**PyTorch**

**简介**

* **pytorch**：用于深度学习的主要框架，提供张量计算和自动求导等功能
* **torchvision**：提供处理计算机视觉任务所需的功能

**安装**

1. 安装python和python包管理器pip
2. python：https://www.python.org/downloads/
3. 用python包管理器安装torch pip3 install torch torchvision
4. 检查安装结果：pip3 list 查看已安装依赖列表
5. 执行torch代码 python3 test.py

|  |
| --- |
| Bash import torch x = torch.rand(5, 3) print(x) |

python3 -v 进入交互式命令

exit() 或 quit() 退出

查看安装包信息

pip3 show numpy

仅查看版本

pip3 show numpy | grep 'Version:'

**基本概念**

* **张量计算 Tensor computation**：多维数组
* **自动求导 Automatic differentiation**：自动计算张量的梯度（导数）

**引入torch及其数据/方法**

|  |
| --- |
| Python import torch # 导入神经网络模块 from torch import nn # 导入DataLoader类，用于批量加载数据 from torch.utils.data import DataLoader # 导入datasets模块，这个模块包含了常见的计算机视觉数据集 from torchvision import datasets # 导入ToTensor类，用于将PIL图像或numpy数组转换为PyTorch张量 from torchvision.transforms import ToTensor |

|  |
| --- |
| Python import tourch |

**tensor**

|  |
| --- |
| Python x = torch.tensor([5.5, 3]) print(x)  # 创建一个未初始化的 5x3 矩阵 x = torch.empty(5, 3) print(x) # 创建一个随机初始化的 5x3 矩阵 x = torch.rand(5, 3) print(x) # 创建一个全部为 0，数据类型为 long 的矩阵 x = torch.zeros(5, 3, dtype=torch.long) print(x) # 创建 tensor 并直接使用数据初始化 x = torch.tensor([5.5, 3]) print(x) |

**用torch训练模型**

1. **引入模块**

|  |
| --- |
| Python import torch # 用于构建神经网络的模块，提供许多构建神经网络层和其他相关功能的类和函数 from torch import nn # 对数据进行处理的模块，提供对数据进行批处理、打乱顺序等操作 from torch.utils.data import DataLoader # torchvision是处理计算机视觉任务的库，datasets模块提供了很多常用的数据集 from torchvision import datasets # 将图像数据转换为PyTorch张量（tensor）格式的工具 from torchvision.transforms import ToTensor |

2. **获取数据集并定义数据转换**

|  |
| --- |
| Python *# 训练数据集datasets，从datasets中下载 FashionMNIST 的数据集并转换* training\_data = datasets.FashionMNIST(  root="data",  train=True,  download=True,  transform=ToTensor(), # 样本samples转为张量  # 还有target\_transform参数，用于对标签labels进行转换，此处不做转换 ) # 测试数据集 test\_data = datasets.FashionMNIST(  root="data",  train=False,  download=True,  transform=ToTensor(), # 样本samples转为张量  # 还有target\_transform参数，用于对标签labels进行转换，此处不做转换 ) |

3. **加载数据集**

|  |
| --- |
| Python batch\_size = 64 # 一次性送入神经网络进行训练或测试的样本数  # 创建训练数据加载器，DataLoader会将训练数据分割成若干批次，并在每个批次之间打乱数据顺序，以提高训练效果。 train\_dataloader = DataLoader(training\_data, batch\_size=batch\_size) # 创建测试数据加载器 test\_dataloader = DataLoader(test\_data, batch\_size=batch\_size)  # 遍历测试数据加载器，X表示样本，y表示标签 for X, y in test\_dataloader:  # X.shape形状，此处是一个四维张量  # N为批次数量  # C为通道数（比如彩色图像有3个通道，RGB）  # H：高度（图像的高度）  # W：宽度（图像的宽度）  print(f"Shape of X [N, C, H, W]: {X.shape}")  # y.shape形状，通常是一维张量，其长度等于批次大小，宽dtype表示数据类型  print(f"Shape of y: {y.shape} {y.dtype}")  break |

