hidden.weight, nonlinearity='relu') # 初始化
#         self.act = ReLU()
#         # 隐藏层→输出层
#         self.output = Linear(hiddenDim, 1)
#         xavier_uniform_(self.output.weight) # 初始化
#         self.act2 = Sigmoid()
#
#     # 前向传递
#     def forward(self, X):
#         # 输入层→隐藏层
#         X = self.hidden(X)
#         X = self.act(X)
#         # 隐藏层→输出层
#         X = self.output(X)
#         X = self.act2(X)
#         return X

```

```

class MLP(Module): # 隐藏层数 2
    # 定义模型元素
    def __init__(self, n_inputs):
        super(MLP, self).__init__()
        # 输入层→隐藏层 1
        self.hidden1 = Linear(n_inputs, hiddenDim)
        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu') # 初始化
        self.act = ReLU()
        # 隐藏层 1→隐藏层 2
        self.hidden2 = Linear(hiddenDim, hiddenDim)
        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')
        self.act2 = ReLU()
        # 隐藏层 2→输出层
        self.output = Linear(hiddenDim, 1)
        xavier_uniform_(self.output.weight) # 初始化
        self.act3 = Sigmoid()

    # 前向传递
    def forward(self, X):
        # 输入层→隐藏层
        X = self.hidden1(X)
        X = self.act(X)
        # 隐藏层 1→隐藏层 2
        X = self.hidden2(X)
        X = self.act2(X)
        # 隐藏层→输出层
        X = self.output(X)
        X = self.act3(X)
        return X

```

```

def train_model(train_dl, model):
    # 定义优化
    criterion = BCELoss()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    # 枚举 epochs
    for epoch in range(epochNumber):
        # 枚举 mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            # 清除梯度
            optimizer.zero_grad()
            # 计算模型输出
            yhat = model(inputs)
            # 计算 loss
            loss = criterion(yhat, targets)
            # credit assignment
            loss.backward()
            print("epoch: {}, batch: {}, loss: {}".format(epoch, i, loss.data))
            # 更新模型权重
            optimizer.step()

def evaluate_model(test_dl, model):
    predictions, actuals = [], []
    for i, (inputs, targets) in enumerate(test_dl):
        # 用测试集评估模型
        yhat = model(inputs)
        # 检索 numpy 数组
        yhat = yhat.detach().numpy()

        actual = targets.numpy()
        actual = actual.reshape((len(actual), 1))
        # 四舍五入到类值
        yhat = yhat.round()
        # 存储
        predictions.append(yhat)
        actuals.append(actual)
    predictions, actuals = vstack(predictions), vstack(actuals)
    # 计算精度
    acc = accuracy_score(actuals, predictions)
    return acc

def predict(row, model):
    # 将行转换为数据
    row = Tensor([row])

```

```

# 做出预测
yhat = model(row)
# 检索 numpy 数组
yhat = yhat.detach().numpy()
return yhat

# 定义文件路径, 获取训练集, 测试集
path = './ionosphere/ionosphere.data'
train_dl, test_dl = prepare_data(path)
print(len(train_dl.dataset), len(test_dl.dataset))

# 定义模型
model = MLP(34)
print(model)

# 定义优化器, loss 函数并用训练集训练模型
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
lossfunc = torch.nn.NLLLoss().cuda()
train_model(train_dl, model)

# 评估模型, 输出精度
acc = evaluate_model(train_dl, model)
print('Close Accuracy: %.3f' % acc)
acc = evaluate_model(test_dl, model)
print('Open Accuracy: %.3f' % acc)

# 用模型进行预测
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()))

```

三、使用的算法名称（若无，可以不填）

四、程序源码（拷贝至此处，同时作为附件和报告再一份单独的程序）

a) 装载数据

```

from numpy import vstack

from pandas import read_csv

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import accuracy_score

```

```

import torch

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_


batchNumber = 64

epochNumber = 100

hiddenDim = 30


class CSVDataset(Dataset):
    # 加载数据集

    def __init__(self, path):
        # 将 csv 文件读取为 dataframe

        df = read_csv(path, header=None)

        # 存储输入与输出, 每行末尾列为因变量, 其余为自变量

        self.X = df.values[:, :-1]

        self.y = df.values[:, -1]

        # 输入为浮点型

        self.X = self.X.astype('float32')

        # 标记编码, 确保因变量为浮点数

        self.y = LabelEncoder().fit_transform(self.y)

        self.y = self.y.astype('float32')

        self.y = self.y.reshape((len(self.y), 1))


