Note Information

Author : AbbasXu
Date : 2022-08-18
Title : pytorch基础
Keywords : #pytroch #模型搭建 #训练 #损失函数

基本配置

常用的包

- 表格处理-pandas
- 图像处理-cv2
- 可视化-pyecharts、matplotlib、seaborn
- 下游分析、指标计算-scikit-learn。

超参数设置

- batchsize
- 初始学习率
- 训练次数
- GPU配置

GPU配置

GPU的设置有两种常见的方式

```
# 方案一：使用os.environ，这种情况如果使用GPU不需要设置
os.environ['CUDA_VISIBLE_DEVICES'] = '0,1'

# 方案二：使用“device”，后续对要使用GPU的变量用.to(device)即可
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
```

数据读入

数据输入的过程可以定义自己的Dataset类来实现快速读取，，定义的类需要继承PyTorch自身的Dataset类。主要包含三个函数：

- `__init__`: 用于向类中传入外部参数，同时定义样本集
- `__getitem__`: 用于逐个读取样本集合中的元素，可以进行一定的变换，并将返回训练/验证所需的数据
- `__len__`: 用于返回数据集的样本数

构建好Dataset后，就可以使用DataLoader来按批次读入数据了

其中:

- batch_size：样本是按“批”读入的，batch_size就是每次读入的样本数
- num_workers：有多少个进程用于读取数据
- shuffle：是否将读入的数据打乱
- drop_last：对于样本最后一部分没有达到批次数的样本，使其不再参与训练

模型构建

神经网络构建

Module 类是 nn 模块里提供的一个模型构造类，是所有神经网网络模块的基类，我们可以继承它来定义我们想要的模型。下面继承 Module 类构造多层感知机。这里定义的 MLP 类重载了 Module 类的 init 函数和 forward 函数。它们分别用于创建模型参数和定义前向计算。前向计算也即正向传播。

```
import torch
from torch import nn

class MLP(nn.Module):
    # 声明带有模型参数的层，这里声明了两个全连接层
    def __init__(self, **kwargs):
        # 调用MLP父类Module的构造函数来进行必要的初始化。这样在构造实例时还可以
指定其他函数
        # 参数，如“模型参数的访问、初始化和共享”一节将介绍的模型参数params
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Linear(784, 256) # 隐藏层
        self.act = nn.ReLU()
        self.output = nn.Linear(256, 10)  # 输出层
```

```
    # 定义模型的前向计算，即如何根据输入x计算返回所需要的模型输出
    def forward(self, x):
        a = self.act(self.hidden(x))
        return self.output(a)
```

神经网络常见的层

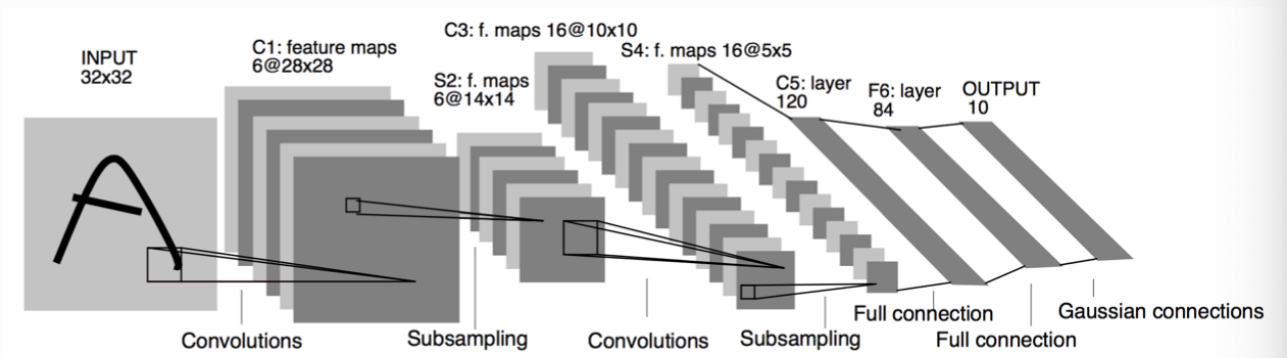
- 二维卷积层
  - 宽高
  - 填充（padding）
  - 步幅（stride）
- 池化层
  - 最大\平均

模型示例

一个神经网络的典型训练过程如下

- 定义包含一些可学习参数(或者叫权重）的神经网络
- 在输入数据集上迭代
- 通过网络处理输入
- 计算 loss (输出和正确答案的距离)
- 将梯度反向传播给网络的参数
- 更新网络的权重，一般使用一个简单的规则：weight = weight - learning_rate * gradient

LeNet



LeNet网络结构

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

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 输入图像channel：1；输出channel：6；5x5卷积核
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # 2x2 Max pooling
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # 如果是方阵,则可以只使用一个数字进行定义
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # 除去批处理维度的其他所有维度
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

注：`torch.nn`只支持小批量处理 (mini-batches) 。整个`torch.nn`包只支持小批量样本的输入，不支持单个样本的输入。比如，`nn.Conv2d`接受一个4维的张量，即`nSamples x`

`nChannels x Height x Width` 如果是一个单独的样本，只需要使用`input.unsqueeze(0)` 来添加一个“假的”批大小维度。

