实践报告：手写字符识别

学号：2212534

姓名：魏思诚

日期：2024/11/10

南开大学 软件学院

目 录

[1 实验介绍 3](#_Toc7jmh2f)

[1)实验环境 3](#_Toclypsj7)

[2)实验内容 3](#_Toczzwyqs)

[2 实验目的 4](#_Tocts8p9h)

[1)了解数据预处理流程1 4](#_Tock523rk)

[2)了解训练流程n 4](#_Tocpmdw9e)

[3 实验流程 5](#_Toc5ihwrt)

[1)项目设置 5](#_Toc8tbxzn)

[2)代码编写 5](#_Toc5489qp)

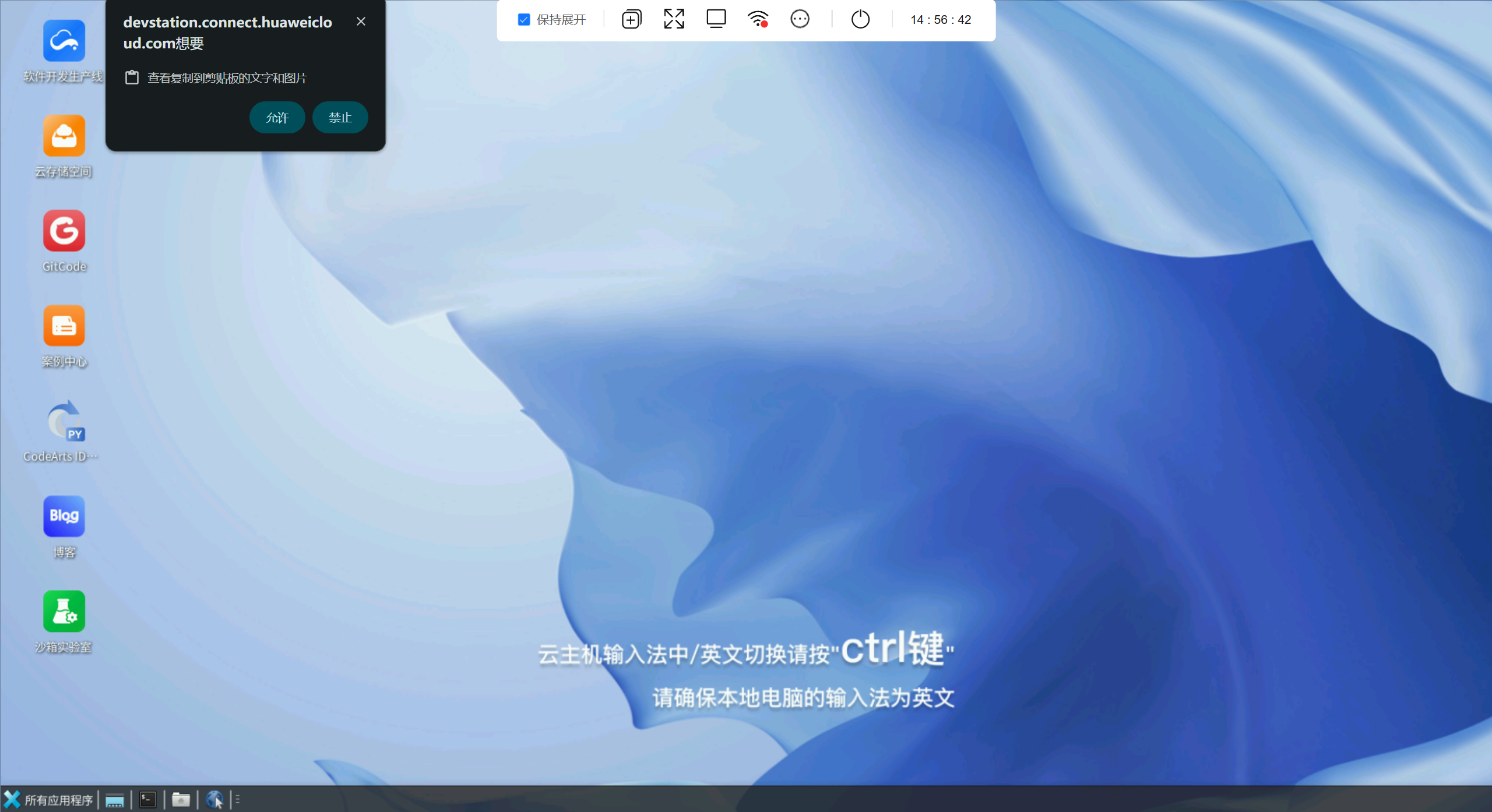
[4 实验结果或结论 6](#_Toc2fe5jy)

[1)实验结果1 6](#_Tocmqn8po)