4. **初始化模型**

为了提高处理速度，优先使用GPU和MPS

|  |
| --- |
| Python device = (  # GPU，CUDA是NVIDIA推出的并行计算架构  "cuda"  if torch.cuda.is\_available()  # MPS，是Apple为其设备（如MacBook和iPhone）提供的一种高效计算框架  else "mps"  if torch.backends.mps.is\_available()  # CPU相比GPU来说，计算速度会慢一些，但兼容性最好  else "cpu" ) print(f"Using *{*device*}* device") |

在\_\_init\_\_函数定义神经网络的层

在forward函数定义数据处理的顺序（过程）

|  |
| --- |
| Python class NeuralNetwork(nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  # 将输入的二维图像（28X28）展平成一维向量（784）  # 在图像分类任务中，常用这种方法将图像转换为一维向量以便输入到全连接层  self.flatten = nn.Flatten()   # nn.Sequential定义一个包含多个层的顺序容器   self.linear\_relu\_stack = nn.Sequential(  # 全连接层，将784维输入（28x28）印社到512维输出  nn.Linear(28\*28, 512),  # Rectified Linear Unit激活函数，将线性变换后的结果进行非线性变换，以引入非线性特征  nn.ReLU(),  # 另一个全连接层，将512维输入映射到512维输出  nn.Linear(512, 512),  nn.ReLU(),  # 最后一层全连接层，将512维输入映射到10维输出。降维，通常用于分类任务的最终层  nn.Linear(512, 10)  )  def forward(self, x):  x = self.flatten(x)  logits = self.linear\_relu\_stack(x)  return logits model = NeuralNetwork().to(device) print(model) |

5. **定义损失函数和优化器，优化模型参数**

* **损失函数loss function**：计算输出和目标之间的误差
* **优化器optimizer**：更新模型外部参数以**最小化损失**的值。常见的优化器包括SGD（随机梯度下降）和Adam（自适应矩估计）

|  |
| --- |
| Python loss\_fn = nn.CrossEntropyLoss() optimizer = torch.optim.SGD(model.parameters(), lr=1e-3) |

6. **训练过程**

**定义训练函数**

用训练数据集进行训练

|  |
| --- |
| Python def train(dataloader, model, loss\_fn, optimizer):  # 获取数据集的总大小  size = len(dataloader.dataset)  # 将模型设置为训练模式，会启用dropout等训练专用的功能  model.train()  # 循环训练，按批次进行循环  # batch为当前批次的索引，X为当前批次的输入特征，y是当前批次的标签  for batch, (X, y) in enumerate(dataloader):  # 将数据移动到指定的设备（如GPU），以加速计算  X, y = X.to(device), y.to(device)    # 获取模型预测结果  pred = model(X)  # 计算预测结果误差  loss = loss\_fn(pred, y)    # 反向传播**Backpropagation：**一种计算梯度的算法，用于更新神经网络中的参数  loss.backward()  # 使用优化器根据梯度更新模型参数  optimizer.step()  # 清零优化器中的梯度缓存，以免在下一次迭代中累积  optimizer.zero\_grad()    # 每处理100个批次，打印一次当前的算是和进度  if batch % 100 == 0:  # loss.item() 获取损失的标量值，current 表示当前处理的数据量  loss, current = loss.item(), (batch + 1) \* len(X)  print(f"误差: {loss:>7f} [{current:>5d}/{size:>5d}]") |

**定义验证函数**

用测试数据集进行验证

|  |
| --- |
| Python def test(dataloader, model, loss\_fn):  # 测试数据集的总大小  size = len(dataloader.dataset)  # 总批次  num\_batches = len(dataloader)  # 将模型设置为评估模式  model.eval()  # test\_loss测试总损失，correct预测正确总次数  test\_loss, correct = 0, 0  # 禁用梯度计算，提高推理速度并降低内存损耗  with torch.no\_grad():  for X, y in dataloader:  X, y = X.to(device), y.to(device)  pred = model(X)  test\_loss += loss\_fn(pred, y).item()  correct += (pred.argmax(1) == y).type(torch.float).sum().item()  test\_loss /= num\_batches # 每批次平均损失  correct /= size # 预测正确概率  print(f"Test Error: \n Accuracy: {(100\*correct):>0.1f}%, Avg loss: {test\_loss:>8f} \n") |

