

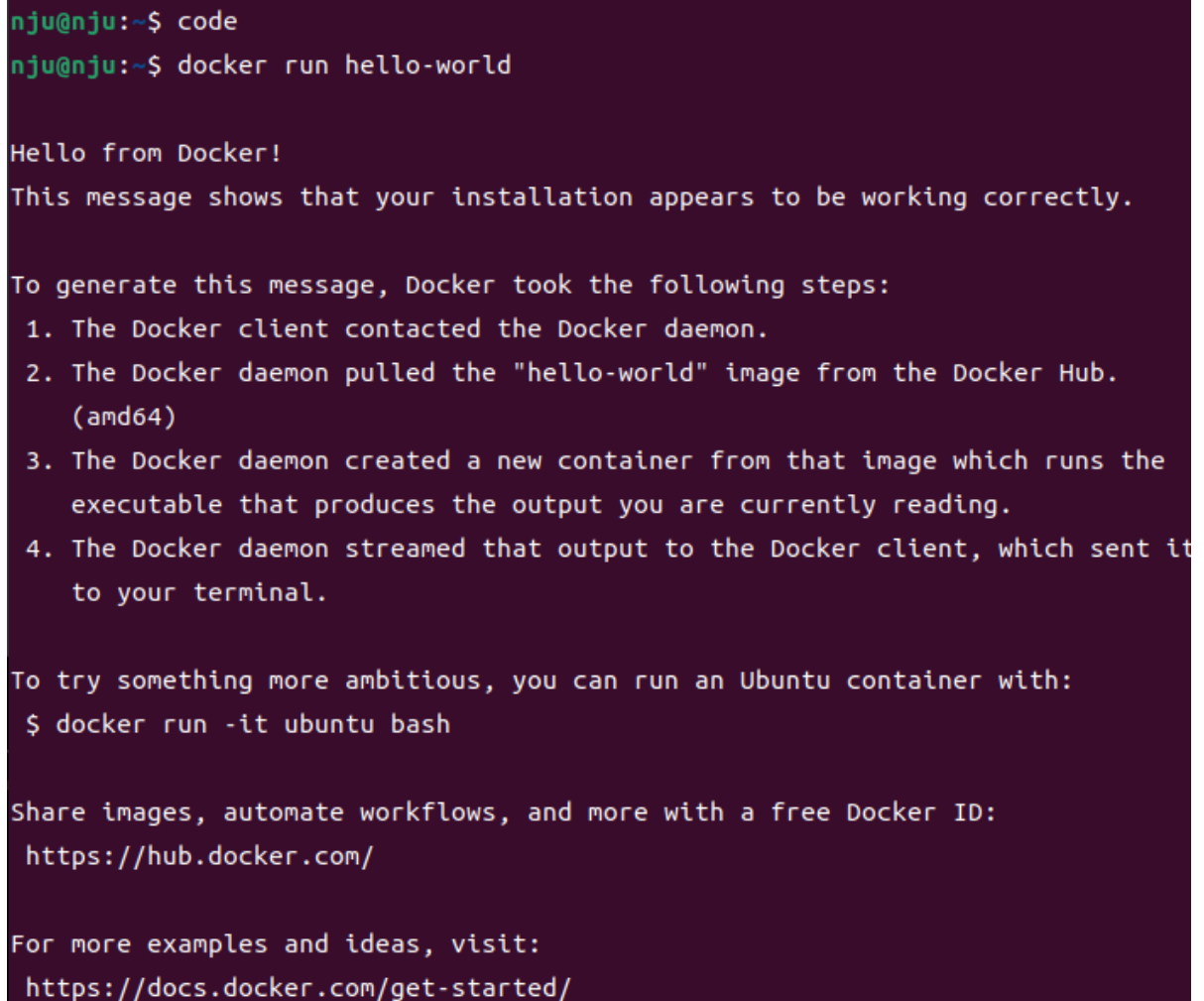
实验报告

221900073 孙佳琪

lab1

安装Docker

安装成功的截图如下所示：

A terminal window with a dark purple background and light green text. The prompt is 'nju@nju:~\$'. The user enters 'code' and then 'docker run hello-world'. The output shows 'Hello from Docker!' followed