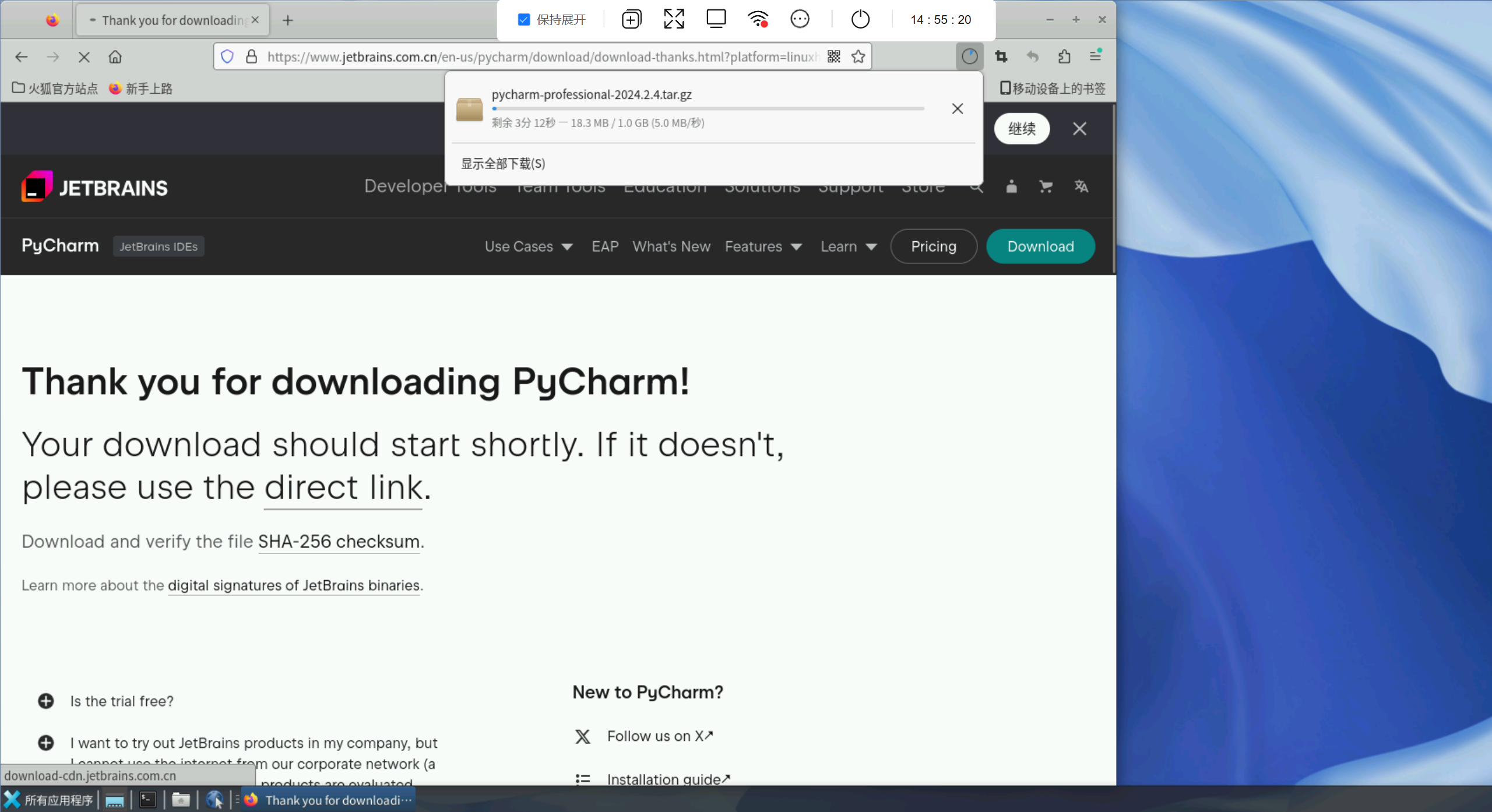
[2)实验结论n 6](#_Tocaflm23)

