

Pytorch

pytorch与tensorflow的区别

pytorch的特点

- 动态图模式
- 支持GPU加速

tensorflow的特点

- 静态图模式
- 支持GPU加速

datasets

```
from torchvision import datasets
```

使用

- `.dataset`
 - `root` 数据集根目录
 - `train` 训练数据集
 - `download` 是否下载
 - `transform` 数据转换

Dataset

在 `torch.utils.data` 模块中定义

继承 `torch.utils.data.Dataset` 类

- `__len__()` 返回数据集长度
- `__getitem__(idx)` 返回索引 `idx` 的数据

DataLoader

在 `torch.utils.data` 模块中定义 `Dataset` 的加载器

- `batch_size` 每个batch的大小
- `shuffle=True` 是否在每个epoch中打乱数据
- `num_workers` 使用的进程数/线程数

参数	说明
<code>dataset</code>	要加载的Dataset对象
<code>batch_size</code>	每个batch的大小
<code>shuffle</code>	是否在每个epoch中打乱数据
<code>num_workers</code>	使用的进程数/线程数
<code>drop_last</code>	是否丢弃最后一个batch
<code>pin_memory</code>	是否将数据pin到GPU内存

Transform

torchvision

函数	描述
ToTensor()	PIL 转换为 Tensor
Normalize(mean, std)	归一化
RandomCrop(size)	随机裁剪
RandomHorizontalFlip()	随机水平翻转

神经网络

torch.nn.Function
 $\text{ReLU}(x) = \max(0, x)$

- Relu
 - 输出范围 $[0, +\infty)$
 - 与 Sigmoid 类似
 - 不可导

$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

- Tanh
 - 输出范围 $(-1, 1)$
 - 与 Sigmoid 类似
 - 不可导

$\sigma(x) = \frac{1}{1 + e^{-x}}$

- Sigmoid
 - 输出范围 $(0, 1)$
 - 与 Tanh 类似
 - 不可导

$\text{GELU}(x) = x \cdot \Phi(x)$

- GELU
 - ReLU 的改进
 - $\Phi(x)$ 是标准正态分布的累积分布函数
 - Transformer 使用

$\text{Swish}(x) = x \cdot \sigma(\beta x)$

- Swish
 - σ 是 Sigmoid, β 是缩放因子
 - 比 ReLU 更平滑
 - Transformer 使用

```
class Swish(nn.Module):
    def __init__(self, beta = 1.0):
        super().__init__()
        self.beta = beta

    def forward(self, x):
        return x * torch.sigmoid(self.beta * x)
```

$\text{PreLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{if } x < 0 \end{cases}$

- **PreLU**
 - 可微分
 - 在CNN中表现良好

损失函数

在 `torch.nn` 中

1. 均方误差 **MSE**

- 计算

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- 计算
- `nn.MSELoss()`

2. 交叉熵 **Cross-Entropy**

- 计算

$$\text{CE} = -\sum_{i=1}^n y_i \log(\hat{y}_i)$$

- 计算
- `nn.CrossEntropyLoss()`

优化器

在 `torch.optim` 中

初始化

1. 初始化 `nn.Linear` 或 `nn.Conv2d`
2. 初始化 `optim.SGD` 或 `optim.Adam`
3. 训练循环
 - `optimizer.zero_grad()` 清零梯度
 - `loss.backward()` 反向传播
 - `optimizer.step()` 更新参数

SGD 初始化

- 初始化
 - `model.parameters()` 获取参数
 - `lr` 学习率
 - `momentum` 动量
 - `dampening` 阻尼系数 0

- `weight_decay` L2 0

$$v_t = \beta v_{t-1} + (1-\beta) \frac{\partial J}{\partial w} \quad w = w - \alpha v_t$$

Adam

- 初始化
- `model.parameters()`
 - `lr`
 - `betas`
 - `β_1`
 - `β_2`
 - `eps` 0
 - `weight_decay` L2

1. 初始化

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

2. 更新

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

3. 更新

$$\theta_{t+1} = \theta_t - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$$

RMSprop

- 初始化
- `model.parameters()`
 - `lr`
 - `alpha`
 - `eps` 0
 - `weight_decay` L2

$$v_t = \beta v_{t-1} + (1-\beta) \left(\frac{\partial J}{\partial w} \right)^2 \quad w = w - \alpha \frac{\frac{\partial J}{\partial w}}{\sqrt{v_t + \epsilon}}$$