by a confirmation message and a list of steps Docker took. It also provides instructions on how to run an Ubuntu container and links to Docker resources.

```
nju@nju:~$ code
nju@nju:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

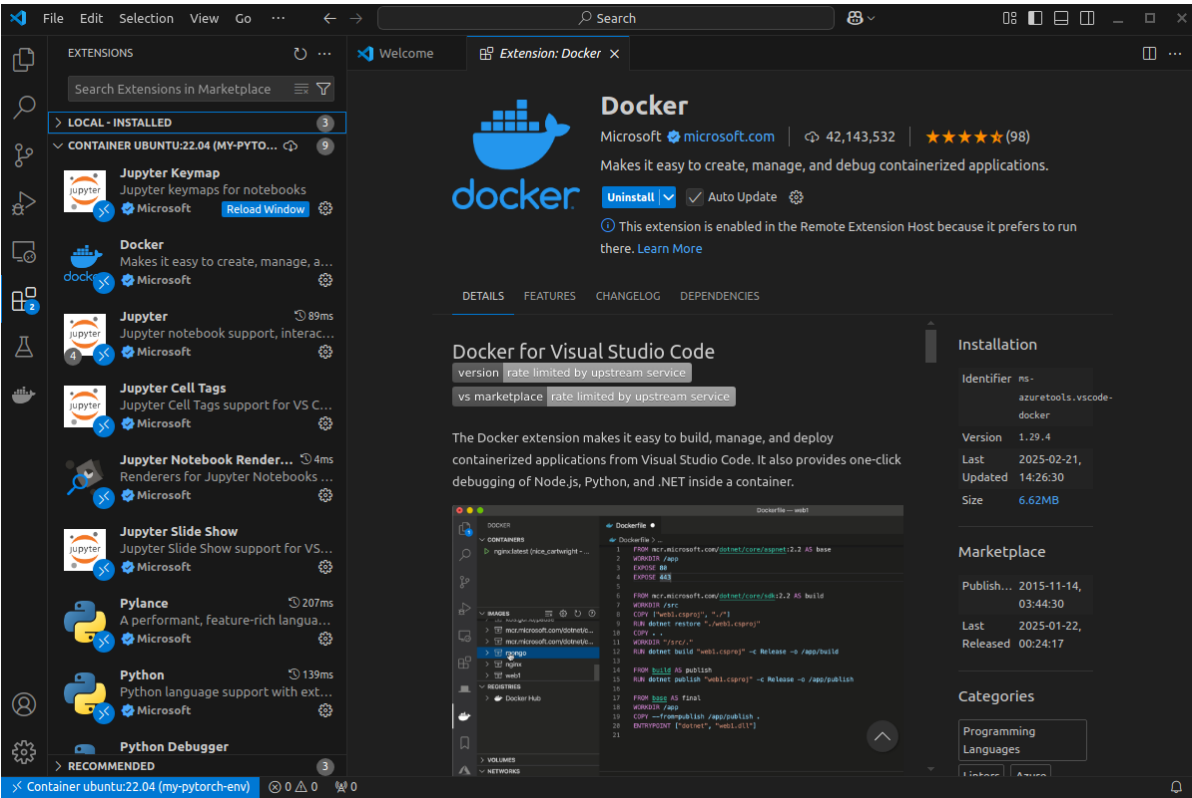
