PyTorch DL Demo

PyTorch 编写深度学习算法模型的时候，其代码结构基本和 ML 的代码结构一致；都分为模型训练、模型评估、模型推理应用等各个阶段,：

▼ 模型训练流程 纯文本

```
1. 数据加载 + 数据转换
   ## 在深度学习中一般不太会做专门的特征工程，所以会直接在数据加载的时候直接对数据进行必要的处理转换
   ### 比如：图像的大小缩放、文本的分词转换等
2. 模型创建(模型结构 + 损失函数 + 优化器)
   ## 深度学习中，当使用原生的PyTorch框架进行编写的时候，需要构建以下内容：
   #### 当前任务确定的模型结构 + 当前模型对应的初始参数等信息；
   #### 当前任务对应的损失函数(eg: 交叉熵损失函数)
   #### 当前任务决定采用的优化器(eg: SGD)
   ## PS: 这些内容在机器学习中一般是以对象的参数存在的；
3. 模型训练 + 模型评估 + 模型持久化
   ## 编写数据迭代训练的代码，对数据进行划分后，按批次进行模型训练(前向 + 反向)；
   ## 由于模型训练是迭代执行的，所以一般在训练中途就会进行模型评估，
   ### 并不会等到模型训练完成(一般训练完后也可以增加一次评估操作)；
4. 模型持久化
   ## PS: 为了防止训练过程中的训练中断导致模型参数丢失问题，一般情况下在训练过程中也会做持久化保存；
```

▼ 模型推理部署流程 纯文本

```
1. 模型恢复(一般和模型持久化对应)
2. 数据转换处理：对待预测数据进行和训练相同的数据转换操作
3. 模型预测
4. 后处理：基于模型预测结果进行转换构造要求的输出格式数据

PS1:
#### 其中模型恢复仅需要恢复一次，其它操作需要针对每个待预测样本进行处理转换，
#### 故一般建议使用面向对象的方式来编写程序。
```

PS2:
最终部署模型的时候，一般是对外提供一个类似HTTP的接口
实际上和本地推理测试的逻辑相同，仅额外在外围增加一个接收HTTP请求参数以及处理结果返回的框架代码；

一、模型训练

▼ import 导包 Python

```python
import os
import warnings

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn import metrics
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split

warnings.filterwarnings('ignore')
```

1.1 数据准备

功能：从数据源加载数据到内存中；

PS: 当前准备构造一个双圆分类数据，内部是一个类别，外部是另外一个类别数据；

▼ 加载数据 Python

```python
# 1. 加载数据
X, Y = make_circles(
    n_samples=1000,  # 样本数目
    noise=0.1,  # 噪声样本比例
    factor=0.2,  # 内圈直径是外圈直径的factor倍
    random_state=24  # 随机数种子
)
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=24)
print(f"训练数据shape形状为：{type(x_train)} - {x_train.shape} -- {type(y_train)} - {y_train.shape}")
print(f"评估数据shape形状为：{type(x_test)} - {x_test.shape} -- {type(y_test)} - {y_test.shape}")
print(f"类别取值：{np.unique(y_train)} - {np.bincount(y_train)} -- {np.unique(y_test)} - {np.bincount(y_test)}")
```

1.2 模型创建

功能：需要创建网络结构、选择当前任务对应的损失函数以及优化器等信息；

1.2.1 模型定义

▼ 分类算法模型结构的定义 Python

```python
class ClassifyNetwork(nn.Module):
    def __init__(self, in_features: int, num_classes: int):
        """
        分类模型
        :param in_features: 分类模型输入的原始特征向量数目
        :param num_classes: 分类模型对应的类别数目
        """
        super().__init__()

        self.in_features = in_features
        self.num_classes = num_classes

        # 1. 特征提取
        self.features = nn.Sequential(
            nn.Linear(self.in_features, 8),
            nn.ReLU(),
            nn.Linear(8, 8),
            nn.ReLU()
        )

        # 2. 决策输出
        self.classify = nn.Linear(8, self.num_classes)

    def forward(self, x):
        """
        前向过程
            NOTE:
                bs: 样本数目
                in_features: 每个样本的向量维度大小
                置信度：模型损失计算前的数据对象
        :param x: 输入的原始特征向量，FloatTensor格式，shape形状为: [bs, in_features]
        :return: 前向结果，训练时候一般为置信度值，推理的时候可以直接返回预测结果，FloatTens
or格式，shape形状为: [bs, num_classes]
        """
        # 1. 样本特征向量提取 [bs,in_features] --> [bs,8]
        features = self.features(x)

        # 2. 基于提取的特征向量进行分类决策 [bs,8] --> [bs,num_classes]
        score = self.classify(features)

        # 3. 基于不同的结果返回不同要求的数据
        if self.training:
```

```
            return score
        return torch.softmax(score, dim=1)
```

1.2.2 创建对象

对象创建 Python

```
# 2. 模型创建
# 模型结构创建
net = ClassifyNetwork(in_features=x_train.shape[1], num_classes=len(np.unique(y_t
rain)))
# 损失函数创建
loss_fn = nn.CrossEntropyLoss()
# 优化器创建
opt = optim.SGD(params=net.parameters(), lr=0.01)
```