# 实验介绍

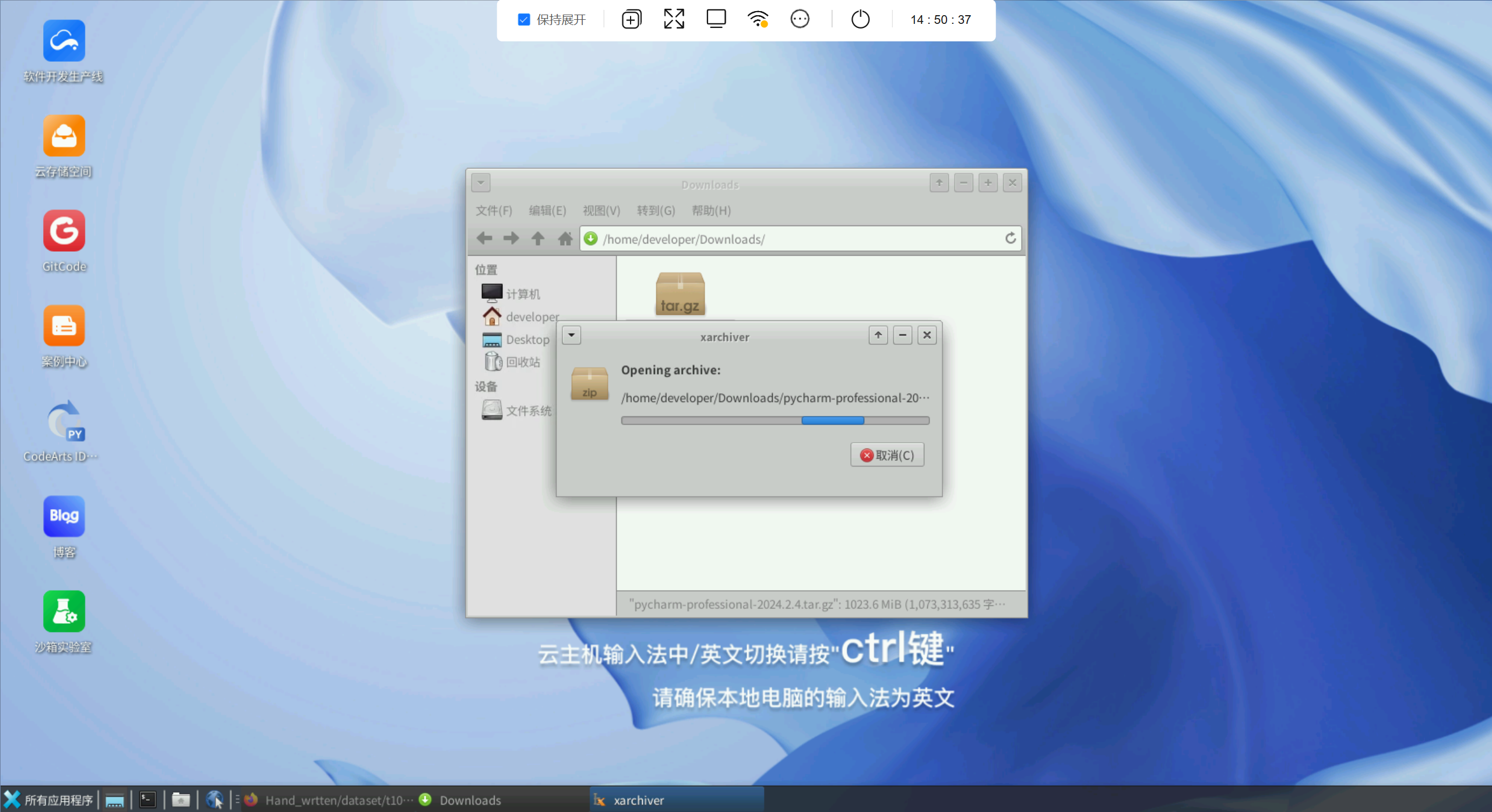
## 实验环境

使用华为云主机搭建环境,

进入网页https://www.jetbrains.com.cn/en-us/pycharm/download/download-thanks.html?platform=linuxhttps://home.firefoxchina.cn

下载编译器jetbrains

解压压缩包:



使用pycharm作为解释器,数据集使用

<https://github.com/Hurri-cane/Hand_wrtten/blob/master/dataset/t10k-images.idx3-ubyte>

<https://github.com/Hurri-cane/Hand_wrtten/blob/master/dataset/t10k-labels.idx1-ubyte>

<https://github.com/Hurri-cane/Hand_wrtten/blob/master/dataset/train-images.idx3-ubyte>

<https://github.com/Hurri-cane/Hand_wrtten/blob/master/dataset/train-labels.idx1-ubyte>

## 实验内容

实验内容主要包括使用四个从GitHub获取的手写字符数据集，应用机器学习技术进行训练，搭建一个能够有效识别手写字符的模型。首先，将下载的数据集进行预处理，包括归一化、去噪以及数据增强操作，以确保数据在输入模型前具有良好的质量和一致性。通过数据增强，可以丰富训练集中的样本种类，提升模型在不同书写风格和字符变形下的识别能力。

在模型的选择和设计上，实验尝试了几种常见的机器学习算法和深度学习模型，包括支持向量机（SVM）、k近邻算法（KNN）、卷积神经网络（CNN）等。对比不同模型的训练效果，优化模型架构，最终选取性能最优的模型用于手写字符的分类任务。训练过程中采用分epoch训练的方式，逐步调整模型的超参数，如学习率、批量大小等，以达到最佳的训练效果。

为了更深入地分析模型的表现，还将各模型的结果在测试集上进行验证，通过准确率、精度、召回率等指标评估模型的性能。此外，实验记录了训练时间和资源消耗，以分析不同模型在效率和精度上的优缺点。最终，选取表现最佳的模型进行进一步优化，以确保其能够在实际应用中有效识别各种手写字符。