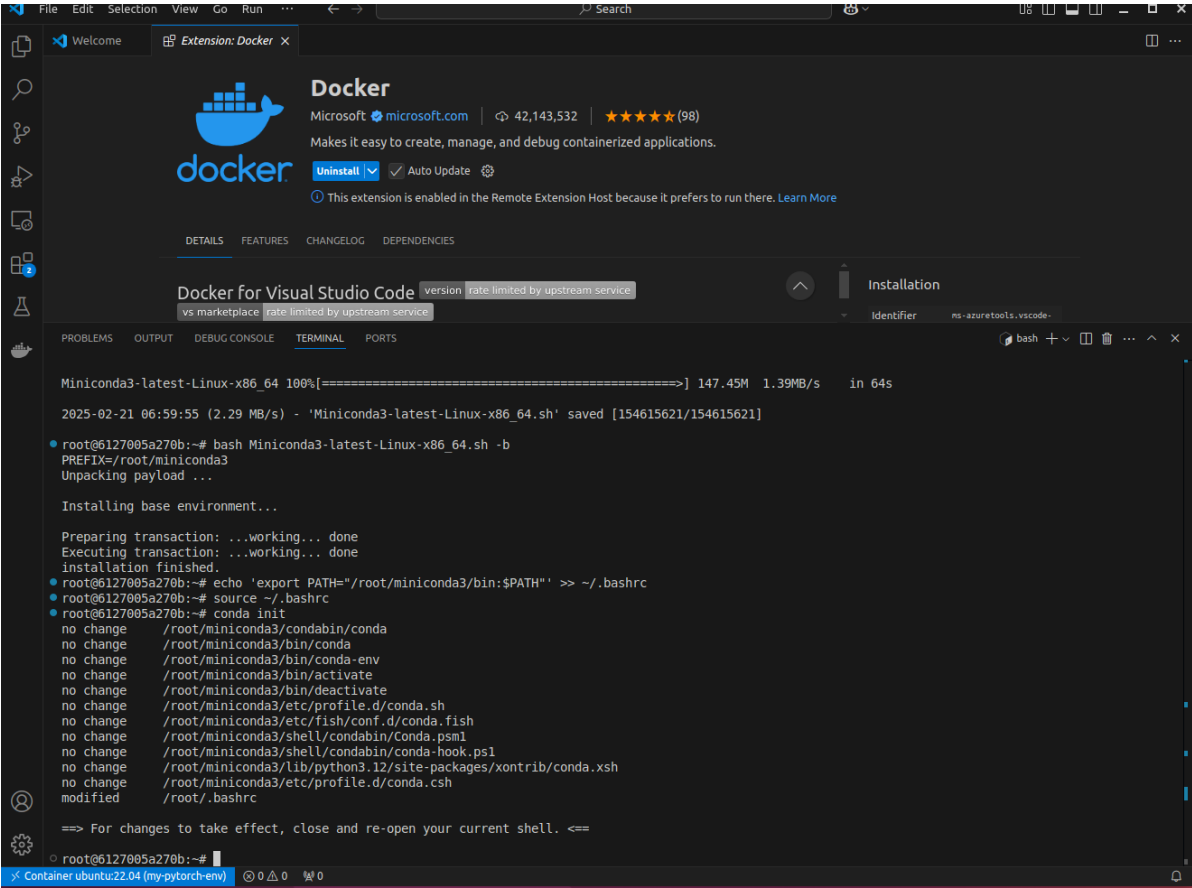
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

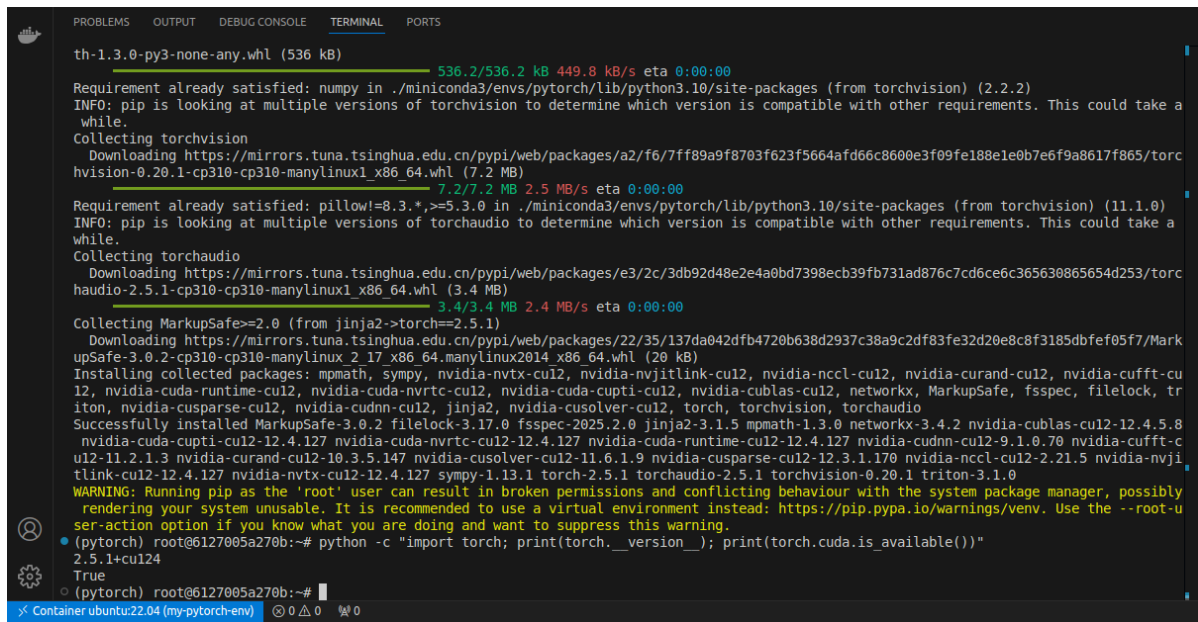
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

配置VSCode使用Docker



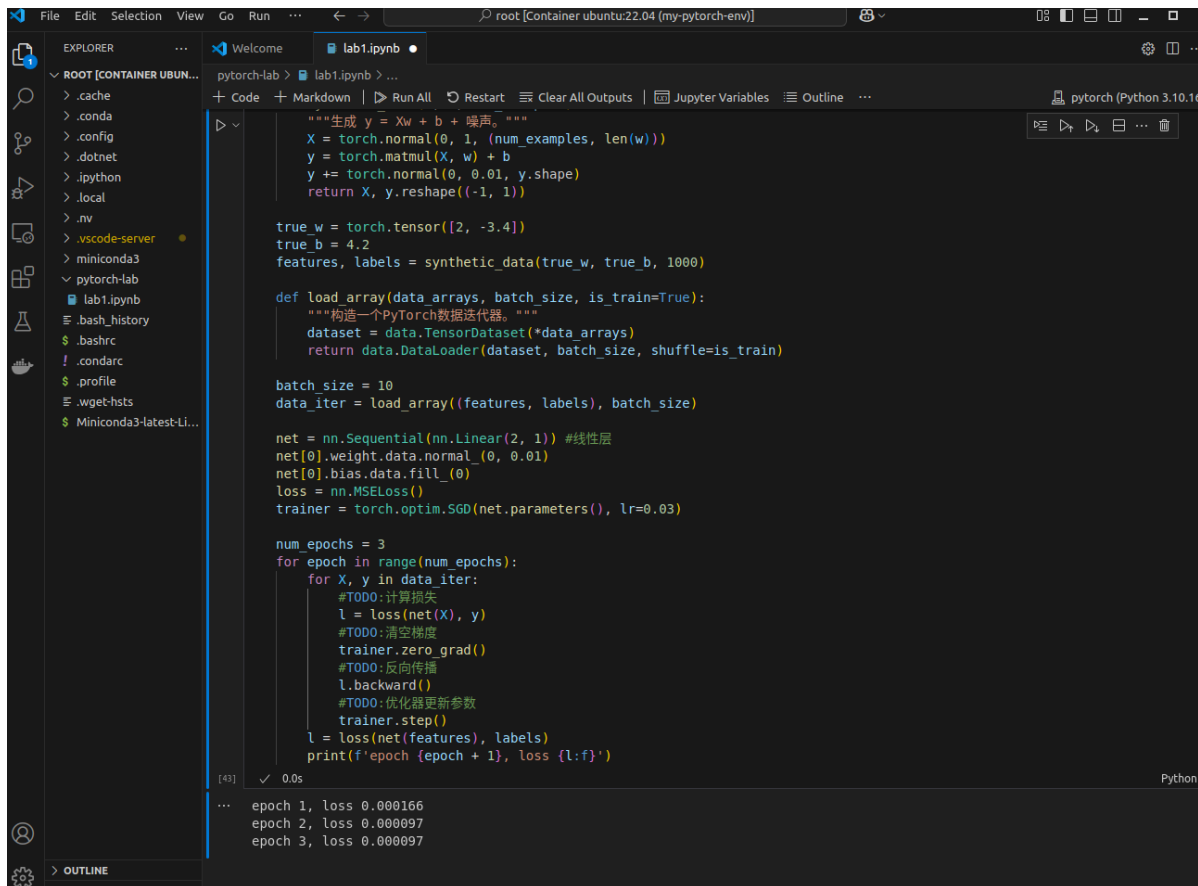
根据该截图所示，已经进入到Docker容器内部，相关插件也已经下载完毕。