初始化

初始化

```
from torch.optim import lr_scheduler
```

StepLR

参数	说明
<code>optimizer</code>	优化器
<code>step_size</code>	每隔多少epoch
<code>gamma</code>	学习率衰减系数0.1

last_epoch	현재 epoch
------------	----------

`scheduler.step()` 현재 scheduler

ReduceLROnPlateau (현재)

key	value
optimizer	optimizer
mode	'min' 또는 'max'
factor	factor
patience	epoch
threshold	threshold
threshold_mode	'rel' 또는 'abs'
cooldown	epoch
min_lr	min_lr
verbose	verbose

`scheduler.step(val_loss)` 현재

CosineAnnealingLR (현재)

key	value
optimizer	optimizer
T_max	epoch
eta_min	eta_min
last_epoch	현재 epoch

`scheduler.step()` 현재

Conv

`torch.nn` Convolutional layer

`$(in_channels,H,W)$`

- Convolutional layer
- Convolutional layer

Conv

- `in_channels` RGB 3
- `out_channels` Convolutional layer
- `kernel_size` Convolutional layer
- `stride` Convolutional layer
- `padding` Convolutional layer

- `bias` 是否
- `dilation` 膨胀率
- `groups` 组数

计算公式 $H_{out} = \frac{H_{in} + 2 \times padding - dilation \times (kernel_size - 1) - 1}{stride}$

Transposed Conv

`torch.nn.ConvTranspose2d`

参数

输出通道数

步长

1. 输入通道数 `s-1` 是否 0
2. 输入通道数 `k-p-1` 是否 0
3. 是否
4. 是否

参数

- `in_channels` 输入通道数 RGB 3
- `out_channels` 输出通道数
- `kernel_size` 卷积核大小
- `stride` 步长 0
- `padding` 填充
- `bias` 是否
- `out_padding` 输出填充



image-20250805130358084



image-20250805130421265

计算公式 $H_{out} = (H_{in} - 1) \times stride - 2 \times padding + kernel_size + output_padding$

Pool

`torch.nn` 池化层

参数

- `nn.MaxPool2d` 最大池化
- `nn.AvgPool2d` 平均池化
- `nn.AdaptiveMaxPool2d` 自适应最大池化

参数

- `kernel_size` 卷积核大小
- `stride` 步长 `kernel_size`
- `padding` 填充
- `dilation` 膨胀率

计算公式
$$H_{out} = \frac{H_{in} + 2 \times padding - dilation \times (kernel_size - 1) - 1}{stride}$$

Linear

`torch.nn` 线性层 $y = x \cdot W^T + b$

- `x`: 输入特征 `in_features`

参数

- `in_feature`: 输入特征数
- `out_feature`: 输出特征数
- `bias`: 偏置

线性层用于将输入特征映射到输出特征。

Flatten

`torch.nn` 展平层

- 将输入张量展平为一维向量 (batch_size, channels, height, width)
- 将输入张量展平为一维向量 (batch_size, channels * height * width)

Normalization

`torch.nn`

Batch Normalization

批归一化

类 `nn.BatchNorm1d`

- 参数

类 `nn.BatchNorm2d` 批归一化 (Batch Normalization)

- 参数 `[Batch_size, Channels, Height, Width]`
- 参数 `Batch` 归一化 (Batch Normalization) **Batch**
- 参数 `Channels`

参数

- 参数
- 参数 `batch` 归一化 (Batch Normalization)
- 参数

Layer Normalization

层归一化

类 `nn.LayerNorm`

参数

- 参数
- 参数 **RNN/Transformer**

Group Normalization

nn.GroupNorm

nn.GroupNorm 1D, 2D, 3D, 4D, 5D tensors

args

- num_groups 1D tensors
- num_channels 1D tensors
- affine 1D tensors

Upsample

nn.Upsample

nn.Upsample 1D, 2D, 3D, 4D, 5D tensors

args

args	type	default
size	int tuple	1D tensors (256, 256) or None or scale_factor
scale_factor	float tuple	1D tensors 2D tensors (2, 3) or 2D tensors 3D
mode	str	1D tensors 'nearest' or 'bilinear' or 'bicubic' or 'nearest'
align_corners	bool	1D tensors bilinear/bicubic or False

Dropout

torch.nn.Dropout 0D tensors

.train() or .eval() 0D tensors

Interpolate

torch.nn.functional.interpolate

1D, 2D, 3D, 4D, 5D tensors / 1D, 2D, 3D, 4D, 5D tensors

args

args	type	default
size	(target_h, target_w)	1D tensors
scale_factor	float (float, float)	1D tensors 2.0 or 2D
mode	'nearest', 'bilinear', 'bicubic', 'area'	1D tensors
align_corners	bool	1D tensors True or 1D tensors