# 实验目的

## 学习数据预处理流程

学习将数据划分成训练测试集以及验证集,

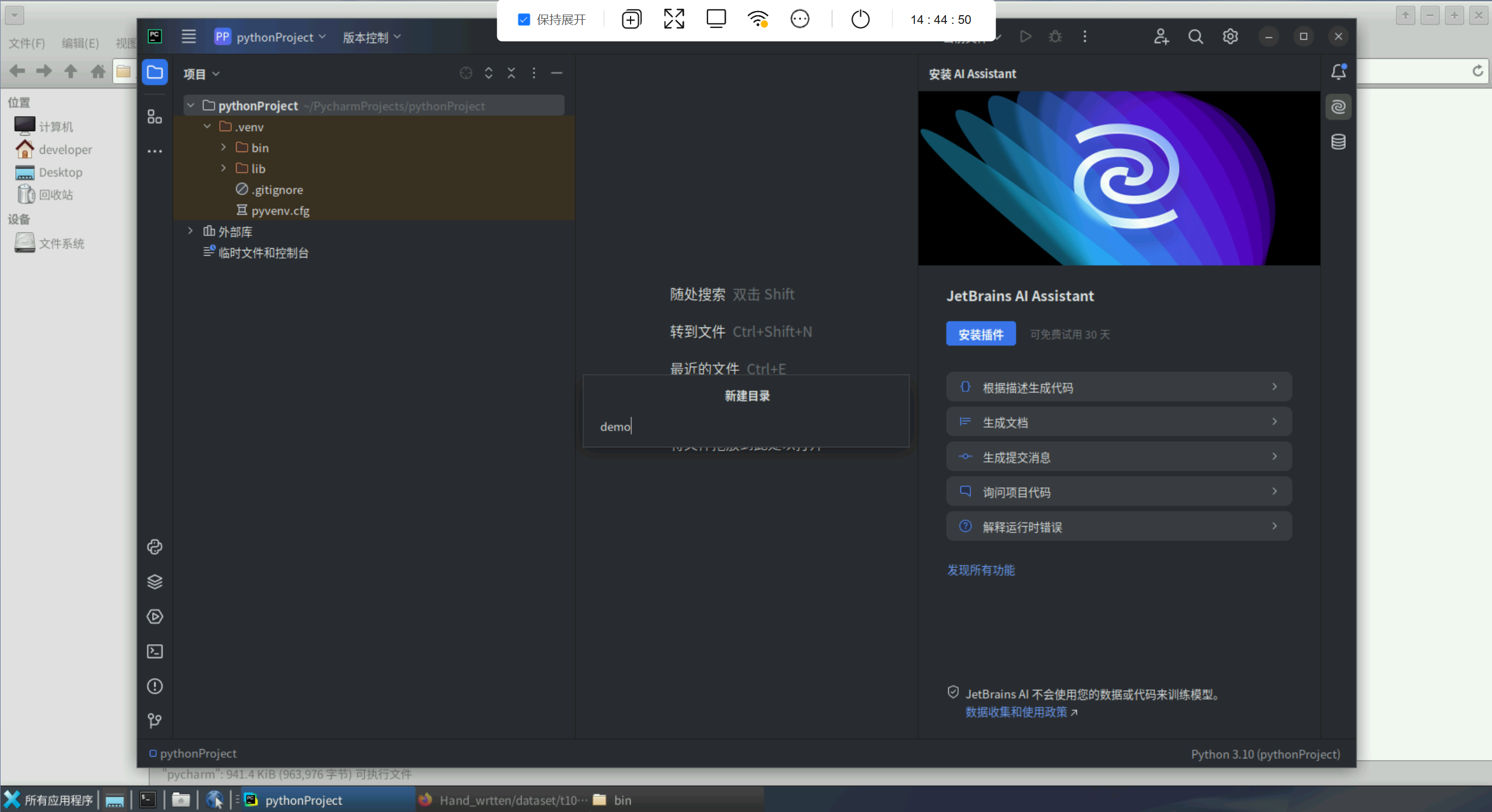
## 训练流程

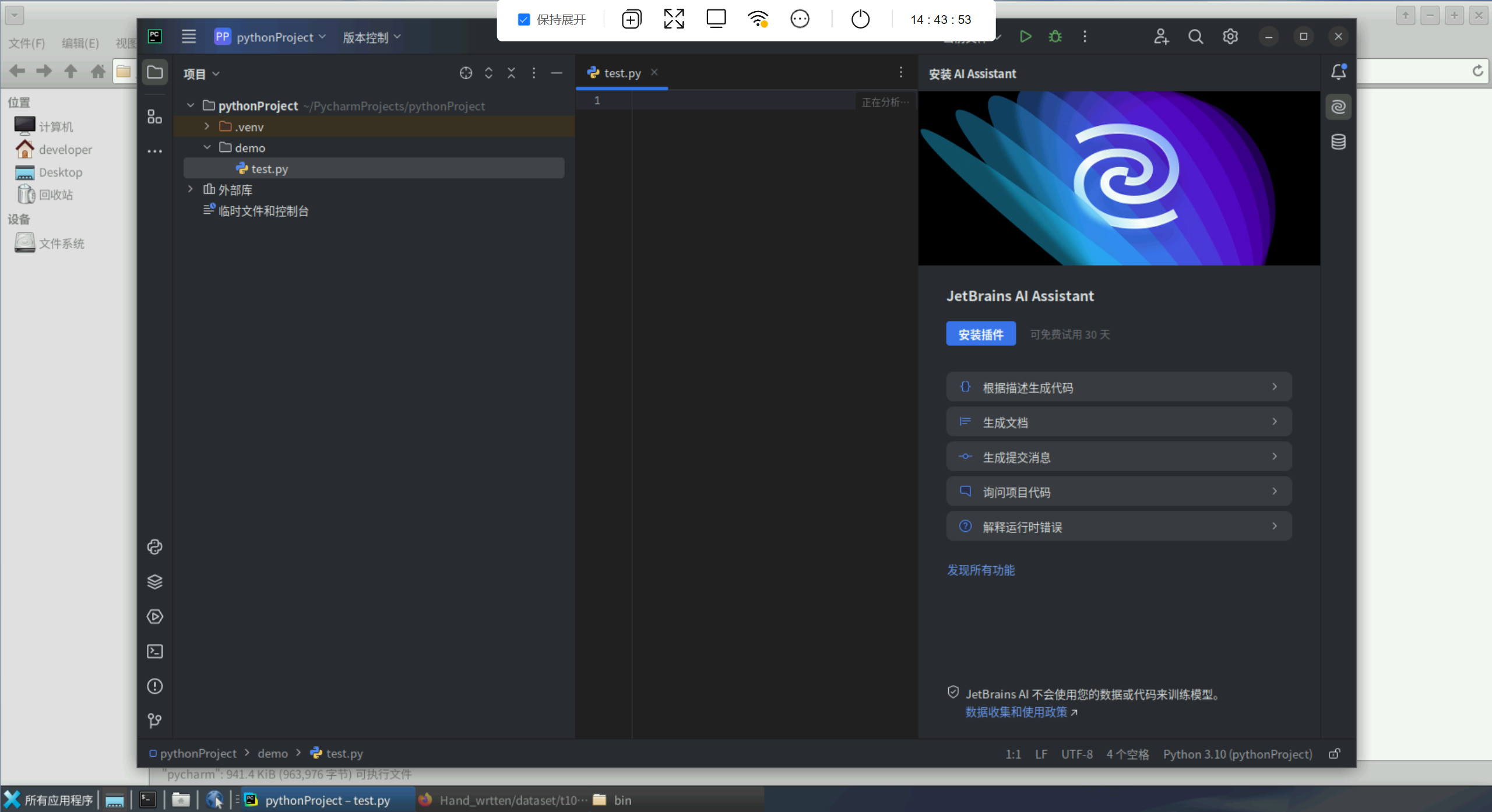
在本实验中，旨在深入学习深度学习模型的训练方法与模型结构的搭建。通过分epoch训练，观察模型在不同训练轮次中的表现，逐步调整模型参数以提升精度与收敛速度。此外，实验还探讨了ReLU激活函数的作用，通过引入非线性变换，使模型能够学习更复杂的特征，并在训练过程中避免梯度消失问题，从而加速收敛。