模型初始化

torch.nn.init
计算增益：
nonlinearity	gain
Linear/Identity	1
Conv{1,2,3}D	1
Sigmod	1
Tanh	5月3日
ReLU	sqrt(2)
Leaky Relu	sqrt(2/1+neg_slop^2)
对`conv`进行kaiming初始化，对linear进行常数初始化.....最后将这些函数封装成`initialize_weights()`

损失函数

它是数据输入到模型当中，产生的结果与真实标签的评价指标
仅记录部分常用的损失函数

二分类交叉熵损失函数

功能

计算二分类任务时的交叉熵（Cross Entropy）函数。在二分类中，label是{0,1}。对于进入交叉熵函数的input为概率分布的形式。一般来说，input为sigmoid激活层的输出，或者softmax的输出。

函数

`torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')`

主要参数

- weight:每个类别的loss设置权值

- size_average:数据为bool，为True时，返回的loss为平均值；为False时，返回的各样本的loss之和。
- reduce:数据类型为bool，为True时，loss的返回是标量。

计算公式

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'} \\ \text{sum}(L), & \text{if reduction} = \text{'sum'} \end{cases}$$

交叉熵损失函数

功能

计算交叉熵函数

函数

```
torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')
```

主要参数

- weight:weight:每个类别的loss设置权值。
- size_average:数据为bool，为True时，返回的loss为平均值；为False时，返回的各样本的loss之和。
- ignore_index:忽略某个类的损失函数。
- reduce:数据类型为bool，为True时，loss的返回是标量。

计算公式

$$\text{loss}(x,\ \text{class}\ ) = -\log\left(\frac{\exp(x[\ \text{class}\ ])}{\sum_j \exp(x[j])}\right) = -x[\ \text{class}\ ] + \log\left(\sum_j \exp(x[j])\right)$$

MSE损失函数

功能

计算输出y和真实标签target之差的平方。
和L1Loss一样，MSELoss损失函数中，reduction参数决定了计算模式。有三种计算模式可选：none：逐个元素计算。 sum：所有元素求和，返回标量。默认计算方式是求平均。

函数

`torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')`

主要参数

无

计算公式

$$l_n = (x_n - y_n)^2$$

KL散度

功能

计算KL散度，也就是计算相对熵。用于连续分布的距离度量，并且对离散采用的连续输出空间分布进行回归通常很有用

函数

`torch.nn.KLDivLoss(size_average=None, reduce=None, reduction='mean', log_target=False)`

主要参数

- reduction：计算模式，可为 none/sum/mean/batchmean。
  - none：逐个元素计算。
  - sum：所有元素求和，返回标量。
  - mean：加权平均，返回标量。
  - batchmean：batchsize 维度求平均值。

计算公式

$$D_{\mathrm{KL}}(P, Q) = \mathrm{E}_{X\sim P}\left[\log \frac{P(X)}{Q(X)}\right] = \mathrm{E}_{X\sim P}[\log P(X) - \log Q(X)]$$
$$= \sum_{i=1}^{n} P\left(x_i\right)\left(\log P\left(x_i\right) - \log Q\left(x_i\right)\right)$$

余弦相似度

功能

对两个向量做余弦相似度

函数

`torch.nn.CosineEmbeddingLoss(margin=0.0, size_average=None, reduce=None, reduction='mean')`

主要参数

- reduction：计算模式，可为 none/sum/mean。
- margin：可取值[-1,1]，推荐为[0,0.5]。

计算公式

$$\operatorname{loss}(x, y)=\begin{cases}1-\cos\left(x_1, x_2\right), & \text{if } y=1 \\ \max\left\{0, \cos\left(x_1, x_2\right)-\text{ margin }\right\} & \text{if } y=-1\end{cases}$$

训练与评估

下列操作二选一即可：

```
model.train()   # 训练状态
model.eval()   # 验证/测试状态
```

整个训练过程如下所示：

1. 进入循环
2. 数据放入GPU中
3. 开始用当前批次数据做训练时，应当先将优化器的梯度置零
4. 之后将data送入模型中训练
5. 根据预先定义的criterion计算损失函数
6. 将loss反向传播回网络
7. 使用优化器更新模型参数

Pytorch优化器

- `torch.optim`
  - torch.optim.ASGD

    - torch.optim.Adadelta
    - torch.optim.Adagrad
    - torch.optim.Adam
    - torch.optim.AdamW
    - torch.optim.Adamax
    - torch.optim.LBFGS
    - torch.optim.RMSprop
    - torch.optim.Rprop
    - torch.optim.SGD
    - torch.optim.SparseAdam

而以上这些优化算法均继承于`Optimizer`，其含有三个属性

- `defaults`：存储的是优化器的超参数
- `state`：参数的缓存
- `param_groups`：管理的参数组，是一个list，其中每个元素是一个字典，顺序是params，lr，momentum，dampening，weight_decay，nesterov
  同时含有以下几种方法：
- `zero_grad()`：清空所管理参数的梯度，PyTorch的特性是张量的梯度不自动清零，因此每次反向传播后都需要清空梯度。
- `step()`：执行一步梯度更新，参数更新。
- `add_param_group()`：添加参数组
- `load_state_dict()`：加载状态参数字典，可以用来进行模型的断点续训练，继续上次的参数进行训练
- `state_dict()`：获取优化器当前状态信息字典