```
th-1.3.0-py3-none-any.whl (536 kB)
Requirement already satisfied: numpy in ./miniconda3/envs/pytorch/lib/python3.10/site-packages (from torchvision) (2.2.2)
INFO: pip is looking at multiple versions of torchvision to determine which version is compatible with other requirements. This could take a while.
Collecting torchvision
  Downloading https://mirrors.tuna.tsinghua.edu.cn/pypi/web/packages/a2/f6/7ff89a9f8703f623f5664afd66c8600e3f09fe188e1e0b7e6f9a8617f865/torcvision-0.20.1-cp310-cp310-manylinux1_x86_64.whl (7.2 MB)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in ./miniconda3/envs/pytorch/lib/python3.10/site-packages (from torchvision) (11.1.0)
INFO: pip is looking at multiple versions of torchaudio to determine which version is compatible with other requirements. This could take a while.
Collecting torchaudio
  Downloading https://mirrors.tuna.tsinghua.edu.cn/pypi/web/packages/e3/2c/3db92d48e2e4a0bd7398ecb39fb731ad876c7cd6ce6c365630865654d253/torchaudio-2.5.1-cp310-cp310-manylinux1_x86_64.whl (3.4 MB)
Collecting MarkupSafe>=2.0 (from jinja2->torch==2.5.1)
  Downloading https://mirrors.tuna.tsinghua.edu.cn/pypi/web/packages/22/35/137da042dfb4720b638d2937c38a9c2df83fe32d20e8c8f3185dbfef05f7/MarkupSafe-3.0.2-cp310-cp310-manylinux2014_x86_64.whl (20 kB)
Installing collected packages: mpmath, sympy, nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-cuda-runtime-cu12, nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, networkx, MarkupSafe, fsspec, filelock, triton, nvidia-cuspars-cu12, nvidia-cudnn-cu12, jinja2, nvidia-cusolver-cu12, torch, torchvision, torchaudio
Successfully installed MarkupSafe-3.0.2 filelock-3.17.0 fsspec-2025.2.0 jinja2-3.1.5 mpmath-1.3.0 networkx-3.4.2 nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-cu12-12.4.127 nvidia-cuda-nvrtc-cu12-12.4.127 nvidia-cuda-runtime-cu12-12.4.127 nvidia-cudnn-cu12-9.1.0.70 nvidia-cufft-cu12-11.2.1.3 nvidia-curand-cu12-10.3.5.147 nvidia-cusolver-cu12-11.6.1.9 nvidia-cuspars-cu12-12.3.1.170 nvidia-nccl-cu12-2.21.5 nvidia-nvjitlink-cu12-12.4.127 nvidia-nvtx-cu12-12.4.127 sympy-1.13.1 torch-2.5.1 torchaudio-2.5.1 torchvision-0.20.1 triton-3.1.0
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager, possibly rendering your system unusable. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv. Use the --root-user-action option if you know what you are doing and want to suppress this warning.
(pytorch) root@6127005a270b:~# python -c "import torch; print(torch.__version__); print(torch.cuda.is_available())"
2.5.1+cu124
True
(pytorch) root@6127005a270b:~#
```

版本号以及是否能运行pytorch如上所示。

pytorch练习



