linux 安装 anaconda 及配置 pytorch 环境

1.下载与安装 Anaconda

- **下载方式**：可将 Windows 上的安装包通过 xftp 传到 Linux 服务器，也可在 Linux 上用`wget`命令直接下载，示例命令为`wget https://repo.anaconda.com/archive/Anaconda3-5.2.0-Linux-x86_64.sh`，还可选择清华镜像源以提高速度。
- 安装步骤：
  - 进入下载目录，运行`bash Anaconda3-5.2.0-Linux-x86_64.sh`命令开始安装。
  - 按回车查看协议，直至出现接受协议的提示，输入“yes”。
  - 直接回车选择默认安装位置（家目录），等待安装完成。
  - 提示是否将 Anaconda3 安装位置添加到 PATH 时，输入“yes”。
  - 提示是否下载 VSCode 时，输入“no”跳过。
- **配置检查**：安装完成后，用`conda -V`命令查看版本，若提示“未找到命令”，则用`export PATH=/home/yourName/anaconda3/bin/:$PATH`命令将 conda 加入系统路径，再重新查看版本确认安装配置完成。

2.使用 conda 创建新环境及安装 PyTorch

- **添加国内镜像源**：通过一系列`conda config --add channels`命令添加清华等国内镜像源，可用`conda config --show-sources`查看配置的源。
- **创建新环境**：使用`conda create -n envName python=3.8`命令创建新环境，其中`envName`为环境名，之后用`source activate envName`进入该环境。
- 安装 PyTorch：
  - 用`nvidia-smi`命令查看服务器的 CUDA 版本，选择对应版本的 PyTorch。
  - 从 PyTorch 官网获取安装命令，去掉`-c pytorch`（避免从官网下载速度慢），示例命令为`conda install pytorch torchvision torchaudio cudatoolkit=10.2`。
  - 若 conda 安装报错，可改用 pip 安装，命令为`pip3 install torch torchvision torchaudio`。

3.检验安装结果

- **判断 PyTorch 是否安装成功**：在命令行输入`python`，再输入`import torch`，若未报错则安装成功。
- **检验是否可使用 GPU**：在 Python 环境中输入`torch.cuda.is_available()`，返回“true”表示可使用 GPU。

在 WSL2/Linux 中安装 Jupyter Notebook 并在 Windows 中使用

一、安装环境准备

1. 启用 WSL2 并安装 Linux 发行版

- 以管理员身份打开 PowerShell，执行以下命令启用 WSL 功能：

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux
/all /norestart
```

- 启用虚拟机平台功能：

```
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all
/norestart
```

- 重启计算机。
- 从 Microsoft Store 安装喜欢的 Linux 发行版，如 Ubuntu。

2. 更新和升级 Linux 系统

打开安装好的 Linux 发行版，执行以下命令更新包列表并升级系统：

```
sudo apt update
sudo apt upgrade -y
```

二、安装 Jupyter Notebook

1. 安装 Python 和 pip

- 大多数 Linux 发行版已预装 Python，但可能需要安装 pip：

```
sudo apt install python3-pip -y
```

- 验证安装：

```
python3 --version
pip3 --version
```

2. 使用 pip 安装 Jupyter Notebook

```
pip3 install jupyter notebook
```

3. 验证安装

```
jupyter --version
```

三、配置 Jupyter Notebook

1. 生成配置文件

```
jupyter notebook --generate-config
```

配置文件将生成在 `~/.jupyter/jupyter_notebook_config.py`。

2. 配置远程访问

- 打开配置文件：

```
nano ~/.jupyter/jupyter_notebook_config.py
```

- 找到并修改以下行（取消注释并修改值）：

```
c.NotebookApp.ip = '0.0.0.0'          # 允许所有 IP 访问
c.NotebookApp.port = 8888             # 设置端口（可选）
c.NotebookApp.open_browser = False    # 不自动打开浏览器
c.NotebookApp.notebook_dir = '~/'     # 设置工作目录（可选）
```

- 按 `Ctrl+X`，`Y`，`Enter` 保存并退出。

3. 创建密码

- 在终端中运行：