验还设计了Linear层（全连接层）和最大池化层（MaxPool）的结构，以实现特征提取和降维。Linear层通过全连接的方式将特征映射至分类标签空间，帮助完成最终的分类任务；而MaxPool层通过选择局部区域内的最大值来缩小数据尺寸，减少计算量，同时增强模型对局部特征的鲁棒性。实验的整体目标在于理解这些关键层的作用，并在手写字符识别任务中验证它们的效果。

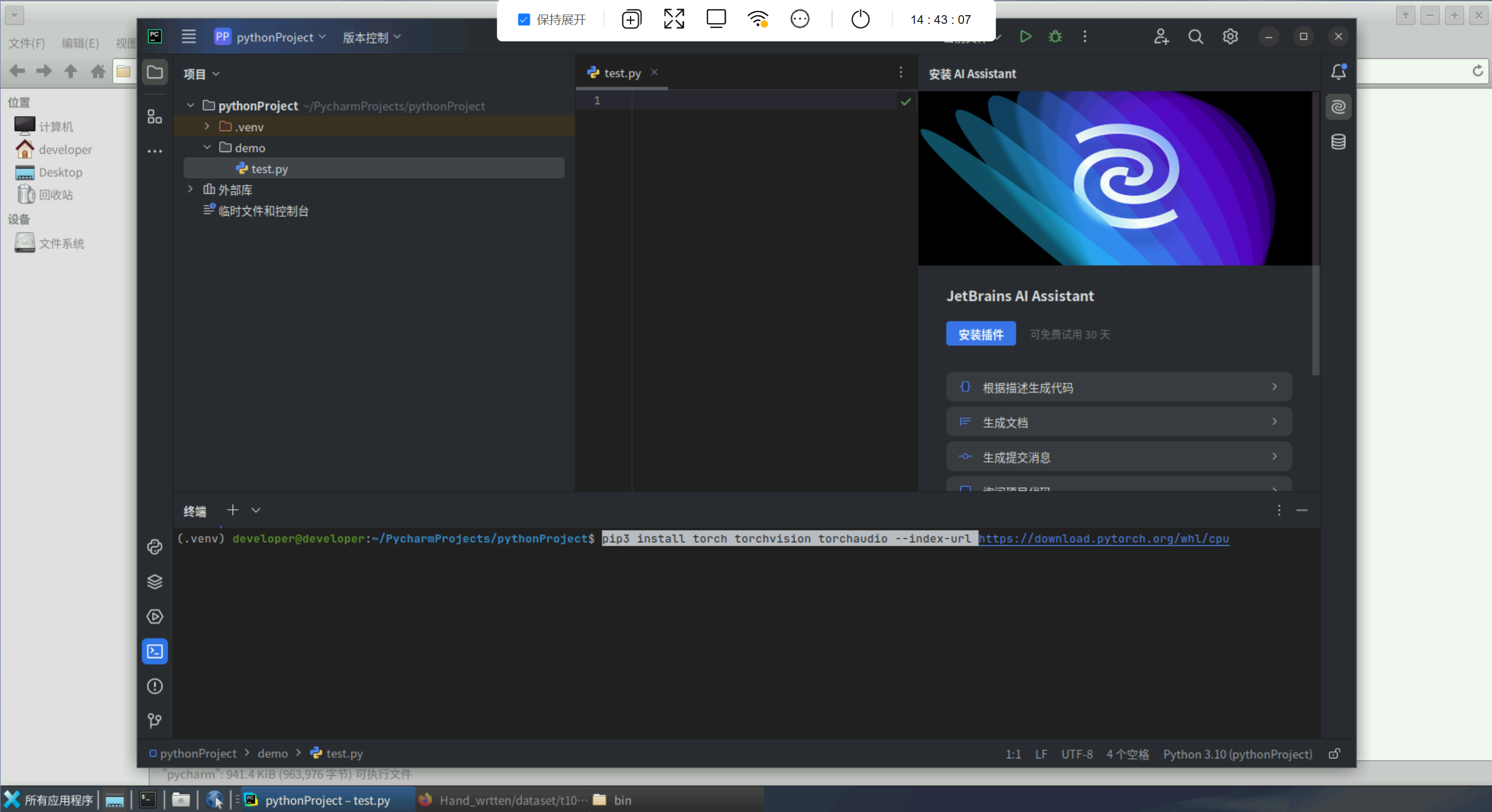
# 实验流程

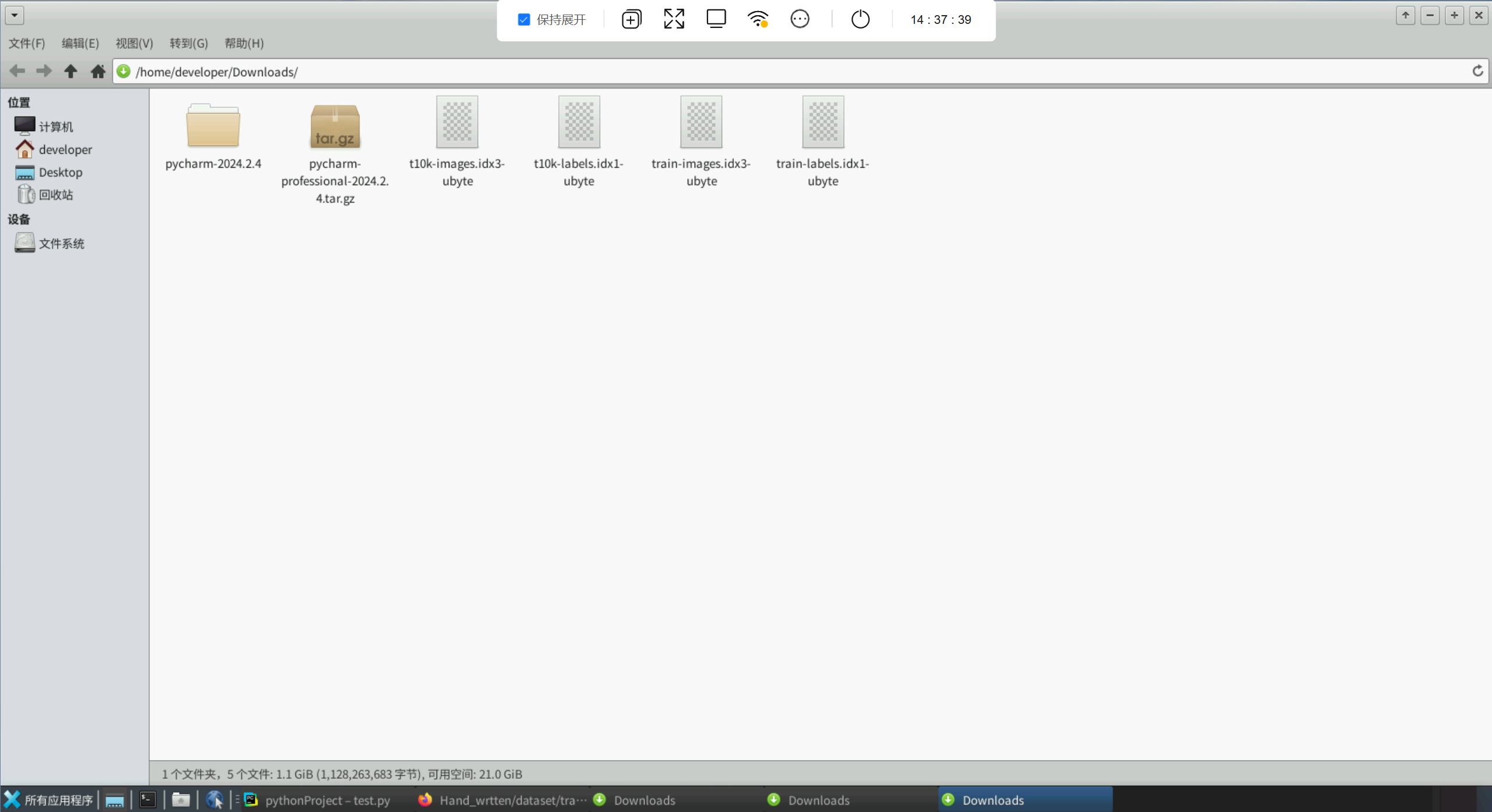
## 项目设置



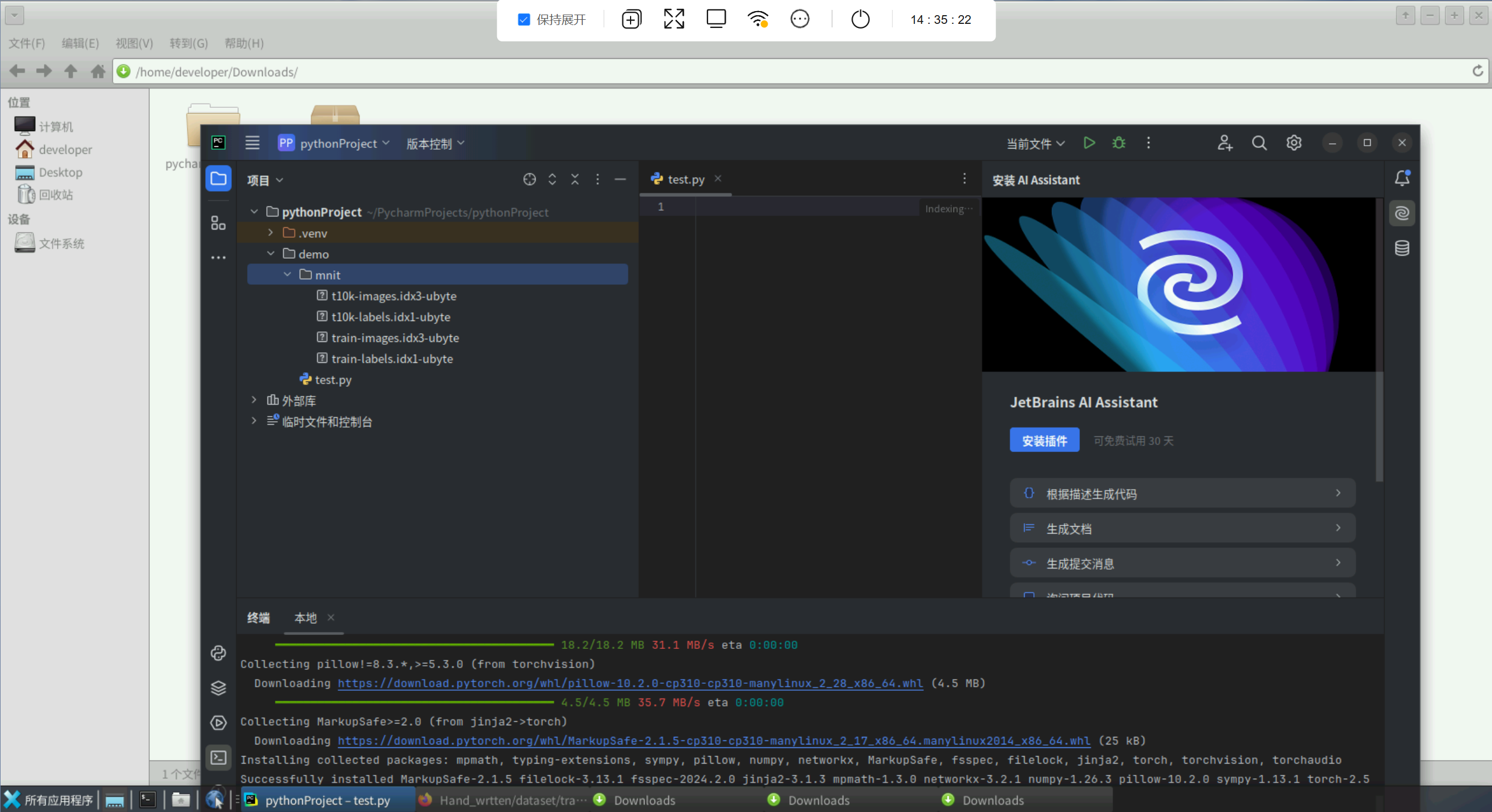