```
File Edit Selection View Go Run ... lab1.ipynb
pytorch-lab > lab1.ipynb > ...
+ Code + Markdown | Run All | Restart | Clear All Outputs | Jupyter Variables | Outline ...
pytorch (Python 3.10.16)

"""生成 y = Xw + b + 噪声."""
X = torch.normal(0, 1, (num_examples, len(w)))
y = torch.matmul(X, w) + b
y += torch.normal(0, 0.01, y.shape)
return X, y.reshape((-1, 1))

true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)

def load_array(data_arrays, batch_size, is_train=True):
    """构造一个PyTorch数据迭代器."""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)

net = nn.Sequential(nn.Linear(2, 1)) #线性层
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
loss = nn.MSELoss()
trainer = torch.optim.SGD(net.parameters(), lr=0.03)

num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        #TODO:计算损失
        l = loss(net(X), y)
        #TODO:清空梯度
        trainer.zero_grad()
        #TODO:反向传播
        l.backward()
        #TODO:优化器更新参数
        trainer.step()
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')

[43] ✓ 0.0s
... epoch 1, loss 0.000166
epoch 2, loss 0.000097
epoch 3, loss 0.000097
```

按序使用所给代码中的函数，计算损失、清空梯度、反向传播、优化器更新参数，最后得到的运行结果如图所示。

通过本次实验，我对张量（tensor）有了一些基本的认识，也学会了如何创建、操作和转换张量，例如使用 `torch.arange` 生成序列、使用 `reshape` 改变张量的形状等。这些操作也会是构建和训练模型的基础。模仿第四件的训练过程，使用文档中介绍的API最终补全了训练过程。

lab2

lab2_p1

思考题

如果不打乱训练集，在训练过程中可能会遇到以下问题：

过拟合：模型可能会过度拟合训练数据的顺序，导致在测试数据上的表现不佳。

训练不稳定：如果训练数据有某种顺序（例如，所有同一类别的样本都集中在一起），模型可能会在训练过程中偏向于某些类别，导致训练过程不稳定。

收敛速度慢：打乱数据可以帮助模型更快地收敛，因为每次迭代都能看到不同的数据分布。如果不打乱数据，模型可能需要更多的迭代次数才能收敛。

准确率函数

定义准确率函数

+ 代码 + Markdown

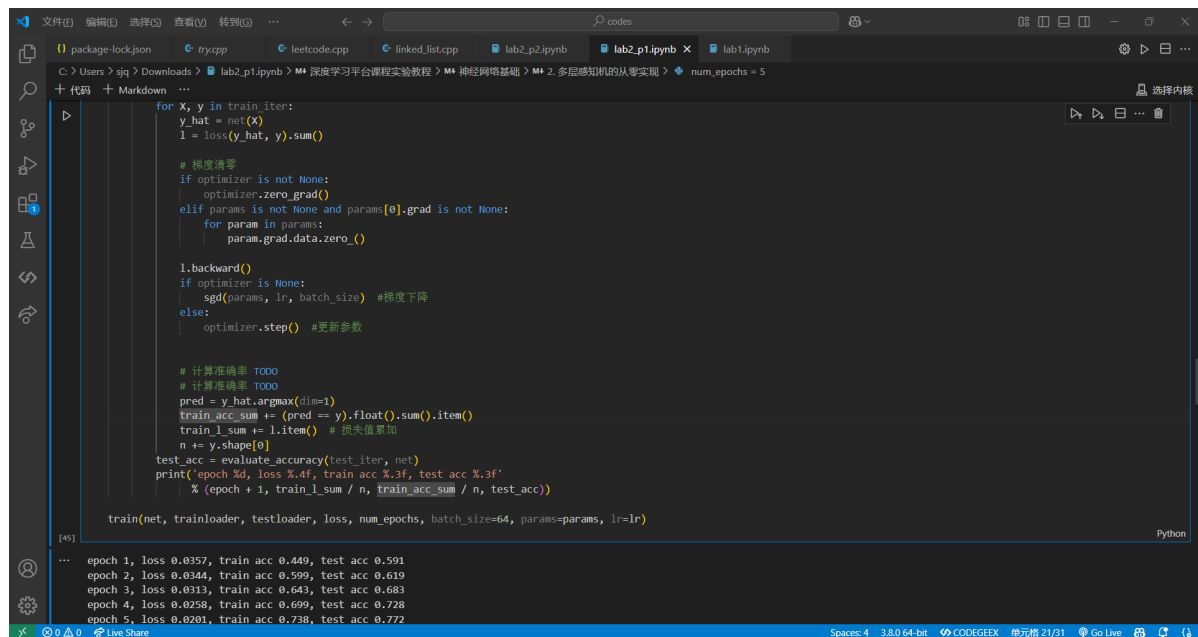
```
def evaluate_accuracy(dataloader, net):
    acc_sum, n = 0.0, 0
    for x, y in dataloader:
        y_hat = net(x)
        pred = y_hat.argmax(dim=1)
        acc_sum += (pred == y).float().sum().item()
        n += y.shape[0]
    return acc_sum / n
```

[44] Python

函数的实现如图所示，`y_hat` 是模型对每个样本的预测结果，通常是一个概率分布。使用 `argmax(dim=1)` 函数从 `y_hat` 中提取每个样本的预测类别。`dim=1` 表示在类别维度上取最大值对应的索引。将预测类别 `pred` 与真实标签 `y` 进行比较，得到一个布尔张量。将当前批次的样本数量 `y.shape[0]` 累加到 `n` 中，最终准确率通过 `acc_sum / n` 计算，即正确预测的样本数量除以总样本数量。

在多层感知机的简单实现部分中，准确率函数也是这样子实现。

训练模型



```
for x, y in train_iter:
    y_hat = net(x)
    l = loss(y_hat, y).sum()

    # 梯度清零
    if optimizer is not None:
        optimizer.zero_grad()
    elif params is not None and params[0].grad is not None:
        for param in params:
            param.grad.data.zero_()

    l.backward()
    if optimizer is None:
        sgd(params, lr, batch_size) # 梯度下降
    else:
        optimizer.step() # 更新参数