kwargs

1D, 2D, 3D, 4D, 5D tensors / 1D, 2D, 3D, 4D, 5D tensors

- torch.utils.checkpoint
 - 1D

1. `checkpoint` `fn`
2. `checkpoint` `args`

• `fn`

- `fn` `args`
- `**args` `kwargs`

`fn`

```
import torch
import torch.nn as nn
from torch.utils.checkpoint import checkpoint

class BigModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(1000, 1000)
        self.layer2 = nn.Linear(1000, 1000)
        self.layer3 = nn.Linear(1000, 1000)

    def forward(self, x):
        # checkpoint
        x = checkpoint(self.layer1, x) # checkpoint self.layer1(x)
        x = checkpoint(self.layer2, x)
        x = self.layer3(x) # checkpoint
        return x

## fn
model = BigModel()
inputs = torch.randn(32, 1000)
outputs = model(inputs)
loss = outputs.sum()
loss.backward() # checkpoint layer1 layer2
```

- `torch.utils.checkpoint_sequential` `Sequential`

• `fn`

- `functions` `Sequential`
- `input` `kwargs`
- `segments` `Sequential`
 - `nn.Sequential` `segments` `checkpoint_sequential`
 - `segments` `checkpoint_sequential`

`fn`

```
from torch.utils.checkpoint import checkpoint_sequential

model = nn.Sequential(
    nn.Linear(10, 10),
    nn.ReLU(),
    nn.Linear(10, 10),
    nn.ReLU(),
```

```

        nn.Linear(10, 10)
    )

    input_tensor = torch.randn(2, 10, requires_grad=True)
    ## 验证 chunk 2 是否 forward 到 checkpoint
    output = checkpoint_sequential(model, chunks=2, input=input_tensor)

```

验证

验证

```

## 验证
accumulation_steps = 4
## 验证
optimizer.zero_grad()

for i, (inputs, labels) in enumerate(data_loader):
    outputs = model(inputs)
    # 验证
    loss = criterion(outputs, labels)
    # 验证
    loss = loss / accumulation_steps
    # 验证
    loss.backward() # 验证

    if (i + 1) % accumulation_steps == 0:
        # 验证
        optimizer.step()
        # 验证
        optimizer.zero_grad()

```

验证

- $\text{batch_size} = 36$

$\nabla L_{\text{true}} = \frac{1}{32} \sum_{i=1}^{32} \nabla \text{loss}_i$

- $\text{mini_batch} = 8$

$\nabla L_{\text{accu}} = \sum_{j=1}^4 \left(\frac{1}{4} \cdot \frac{1}{8} \sum_{k=1}^8 \nabla \text{loss}_{jk} \right) = \frac{1}{32} \sum_{i=1}^{32} \nabla \text{loss}_i$

验证

`torch.nn.utils` 验证

验证

验证 `clip_grad_value_`

- 验证
- `torch.nn.utils.clip_grad_value_`
 - `model.parameters()` 验证

- clip_value 0.000000

0.000000 clip_grad_norm_ 0

- 0.000000 L2 0.000000
- torch.nn.utils.clip_grad_norm_
 - model.parameters() 0.000000
 - max_norm 0.000000 L2 0.000000

0.000000

pytorch 0.000000

```
class EarlyStopping:
    def __init__(self, patience=3):
        self.patience = patience
        self.best_loss = float('inf')
        self.counter = 0
    # 0.000000
    def __call__(self, val_loss):
        if val_loss < self.best_loss:
            self.best_loss = val_loss
            self.counter = 0
            return False
        else:
            self.counter += 1
            return self.counter >= self.patience
```

0.000000

0.00

0.000000

0.000000 int8 0.000000 float32

- torch.quantization 0.00 from torch.quantization import quantize_dynamic
 - model 0.000000
 - {nn.Linear, nn.LSTM} 0.000000
 - dtype 0.000000

0.000000 int8

- 0.00
 1. 0.000000
 2. 0.000000 fuse_modules 0.00
 - torch.quantization.fuse_modules
 3. 0.000000 qconfig 0.00
 - torch.quantization.get_default_qconfig
 - fbgemm 0.0x86 qnnpack 0.0ARM

```
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
```

4. `prepare`

- `torch.quantization.prepare()`
 - `inplace` `bool`

5. `forward`

6. `convert`

- `torch.quantization.convert`

####

```
model = MyModel()
model.eval()

## 1. 融合
model_fused = torch.quantization.fuse_modules(model, [['conv', 'relu']])

## 2. qconfig
model_fused.qconfig = torch.quantization.get_default_qconfig('fbgemm')

## 3. 准备
torch.quantization.prepare(model_fused, inplace=True)

## 4. 校准
with torch.no_grad():
    for data, _ in calibration_loader:
        model_fused(data)

## 5. 转换
torch.quantization.convert(model_fused, inplace=True)
```