1.3 模型训练 + 模型评估 + 模型持久化

功能：迭代数据拼装批次数据，进行批次的模型训练、模型评估及持久化；

模型训练 + 评估 + 持久化 Python

```
# 3. 模型训练+模型评估+模型持久化
total_epoch = 100
batch_size = 8
test_batch_size = batch_size * 2
total_train_batch = len(x_train) // batch_size
total_test_batch = len(x_test) // test_batch_size + (1 if len(x_test) % test_batc
h_size != 0 else 0)
model_output_dir = "./output/02/models"
os.makedirs(model_output_dir, exist_ok=True)
for epoch in range(total_epoch):
    # 训练
    net.train()
    train_rnd_indexes = np.random.permutation(len(x_train))
    for batch_idx in range(total_train_batch):
        # 获取当前批次的数据x + y
        si = batch_size * batch_idx
        ei = si + batch_size
        train_batch_indexes = train_rnd_indexes[si: ei]
        batch_x_train = torch.tensor(x_train[train_batch_indexes], dtype=torch.fl
oat32)
        batch_y_train = torch.tensor(y_train[train_batch_indexes], dtype=torch.in
t64)

        # 前向过程
        score = net(batch_x_train)  # [bs,num_classes]
        loss = loss_fn(score, batch_y_train)
```

```
        # 反向过程
        opt.zero_grad()  # 重置当前优化器对应的所有参数的梯度为0
        loss.backward()  # 计算和当前损失相同的所有参数的梯度值
        opt.step()  # 参数更新

        print(f"Train Epoch {epoch}/{total_epoch} Batch {batch_idx}/{total_train_
batch} Loss:{loss.item():.3f}")

    # 评估
    net.eval()
    with torch.no_grad():
        test_indexes = list(range(len(x_test)))
        for batch_idx in range(total_test_batch):
            # 获取当前批次的数据x + y
            si = test_batch_size * batch_idx
            ei = si + test_batch_size
            test_batch_indexes = test_indexes[si: ei]
            batch_x_test = torch.tensor(x_test[test_batch_indexes], dtype=torch.f
loat32)
            batch_y_test = torch.tensor(y_test[test_batch_indexes], dtype=torch.i
nt64)

            # 前向过程
            score = net(batch_x_test)  # [bs,num_classes]
            loss = loss_fn(score, batch_y_test)

            # 效果评估
            pred_idx = torch.argmax(score, dim=1) # 获取预测的类别id
            acc = metrics.accuracy_score(batch_y_test.numpy(), pred_idx.numpy())

            print(f"Test Epoch {epoch}/{total_epoch} Batch {batch_idx}/{total_tes
t_batch} "
                  f"Batch-number:{batch_x_test.shape[0]} Loss:{loss.item():.3f} A
ccuracy:{acc:.3f}")

    # 模型持久化
    torch.save(
        {
            'net': net, # 模型对象(参数 + 结构)
            'net_param': net.state_dict(),  # 模型网络对应的所有参数
            'epoch': epoch
        },
        os.path.join(model_output_dir, f"{epoch:06d}.pkl")
    )
```

二、模型推理

PS: 仅基于 joblib 持久化的模型进行推理预测

▼ import 导包 Python

```python
import joblib
```

2.1 模型恢复

功能：从持久化好的文件中恢复好模型对象，包括参数以及结构(执行逻辑)；

▼ 模型恢复 Python

```python
# 1. 模型恢复
obj = torch.load("./output/02/models/000099.pkl", map_location='cpu')
net = obj['net']
net.eval()  # 进入推理阶段
print(f"模型恢复完成: \n{net}\n\n")
```

2.2 数据转换

功能：使用和训练数据相同的数据处理流程，对待预测数据进行转换处理(如果训练时候没有做数据转换操作，那么推理的时候也不需要进行)；

▼ 数据转换 Python

```python
# 2. 数据转换
x = [
    [0.05, -0.01],
    [0.1, 0.3],
    [-0.4, 0.2],
    [1.0, 1.2],
    [0.0, 0.75],
    [0.0, -1.2]
]
x = torch.tensor(x, dtype=torch.float32)
```

2.3 模型预测

功能：调用模型对应的预测方法，获取预测结果；根据不同的需要，可能需要调用不同的预测方法，比如：返回预测类别 id、返回预测属于各个类别的概率等等；

▼ 模型预测 Python

```python
# 3. 模型预测
y_pred_proba = net(x)
print(f"获取预测概率对象为: {type(y_pred_proba)} - {y_pred_proba.shape}")
```

2.4 结果转换

功能：结合调用方的要求，对模型的预测结果进行转换输出；

结果转换 Python

```python
# 4. 结果拼接
with torch.no_grad():
    y_pred_proba = y_pred_proba.numpy()
    y_pred_idx_per_sample = np.argmax(y_pred_proba, axis=1).tolist()
    y_pred_proba_per_sample = y_pred_proba[range(len(y_pred_idx_per_sample)), y_pred_idx_per_sample].round(3).tolist()
    result = list(map(lambda t: {'id': t[0], 'proba': t[1]}, zip(y_pred_idx_per_sample, y_pred_proba_per_sample)))
    print(result)
# [{'id': 1, 'proba': 1.0}, {'id': 1, 'proba': 0.9800000190734863}, {'id': 1, 'proba': 0.9110000133514404}, {'id': 0, 'proba': 1.0}, {'id': 0, 'proba': 0.9599999785423279}, {'id': 0, 'proba': 1.0}]
```