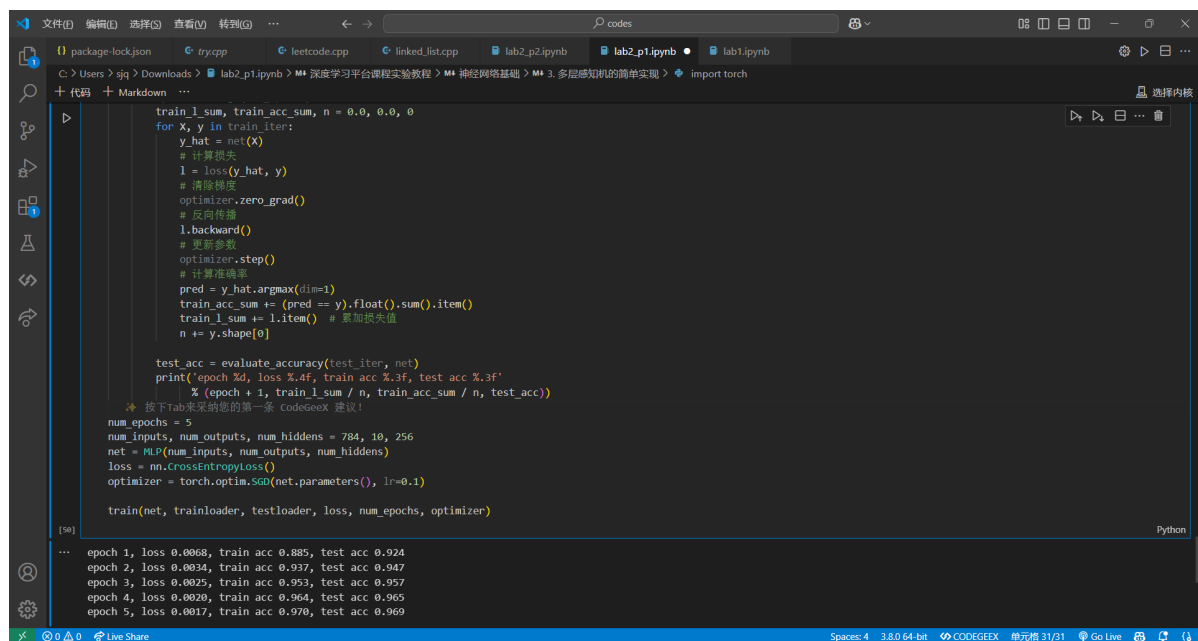
    # 计算准确率 TODO
    # 计算准确率 TODO
    pred = y_hat.argmax(dim=1)
    train_acc_sum += (pred == y).float().sum().item()
    train_l_sum += l.item() # 损失值累加
    n += y.shape[0]
test_acc = evaluate_accuracy(test_iter, net)
print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
      % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))

train(net, trainloader, testloader, loss, num_epochs, batch_size=64, params=params, lr=lr)
```

epoch 1, loss 0.0357, train acc 0.449, test acc 0.591
epoch 2, loss 0.0344, train acc 0.599, test acc 0.619
epoch 3, loss 0.0313, train acc 0.643, test acc 0.683
epoch 4, loss 0.0258, train acc 0.699, test acc 0.728
epoch 5, loss 0.0201, train acc 0.738, test acc 0.772

训练模型的代码以及运行结果如图所示。

需要不全的是损失值和训练准确率的部分，损失值累加即可得到 `train_l_sum` 的值，`train_acc_sum` 则是需要通过累加预测值等于实际值的情况。



```
train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
for x, y in train_iter:
    y_hat = net(x)
    # 计算损失
    l = loss(y_hat, y)
    # 清除梯度
    optimizer.zero_grad()
    # 反向传播
    l.backward()
    # 更新参数
    optimizer.step()
    # 计算准确率
    pred = y_hat.argmax(dim=1)
    train_acc_sum += (pred == y).float().sum().item()
    train_l_sum += l.item() # 累加损失值
    n += y.shape[0]

test_acc = evaluate_accuracy(test_iter, net)
print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
      % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))
# 按下Tab来采纳您的第一条 CodeGeex 建议！

num_epochs = 5
num_inputs, num_outputs, num_hiddens = 784, 10, 256
net = MLP(num_inputs, num_outputs, num_hiddens)
loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

train(net, trainloader, testloader, loss, num_epochs, optimizer)
```

epoch 1, loss 0.0069, train acc 0.885, test acc 0.924
epoch 2, loss 0.0034, train acc 0.937, test acc 0.947
epoch 3, loss 0.0025, train acc 0.953, test acc 0.957
epoch 4, loss 0.0020, train acc 0.964, test acc 0.965
epoch 5, loss 0.0017, train acc 0.970, test acc 0.969

多层感知机部分的训练模型代码以及运行结果如图所示。

按照注释实现相关步骤即可实现。

lab2_p2

激活函数的实现与可视化