**开始训练**

|  |
| --- |
| Python epochs = 5 # 定义训练的周期 for t in range(epochs):  print(f"当前周期 {t+1}\n-------------------------------")  # 开始训练  train(train\_dataloader, model, loss\_fn, optimizer)  # 测试模型  test(test\_dataloader, model, loss\_fn) print("完成训练!") |

7.  **保存模型、使用已有模型**

|  |
| --- |
| Python # 序列化内部状态字典（包含模型参数），再保存到指定路径 torch.save(model.state\_dict(), "model.pth")  # 加载已有模型 model = NeuralNetwork().to(device) model.load\_state.dict(torch.load("model.pth"))  # 使用已有模型 classes = [  "Tshirt/top",  "Trouser",  "Pullover",  "Dress",  "Coat",  "Sandal",  "Shirt",  "Sneaker",  "Bag",  "Ankle boot", ] # 模型设置为评估模式 model.eval() x, y = test\_data[0][0], test\_data[0][1] with torch.no\_grad():  x = x.to(device)  pred = model(x)  predicted, actual = classes[pred[0].argmax(0)], classes[y]  print(f'预测结果: "*{*predicted*}*", 真实结果: "*{*actual*}*"') |

**概念**

**什么是张量Tensor**

* **标量Scalar**：标量是零阶张量，只有一个值。例如，图像中某个像素的亮度值可以用标量表示。表示亮度值，表示像素位置
* **向量Vector**：向量是一阶张量，是一个数值的有序集合，即**一维数组**。例如，一个像素点的RGB值可以用一个向量表示

，R、G、B分别对应一、二、三列，一个红色像素标识为

* 即红色通道的值
* 即绿色通道的值
* 即蓝色通道的值
* **矩阵Matrix**：矩阵是二阶张量，可以表示一张灰色图像，每个像素对应一个像素的亮度值：
* **三阶张量Third-order Tensor**：彩色图像可以用一个三阶张量表示，包含宽度、高度和颜色通道三个维度。H表示高度像素个数，W表示宽度像素个数：

，即

这个三阶张量的尺寸是

* 即第一行第一个像素的红色通道的值
* 即第一行第一个像素的绿色通道的值
* 即第一行第一个像素的蓝色通道的值
* 即第一行第二个像素的红色通道的值

**pytorch中的张量**

张量是一种特殊的数据结构，与数组和矩阵相似。在pytorch中，我们使用张量来编码**模型的输入和输出，以及模型的参数**。

张量类似于**NumPy的ndarrays**，不同之处在于张量可以在GPU或其他硬件加速器上运行。

|  |
| --- |
| Python import numpy as np import torch # 将子面量转换为 tendor data = [[1,2],[3,4]] x\_data = torch.tendor(data)  # 将 numpy 数组转换为 tendor np\_array = np.array(data) x\_np = torch.from\_numpy(np\_array)  # 根据另一个tensor生成一个所有元素都为1的tensor，保留tensor的属性(shape, datatype) x\_ones = torch.ones\_like(x\_data) # 根据另一个tensor随机生成一个新的tensor，保留shape，覆盖另一个tensor的datatype属性 x\_rand = torch.rand\_like(x\_data, dtype=torch.float)  shape = (2,3,) # 括号可省略，datatype默认为torch.float print("random tensor: ", torch.rand(shape)) print("ones tensor: ", torch.ones(shape)) print("zeros tensor: ", torch.zeros(shape))  tensor = torch.rand(3, 4) print("tensor: ", tensor) print("shape: ", tensor.shape) print("datatype: ", tensor.dtype) print("device: ", tensor.device)  # 将tensor移动到GPU上处理 **if** **torch.cuda.is\_available():**  **tensor** **=** **tensor.to(**"cuda"**)**   tensor = torch.rand(4,4) print("tensor", tensor) print("第一行", tensor[0]) print("第一列", tensor[:, 0]) print("最后一列", tensor[..., -1]) # :等同与...  # 合并tensor，不会增加维度，dim指定合并的维度索引，默认0 t1 = torch.cat([tensor, tensor], dim=1) print(t1) # 沿着新维度堆叠，会增加维度 a = torch.stack((tensor, tensor), dim=1)  # 查看tensor尺寸 tensor = torch.tensor([[1,2],[1,2]]) print("size: ", tensor.size()) # torch.size([2, 2]) |