```
jupyter notebook password
```

- 输入并确认密码，密码将被加密存储在配置文件中。

四、启动 Jupyter Notebook 并在 Windows 中访问

1. 在 WSL2 中启动 Jupyter Notebook

```
jupyter notebook
```

- 启动后，终端会显示类似以下信息：

```
[I 12:34:56.789 NotebookApp] Serving notebooks from local directory: /home/user
[I 12:34:56.789 NotebookApp] Jupyter Notebook 6.4.12 is running at:
[I 12:34:56.789 NotebookApp] http://(WSL2-IP or localhost):8888/?
token=your_token_here
```

2. 在 Windows 浏览器中访问

- 打开 Windows 浏览器（如 Chrome、Firefox 等）。
- 输入 Jupyter Notebook 启动时显示的 URL，例如：

```
http://localhost:8888/
```

或使用 WSL2 的 IP 地址（如果 localhost 无法访问）：

```
http://172.24.192.1:8888/  # 替换为你的 WSL2 IP
```

- 如果设置了密码，输入密码登录。

五、Jupyter Notebook 使用基础

1. 界面概述

- **文件浏览器**：显示当前工作目录中的文件和文件夹。
- **新建笔记本**：点击右上角的 "New" 按钮，选择 Python 3 创建新笔记本。
- **打开现有笔记本**：点击笔记本文件名打开。

2. 单元格操作

- **代码单元格**：用于编写和执行代码。
- **Markdown 单元格**：用于添加文本说明，支持 Markdown 语法。
- **切换单元格类型**：使用工具栏中的下拉菜单或快捷键（`Esc + M` 切换为 Markdown，`Esc + Y` 切换为代码）。

3. 执行代码

- 在代码单元格中输入 Python 代码。
- 按 `Shift + Enter` 执行代码并移动到下一个单元格。
- 按 `Ctrl + Enter` 执行代码但不移动。

4. 常用快捷键

- **编辑模式**：单元格内有光标时（绿色边框）。
- 命令模式
  ：点击单元格外部（蓝色边框）。
  - `Esc`：从编辑模式进入命令模式。
  - `Enter`：从命令模式进入编辑模式。
  - `A`：在当前单元格上方插入新单元格。
  - `B`：在当前单元格下方插入新单元格。
  - `D + D`（两次）：删除当前单元格。
  - `M`：将单元格转换为 Markdown。
  - `Y`：将单元格转换为代码。
  - `Shift + Enter`：执行单元格并移动到下一个。
  - `Ctrl + S`：保存笔记本。

5. 保存和导出

- 点击工具栏中的保存按钮或按 `Ctrl + S` 保存笔记本。
- 通过 "File > Download as" 可以将笔记本导出为多种格式（如 PDF、HTML、Python 等）。

六、故障排除

1. localhost 无法访问

- 检查 WSL2 的 IP 地址：

```
hostname -I
```

- 使用该 IP 地址代替 localhost 访问。

2. 端口被占用

- 停止占用端口的程序，或修改 Jupyter 配置文件中的端口号。

3. 密码忘记

- 删除密码文件：

```
rm ~/.jupyter/jupyter_notebook_config.json
```

- 重新设置密码。

---

pytorch常用

一、张量操作（Tensor Operations）

张量是 PyTorch 的核心数据结构，类似于 NumPy 数组，但支持 GPU 加速和自动微分。

1. 创建张量

```
import torch

# 直接创建
x = torch.tensor([1, 2, 3])
x = torch.zeros(5, 3)  # 全零矩阵
x = torch.ones(5, 3)   # 全一矩阵
x = torch.rand(5, 3)   # 随机矩阵（0-1均匀分布）
x = torch.randn(5, 3)  # 随机矩阵（标准正态分布）