```
# 自己实现激活函数及其导数
class MyReLU:
    def __call__(self, x):
        """实现ReLU激活函数:  $f(x) = \max(0, x)$ """
        return torch.maximum(torch.tensor(0), x)

    def derivative(self, x):
        """ReLU的导数:  $f'(x) = 1$  if  $x > 0$  else  $0$ """
        return torch.where(x > 0, torch.tensor(1.0), torch.tensor(0.0))

class MySigmoid:
    def __call__(self, x):
        """实现Sigmoid激活函数:  $f(x) = 1 / (1 + e^{-x})$ """
        return 1 / (1 + torch.exp(-x))

    def derivative(self, x):
        """Sigmoid的导数:  $f'(x) = f(x) * (1 - f(x))$ """
        sigmoid_x = self.__call__(x)
        return sigmoid_x * (1 - sigmoid_x)

class MyTanh:
    def __call__(self, x):
        """实现Tanh激活函数:  $f(x) = (e^x - e^{-x}) / (e^x + e^{-x})$ """
        return (torch.exp(x) - torch.exp(-x)) / (torch.exp(x) + torch.exp(-x))

    def derivative(self, x):
        """Tanh的导数:  $f'(x) = 1 - f(x)^2$ """
        tanh_x = self.__call__(x)
        return 1 - tanh_x ** 2
```

运行结果在.ipynb文件中有显示。

思考题：

神经网络的核心优势在于其能够学习和表示复杂的非线性关系。非线性激活函数（如ReLU、Sigmoid、Tanh等）能够将输入数据映射到非线性空间，使得神经网络能够拟合复杂的函数。如果没有非线性激活函数，无论神经网络有多少层，它都只能表示线性关系。许多实际问题（如图像分类、语音识别等）都是非线性的，使用线性激活函数将无法有效解决这些问题。

梯度消失问题实验

```
def forward(self, x):  
    # TODO: 实现前向传播函数  
    # 1. 首先通过输入层  
    # 2. 应用激活函数  
    # 3. 依次通过每个隐藏层并应用激活函数  
    # 4. 最后通过输出层返回结果  
    x = self.input_layer(x)  
    x = self.activation(x)  
    for layer in self.layers:  
        x = layer(x)  
        x = self.activation(x)  
    x = self.output_layer(x)  
    return x
```

前向传播函数的实现如图所示。

运行结果在.ipynb文件中有显示。

思考题：

Sigmoid函数的导数范围是(0, 0.25)。当输入值较大或较小时，梯度接近于0，容易导致梯度消失问题。Tanh函数的导数范围是(0, 1)。虽然比Sigmoid稍好，但在输入值较大或较小时，梯度仍然接近于0，可能导致梯度消失。ReLU函数的导数在正区间内为1，在负区间内为0。ReLU在正区间内不会出现梯度消失问题，但在负区间内可能导致神经元“死亡”。

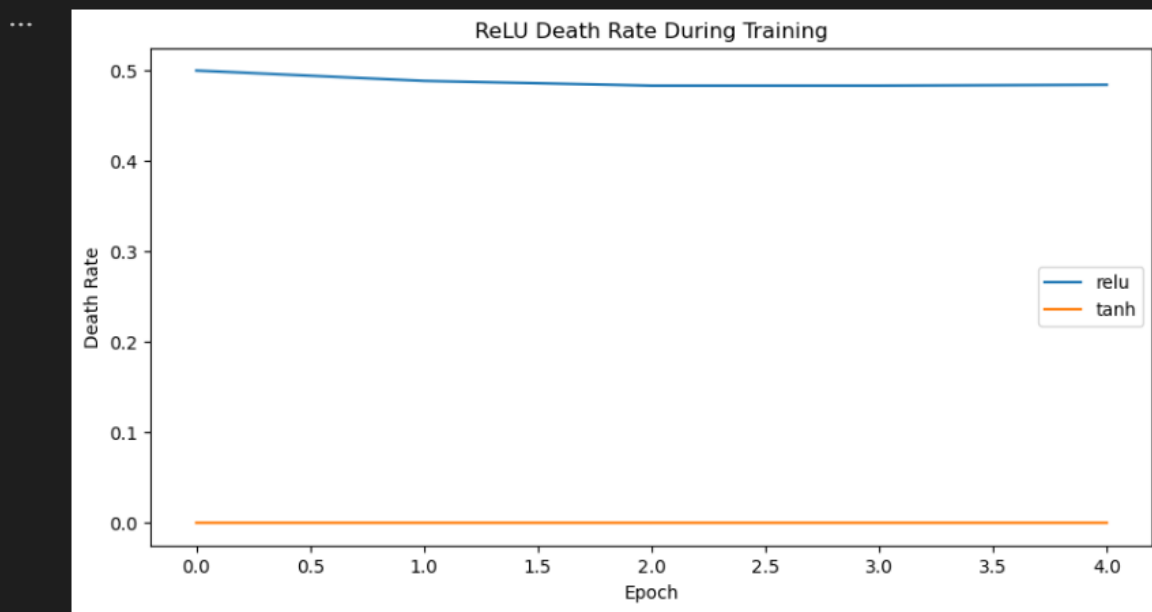
梯度消失问题是指在反向传播过程中，梯度逐渐变小，导致网络参数更新缓慢甚至停止更新。这种现象在深层神经网络中尤为明显，因为梯度需要通过多层传递，每层的梯度都会进一步减小。对于Sigmoid和Tanh来说，于它们的导数在输入值较大或较小时接近于0，容易导致梯度消失问题，特别是在深层网络中。这使得网络难以训练，训练准确率较低。

ReLU在正区间内的导数为1，不会出现梯度消失问题，使得深层网络更容易训练。ReLU的计算非常简单，只需要判断输入是否大于0，计算效率高。ReLU在负区间内输出为0，使得部分神经元不会被激活，从而产生稀疏激活效果，减少计算量并提高模型的泛化能力。

不同激活函数的梯度分布不同，影响反向传播过程中梯度的传递。ReLU的梯度分布更有利于深层网络的训练，避免梯度消失问题，从而提高训练准确率。

ReLU死亡现象实验