部署python框架

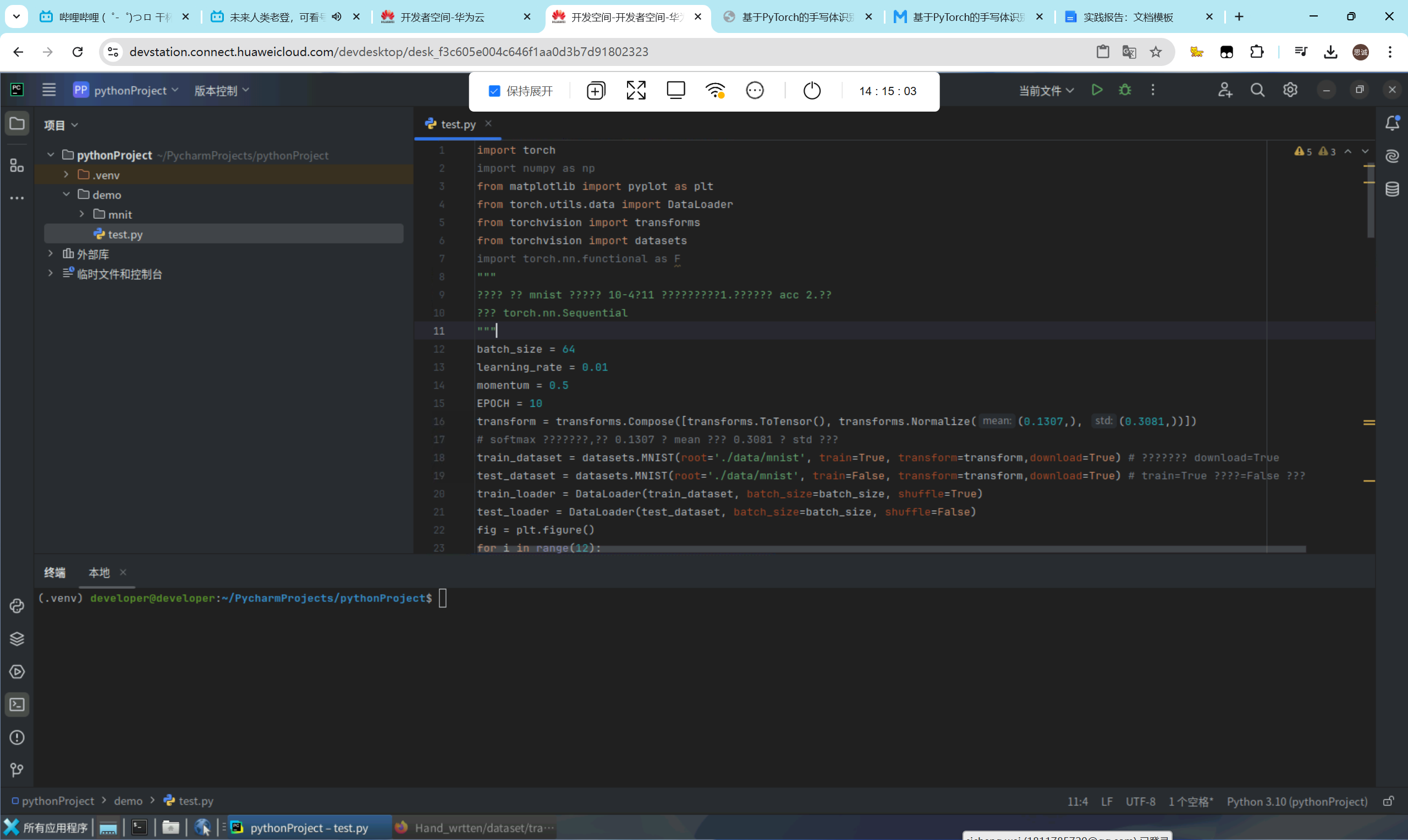


下载数据集:  


项目最终格式:



## 编写代码



代码具体附在文档最后

# 实验结果或结论

## 实验结果



实验总体表现良好,正确率逮到98.2%

## 实验结论

在这个项目中，我深入学习了分epoch训练的概念，理解到将训练数据分成若干个小批量，通过多次遍历数据集来更新模型参数的过程有助于提升模型的准确性和收敛速度。每一个epoch相当于在所有数据上训练一轮，分epoch训练可以更好地观察模型在每轮训练后的表现，便于调试和调参。

在模型搭建的过程中，我学习了如何合理地设计神经网络结构，包括隐藏层的数量、每层神经元的数量等，从而在保证模型具备足够表达能力的同时，避免过度复杂化。特别是ReLU（Rectified Linear Unit）激活函数的作用，它在每个神经元的输出上施加非线性变换，使得网络可以学习更复杂的特征，并加速了训练过程，因为ReLU函数的导数简单、计算效率高。相比于传统的sigmoid或tanh函数，ReLU更容易避免梯度消失问题，因此成为了深度学习中的主流选择。

此外，我还学习了Linear层（全连接层）的功能，它是将上一层的所有神经元与下一层的每一个神经元进行连接。这种连接方式使得模型可以在最后提取出高度抽象的特征并完成分类。Linear层通常放在模型的最后几层，将网络学到的复杂特征映射到分类标签空间，从而进行预测。