# 从 NumPy 转换
import numpy as np
np_array = np.array([1, 2, 3])
torch_tensor = torch.from_numpy(np_array)

# 转换回 NumPy
numpy_array = torch_tensor.numpy()
```

2. 张量操作

```
# 基本运算
x + y  # 加法
x * y  # 逐元素乘法
torch.matmul(x, y)  # 矩阵乘法

# 形状操作
x.reshape(3, 5)  # 重塑张量形状
x.view(3, 5)     # 与 reshape 类似，但可能返回视图而非副本
x.transpose(0, 1)  # 转置

# 统计操作
x.mean()         # 均值
x.sum()          # 求和
x.max(dim=0)     # 最大值及其索引
```

3. 设备管理

```
# 检查 GPU 是否可用
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 将张量移至 GPU
x = x.to(device)

# 创建时直接指定设备
x = torch.tensor([1, 2, 3], device=device)
```

二、自动微分（Autograd）

PyTorch 使用 `autograd` 模块实现自动求导，跟踪张量上的所有操作。

1. 梯度计算

```
# 创建需要计算梯度的张量
x = torch.tensor([2.0], requires_grad=True)

# 定义函数
y = x**2 + 3*x + 1

# 反向传播
y.backward()

# 查看梯度
print(x.grad)  # 输出: tensor([7.])
```

2. 禁用梯度计算（用于推理）

```
with torch.no_grad():
    y = model(x)  # 不计算梯度，提高推理速度
```

三、神经网络模块（nn.Module）

PyTorch 提供了 `nn` 模块，用于构建神经网络。

1. 定义简单网络

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)  # 输入通道3，输出通道6，卷积核5x5
        self.pool = nn.MaxPool2d(2, 2)   # 最大池化，步长2
        self.fc1 = nn.Linear(6 * 14 * 14, 120)  # 全连接层
        self.fc2 = nn.Linear(120, 10)   # 输出层

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = x.view(-1, 6 * 14 * 14)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Net()
```

2. 损失函数和优化器

```
import torch.optim as optim

# 定义损失函数（交叉熵损失）
criterion = nn.CrossEntropyLoss()

# 定义优化器（随机梯度下降）
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

# 训练循环
for epoch in range(10):
    optimizer.zero_grad()  # 清零梯度
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()        # 反向传播
    optimizer.step()       # 更新参数
```

四、数据加载与预处理（Data Loading）

使用 `torch.utils.data` 模块处理数据集。

1. 自定义数据集

```
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def __init__(self, data, labels, transform=None):
        self.data = data
```

```
        self.labels = labels
        self.transform = transform

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

    def __getitem__(self, idx):
        sample = self.data[idx]
        label = self.labels[idx]
        if self.transform:
            sample = self.transform(sample)
        return sample, label

# 创建数据加载器
dataset = MyDataset(data, labels)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

2. 常用数据转换

```
from torchvision import transforms

transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 调整图像大小
    transforms.ToTensor(),          # 转换为张量
    transforms.Normalize(           # 标准化
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

五、预训练模型与迁移学习

PyTorch 提供了 `torchvision.models` 模块，包含许多预训练模型。

```
from torchvision import models

# 加载预训练的 ResNet18
model = models.resnet18(pretrained=True)

# 修改最后一层用于自定义分类任务
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 10)  # 假设我们有10个类别
```

六、保存与加载模型

```
# 保存模型
torch.save(model.state_dict(), 'model.pth')

# 加载模型
model = Net()  # 先初始化模型结构
model.load_state_dict(torch.load('model.pth'))
model.eval()   # 设置为评估模式
```

七、常用工具函数

```
# 随机种子设置（保证结果可复现）
torch.manual_seed(42)

# 模型参数数量统计
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

# 学习率调度
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

八、常见问题与技巧

1. **GPU 内存不足**：
    - 减小批量大小（`batch_size`）
    - 使用半精度训练（`torch.float16`）
    - 释放不需要的张量：`del x; torch.cuda.empty_cache()`
2. **调试技巧**：
    - 使用 `print(x.shape)` 检查张量形状
    - 使用 `assert` 语句验证中间结果
    - 使用 `pdb` 调试器
3. **性能优化**：
    - 使用 `torch.no_grad()` 进行推理
    - 使用 `torch.cuda.amp` 进行混合精度训练
    - 利用 `torch.utils.checkpoint` 减少内存占用