```
... Activation: relu, Epoch 0: Death Rate: 0.50%
Activation: relu, Epoch 1: Death Rate: 0.49%
Activation: relu, Epoch 2: Death Rate: 0.48%
Activation: relu, Epoch 3: Death Rate: 0.48%
Activation: relu, Epoch 4: Death Rate: 0.48%
Activation: tanh, Epoch 0: Death Rate: 0.00%
Activation: tanh, Epoch 1: Death Rate: 0.00%
Activation: tanh, Epoch 2: Death Rate: 0.00%
Activation: tanh, Epoch 3: Death Rate: 0.00%
Activation: tanh, Epoch 4: Death Rate: 0.00%
```



思考题：

ReLU死亡现象成因：

ReLU激活函数在输入为负时输出为0，且导数为0。如果某个神经元的输入一直为负，其梯度在反向传播过程中将一直为0，导致权重无法更新。

如果学习率设置过高，权重更新幅度过大，可能导致神经元输出进入负区间，从而引发ReLU死亡现象。

如果权重初始化不当，可能导致某些神经元的初始输出为负，从而在训练初期就进入死亡状态。

解决方案：

动态调整学习率，如在训练初期使用较高的学习率，随着训练的进行逐渐降低学习率。避免神经元输出进入负区间，从而减少ReLU死亡现象的发生。

可以该并权重初始化，如采用Xavier或者He初始化，减少死亡现象。

LeakyReLU在负区间内引入一个小的斜率（如0.01），使得负输入时输出不为0，从而避免神经元死亡。PReLU在负区间内引入一个可学习的斜率参数，使得模型能够自适应地调整负区间的输出。ELU在负区间内引入一个指数函数，使得负输入时输出不为0，且具有平滑的过渡。使用ReLU变体可以有效缓解ReLU死亡现象，提高神经网络的训练效果和泛化能力。

正则化方法实验

```
def l2_penalty(w):  
    # TODO: 使用torch.sum和.pow方法实现L2范数惩罚  
    # 提示: 对权重参数w平方求和, 再除以2  
    return torch.sum(w.pow(2)) / 2
```

[21] Python

运行结果在.ipynb文件中有显示。

思考题:

在使用了L2正则化之后:

权重数值变小: L2正则化倾向于使权重值变小, 因为较大的权重会增加正则化项的值, 从而增加总损失。因此, 优化过程中会倾向于选择较小的权重。

权重的分布更加集中: L2正则化使得权重分布更加集中, 避免出现极端大的权重值, 从而减少模型的复杂度。

其中, 正则化系数 λ 控制正则化项的强度, 影响模型参数的变化。较大的 λ 会显著增加正则化项的影响, 使得权重值更小, 模型更加简单。然而, 过大的 λ 可能导致模型欠拟合, 无法捕捉数据中的复杂模式。较小的 λ 会减少正则化项的影响, 允许权重值较大, 模型更加复杂。然而, 过小的 λ 可能导致模型过拟合, 无法泛化到新数据。

较小的权重使得模型更加简单, 减少了模型的复杂度, 更倾向于捕捉数据中的主要模式, 使得模型对输入数据的变化更加稳定, 减少了对训练数据中噪声的敏感性, 从而提高模型的泛化能力。较大的权重值可能导致模型对某些特征过度依赖, 增加过拟合的风险。较小的权重值使得模型对所有特征更加均衡地依赖, 减少过拟合的可能性。

Dropout

```
def dropout_layer(X, dropout):  
    assert 0 <= dropout <= 1  
    device = X.device  
    # TODO: 实现dropout层  
    # 1. 如果dropout=1, 返回全0张量  
    # 2. 如果dropout=0, 直接返回输入X  
    # 3. 否则, 生成一个与X形状相同的随机掩码(mask)  
    #     - 使用torch.rand生成随机数, 并与dropout比较创建二元掩码  
    #     - 将X与掩码相乘, 并除以(1-dropout)进行缩放  
    # 注意: 过程中device参数需要与X的设备相同  
    # 在本情况中, 所有元素都被丢弃  
    if dropout == 1:  
        return torch.zeros_like(X, device=device)  
    # 在本情况中, 所有元素都被保留  
    if dropout == 0:  
        # TODO: 实现dropout=0的情况  
        return X  
    # TODO: 实现dropout=其他值的情况  
    mask = (torch.rand(X.shape, device=device) > dropout).float()  
    return mask * X / (1.0 - dropout)
```

Python

dropou层的实现如图所示。

思考题：

Dropout通过随机丢弃神经元，减少了神经元之间的复杂共适应关系。这意味着网络不能依赖于某些特定的神经元或特征组合，从而迫使网络学习更加鲁棒的特征。这有助于防止过拟合，因为模型不会过度依赖训练数据中的特定模式。所以，Dropout有效地起到了正则化的作用。

在训练时，Dropout随机丢弃神经元，增加了模型的多样性，提高了泛化能力。在测试时，所有神经元都参与预测，但为了保持输出的期望值一致，需要对权重进行比例缩放。这种差异处理确保了模型在训练和测试时的一致性，同时保留了Dropout的正则化效果。