最大池化层（MaxPool）也是项目中重要的一部分，通过取池化窗口内的最大值来缩小数据的尺寸，减少计算量，同时保留特征的显著性。MaxPool层可以帮助模型在不同的图像位置上保持特征的一致性，增强模型的抗干扰能力，适应图像的局部变化

此外，在模型调参和测试中，我体会到选择合适的超参数对提高模型精度至关重要，同时也积累了如何应对过拟合、欠拟合等常见问题的经验。这次项目不仅提升了我的Python编程能力，还强化了我对机器学习项目的整体把控力，包括数据处理、模型设计与评估、结果优化等多个环节。

代码:

import torch

import numpy as np

from matplotlib import pyplot as plt

from torch.utils.data import DataLoader

from torchvision import transforms

from torchvision import datasets

import torch.nn.functional as F

"""

卷积运算 使用 mnist 数据集，和 10-4，11 类似的，只是这里：1.输出训练轮的 acc 2.模型

上使用 torch.nn.Sequential

"""

batch\_size = 64

learning\_rate = 0.01

momentum = 0.5

EPOCH = 10

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])

# softmax 归一化指数函数,其中 0.1307 是 mean 均值和 0.3081 是 std 标准差

train\_dataset = datasets.MNIST(root='./data/mnist', train=True, transform=transform,download=True) # 本地没有就加上 download=True

test\_dataset = datasets.MNIST(root='./data/mnist', train=False, transform=transform,download=True) # train=True 训练集，=False 测试集

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)

fig = plt.figure()

for i in range(12):

 plt.subplot(3, 4, i+1)

 plt.tight\_layout()

 plt.imshow(train\_dataset.train\_data[i], cmap='gray', interpolation='none')

 plt.title("Labels: {}".format(train\_dataset.train\_labels[i]))

 plt.xticks([])

 plt.yticks([])

plt.show()

# 训练集乱序，测试集有序

class Net(torch.nn.Module):

 def \_\_init\_\_(self):

    super(Net, self).\_\_init\_\_()

    self.conv1 = torch.nn.Sequential(

    torch.nn.Conv2d(1, 10, kernel\_size=5),

    torch.nn.ReLU(),

    torch.nn.MaxPool2d(kernel\_size=2),

    )

    self.conv2 = torch.nn.Sequential(

    torch.nn.Conv2d(10, 20, kernel\_size=5),

    torch.nn.ReLU(),

    torch.nn.MaxPool2d(kernel\_size=2),

    )

    self.fc = torch.nn.Sequential(

    torch.nn.Linear(320, 50),

    torch.nn.Linear(50, 10),

    )

 def forward(self, x):

    batch\_size = x.size(0)

    x = self.conv1(x) # 一层卷积层,一层池化层,一层激活层(图是先卷积后激活再池

    x = self.conv2(x) # 再来一次

    x = x.view(batch\_size, -1) # flatten 变成全连接网络需要的输入 (batch, 20,4,4)

    x = self.fc(x)

    return x # 最后输出的是维度为 10 的，也就是（对应数学符号的 0~9）

model = Net()

# Construct loss and optimizer ----------------------------------------------------------------------------

criterion = torch.nn.CrossEntropyLoss() # 交叉熵损失

optimizer = torch.optim.SGD(model.parameters(), lr=learning\_rate, momentum=momentum) # lr 学习率，momentum 冲量

# Train and Test CLASS ------------------------------------------------------------------------------------

# 把单独的一轮一环封装在函数类里

def train(epoch):

 running\_loss = 0.0 # 这整个 epoch 的 loss 清零

 running\_total = 0

 running\_correct = 0

 for batch\_idx, data in enumerate(train\_loader, 0):

    inputs, target = data

    optimizer.zero\_grad()

    # forward + backward + update

    outputs = model(inputs)

    loss = criterion(outputs, target)

    loss.backward()

    optimizer.step()

    # 把运行中的 loss 累加起来，为了下面 300 次一除

    running\_loss += loss.item()

    # 把运行中的准确率 acc 算出来

    \_, predicted = torch.max(outputs.data, dim=1)

    running\_total += inputs.shape[0]

    running\_correct += (predicted == target).sum().item()

    if batch\_idx % 300 == 299: # 不想要每一次都出 loss，浪费时间，选择每 300

        print('[%d, %5d]: loss: %.3f , acc: %.2f %%'% (epoch + 1, batch\_idx + 1, running\_loss / 300, 100 \* running\_correct / running\_total))

        running\_loss = 0.0 # 这小批 300 的 loss 清零

        running\_total = 0

        running\_correct = 0 # 这小批 300 的 acc 清零

        # torch.save(model.state\_dict(), './model\_Mnist.pth')

        # torch.save(optimizer.state\_dict(), './optimizer\_Mnist.pth')

def test():

 correct = 0

 total = 0

 with torch.no\_grad(): # 测试集不用算梯度

    for data in test\_loader:

        images, labels = data

        outputs = model(images)

        \_, predicted = torch.max(outputs.data, dim=1) # dim = 1 列是第 0 个维

        total += labels.size(0) # 张量之间的比较运算

        correct += (predicted == labels).sum().item()

        acc = correct / total

        print('[%d / %d]: Accuracy on test set: %.1f %% ' % (epoch+1, EPOCH, 100 \* acc)) # 求测试的准确率，正确数/总数

 return acc

# Start train and Test --------------------------------------------------------------------------------------

if \_\_name\_\_ == '\_\_main\_\_':

 acc\_list\_test = []

 for epoch in range(EPOCH):

    train(epoch)

    # if epoch % 10 == 9: #每训练 10 轮 测试 1 次

    acc\_test = test()

    acc\_list\_test.append(acc\_test)

 plt.plot(acc\_list\_test)

 plt.xlabel('Epoch')

 plt.ylabel('Accuracy On TestSet')

 plt.show()