量化感知训练(QAT)“量化”`FakeQuantize` 模块

- 步骤
 - 1. 融合
 - `torch.quantization.fuse_modules`
 - 2. 设置 `qconfig` 为 QAT
 - `torch.quantization.get_default_qat_qconfig`
 - 3. 准备 `qat`
 - `torch.quantization.prepare_qat()`
 - 4. 训练 epoch
 - 5. 转换
 - `torch.quantization.convert()`

####

```

model.train()
model_fused = torch.quantization.fuse_modules(model, [['conv', 'relu']])

## 1. 加载 QAT 模型
model_fused.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')

## 2. 准备 QAT
torch.quantization.prepare_qat(model_fused, inplace=True)

## 3. 训练
for epoch in range(num_epochs):
    for data, target in train_loader:
        output = model_fused(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()

## 4. 模型评估
model_fused.eval()
torch.quantization.convert(model_fused, inplace=True)

```

剪枝

torch.nn.utils.prune

剪枝方法

- `prune.l1_unstructured` 对 **L1** 范数进行剪枝
 - 对 **L1** 范数进行剪枝
 - 剪枝后的权重为 **0**
- `prune.random_unstructured` 随机剪枝
 - `module` 模块名称
 - `name` 模块名称
 - `amount` 剪枝数量
 - `float` 剪枝比例
 - `int` 剪枝数量
- `prune.ln_structured` 对 **LN** 层进行剪枝
 - 剪枝
 - `amount` 剪枝数量
 - `float` 剪枝比例
 - `int` 剪枝数量
 - `n` 剪枝层数
 - `dim` 剪枝维度
 - `0` 通道
 - `1` 特征图
- `prune.remove` 移除剪枝后的模块
 - `module` 模块名称
 - `name` 模块名称

剪枝后的模型权重为 `mask=0` 表示剪枝后的权重

5/5

tensorboard

```
from torch.utils import tensorboard
```

- `tensorboard`, `tensorboard`

```
from torch.utils.tensorboard import SummaryWriter
## 创建 SummaryWriter 对象

writer = SummaryWriter('C:/study/pytorch/logs')

## 使用 SummaryWriter

writer.close()
```

1. □□□□

- ```
o writer.add_scalar()
 ■ tag 000000
 ■ 00 train/loss 0 eval/loss 0000000000
 ■ scalar_value 00000000000000y0
 ■ global_step 00000000000000x0
```

2. □□□□□□□□□□□□□□

- `writer.add_graph()`
  - `graph`
  - `graph_names`

3.

- `writer.add_hparams()`
  - `hparams`
  - `hparams_proto`

**tqdm**

1010101010

```
from tqdm import tqdm
```

tqdm()

- `trainloader = DataLoader(...)`
- `desc` `trainloader.desc`
- `total` `trainloader.total`
- `leave` `trainloader.leave`

- unit 每秒 it/s 每 batch/s
- ncols 每行

`trainer.set_postfix()` 设置 postfix

- `trainer.train_bar.set_postfix(loss=f"{loss.item():.4f}")`
- `trainer`

```
trainer.train_bar.set_postfix({
 'loss': f"{loss.item():.4f}",
 'acc': f"{acc.item():.2%}" # 准确率
})
```

保存模型

- `torch.save(model.state_dict(), './model_weights.pth')`

`torch.save(model.state_dict(), './model_weights.pth')`

- `model.load_state_dict(torch.load('./model_weights.pth'))`

`model.load_state_dict(torch.load('./model_weights.pth'))`

- `torch.save(model, path)`

`torch.save(model, path)`

- `model = torch.load(path)`

`model = torch.load(path)`

分布式

分布式

- `from torch.nn.parallel import DistributedDataParallel as DDP`
- `from torch.utils.data.distributed import DistributedSampler`
- `import torch.distributed as dist`
- `import torch.multiprocessing as mp`

分布式

1. 分布式

分布式 GPU

2. 分布式

分布式

3. `DDP` 分布式 `DistributedSampler` 分布式
4. `torch.multiprocessing.spawn` 分布式

分布式

- `loss.backward()` 反向传播 `.grad` 梯度

- `optimizer.step()` 更新参数
- `optimizer.zero_grad()` 清空梯度
- 清空梯度并更新参数
- `torch.no_grad()` 防止梯度反向传播
- `.requires_grad_` 设置是否需要梯度反向传播
- `loss.item()` 返回标量值 **Python**
- `optim.zero_grad()` 清空梯度
- `optimizer.step()` 更新参数
- `loss.backward()` 反向传播
- `self.training` 在 `nn.Module` 中为 `True` 表示训练模式 `False` 表示推理模式
- 数据预处理
- 数据增强
- `.permute()` 调整维度 `.reshape()` 调整形状