    # 获取数据集行数

    def __len__(self):
        return len(self.X)

```



```

# 通过索引获取行
def __getitem__(self, idx):
    return [self.X[idx], self.y[idx]]

# 获取训练、测试集行的索引
def get_splits(self, n_test=0.3):
    # 根据比例确定训练集和测试集大小
    test_size = round(n_test * len(self.X))
    train_size = len(self.X) - test_size
    # 计算分裂
    return random_split(self, [train_size, test_size])

def prepare_data(path):
    # 加载数据集
    dataset = CSVDataset(path)
    # 划分训练集和测试集
    train, test = dataset.get_splits()
    # 准备数据加载器
    train_dl = DataLoader(train, batch_size=batchNumber, shuffle=True)
    test_dl = DataLoader(test, batch_size=1024, shuffle=False)
    return train_dl, test_dl

```

b) 构建多层感知器

可以通过继承 `torch.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。

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

```
class MLP(Module):    # 隐藏层数 2

    # 定义模型元素

    def __init__(self, n_inputs):

        super(MLP, self).__init__()

        # 输入层→隐藏层 1

        self.hidden1 = Linear(n_inputs, hiddenDim)

        kaiming_uniform_(self.hidden1.weight, nonlinearity='relu') # 初始化

        self.act = ReLU()

        # 隐藏层 1→隐藏层 2

        self.hidden2 = Linear(hiddenDim, hiddenDim)

        kaiming_uniform_(self.hidden2.weight, nonlinearity='relu')

        self.act2 = ReLU()

        # 隐藏层 2→输出层

        self.output = Linear(hiddenDim, 1)

        xavier_uniform_(self.output.weight) # 初始化

        self.act3 = Sigmoid()


    # 前向传递

    def forward(self, X):

        # 输入层→隐藏层

        X = self.hidden1(X)

        X = self.act(X)
```

```

# 隐藏层 1→隐藏层 2

X = self.hidden2(X)

X = self.act2(X)

# 隐藏层→输出层

X = self.output(X)

X = self.act3(X)

return X

```

```

def train_model(train_dl, model):
    # 定义优化
    criterion = BCELoss()

    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)

    # 枚举 epochs
    for epoch in range(epochNumber):
        # 枚举 mini batches
        for i, (inputs, targets) in enumerate(train_dl):
            # 清除梯度
            optimizer.zero_grad()

            # 计算模型输出
            yhat = model(inputs)

            # 计算 loss
            loss = criterion(yhat, targets)

            # credit assignment
            loss.backward()

            print("epoch: {}, batch: {}, loss: {}".format(epoch, i, loss.data))

            # 更新模型权重
            optimizer.step()

def evaluate_model(test_dl, model):
    predictions, actuals = [], []

```

```

for i, (inputs, targets) in enumerate(test_dl):
    # 用测试集评估模型
    yhat = model(inputs)
    # 检索 numpy 数组
    yhat = yhat.detach().numpy()

    actual = targets.numpy()
    actual = actual.reshape((len(actual), 1))
    # 四舍五入到类值
    yhat = yhat.round()
    # 存储
    predictions.append(yhat)
    actuals.append(actual)
predictions, actuals = vstack(predictions), vstack(actuals)
# 计算精度
acc = accuracy_score(actuals, predictions)
return acc

```

c) 使用感知器网络进行预测

```

def predict(row, model):
    # 将行转换为数据
    row = Tensor([row])
    # 做出预测
    yhat = model(row)
    # 检索 numpy 数组
    yhat = yhat.detach().numpy()
    return yhat

```

d) 训练与测试脚本

```

# 定义文件路径, 获取训练集, 测试集
path = './ionosphere/ionosphere.data'
train_dl, test_dl = prepare_data(path)

```

```

print(len(train_dl.dataset), len(test_dl.dataset))

# 定义模型
model = MLP(34)
print(model)

# 定义优化器, loss 函数并用训练集训练模型
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
lossfunc = torch.nn.NLLLoss().cuda()
train_model(train_dl, model)

# 评估模型, 输出精度
acc = evaluate_model(train_dl, model)
print('Close Accuracy: %.3f' % acc)
acc = evaluate_model(test_dl, model)
print('Open Accuracy: %.3f' % acc)

# 用模型进行预测
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()))

```

五、程序运行结果（将程序运行结果的截图拷贝至此处，或者填写实验结果）

对隐层层数、隐层维度、Batch 大小、Epoch 次数等超参进行调参（不需要穷举，只需要填写下面的表格即可，保证每个超参至少取 2 个不同的值），填写对应的封闭测精度和开放测精度。（表格行数不够，可以自行添加）

序号	隐层层数	隐层维度	Batch 大小	Epoch 次数	封闭精度	开放精度
1	1	10	64	100	0.967	0.905
2	1	20	16	200	0.992	0.924
3	1	30	64	100	0.976	0.952
4	2	30	64	100	0.996	0.952

六、心得体会和遇到的困难

对 AI 生成代码没有把握，对机器学习和神经网络等知识储备不够。