

Pytorch

pytorchとtensorflowの比較

pytorchの特徴

- ・ 複数GPUでの並列化
- ・ ネットワーク構造の再利用

tensorflowの特徴

- ・ GPU利用
- ・ ネットワーク構造の再利用

datasets

```
from torchvision import datasets
```

特徴

- ・ .__init__
 - root データディレクトリ
 - train データセット
 - download ダウンロード
 - transform データ変換

Dataset

torch.utils.data.Dataset

特徴 torch.utils.data.Dataset の特徴

- ・ __len__() データ点の数
- ・ __getitem__(idx) データ idx 番目のデータ

DataLoader

torch.utils.data.DataLoader Dataset の特徴

- ・ batch_size バッチの大きさbatch
- ・ shuffle=True 1epoch 毎のシャッフル
- ・ num_workers データ読み込み/処理の並行度

パラメータ	説明
dataset	データセットDataset
batch_size	バッチの大きさ
shuffle	1epoch毎のシャッフル
num_workers	データ読み込み/CPUの並行度
drop_last	最終バッチを削除する
pin_memory	GPUメモリにデータを転送する

Transform

torchvision 補助モジュール

モジュール	機能
ToTensor()	PIL フォーマット
Normalize(mean, std)	正規化
RandomCrop(size)	ランダムカット
RandomHorizontalFlip()	ランダムフリップ

活性化関数

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

- Relu

- $[0, +\infty)$
- Sigmoid
- Tanh

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

- Tanh

- $(-1, 1)$
- $[-1, 1]$
- $[0, 1]$

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

- Sigmoid

- $[0, 1]$
- $[0, 1]$
- $[0, 1]$

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

- GELU

- ReLU
- $\text{Phi}(x)$
- Transformer

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

- Swish

- $\sigma \cdot \text{Sigmoid}$
- **ReLU**
- Transformer

```

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
 - ပုဂ္ဂနည်သူများ
 - ပုဂ္ဂနည်CNNများ

ပုဂ္ဂနည်

◦ `torch.nn`

1. ပုဂ္ဂနည်MSE

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

2. ပုဂ္ဂနည်Cross-Entropy

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

ပုဂ္ဂနည်

◦ `torch.optim`

ပုဂ္ဂနည်ပေါ်ပေါ်လေ့လာမှု

1. ပုဂ္ဂနည် `nn.Linear` မှ `nn.Conv2d` ထဲ
2. ပုဂ္ဂနည် `optim.SGD` မှ `optim.Adam` ထဲ
3. ပုဂ္ဂနည်ပေါ်ပေါ်လေ့လာမှု
 - `optimizer.zero_grad()` ပုဂ္ဂနည်
 - `loss.backward()` ပုဂ္ဂနည်
 - `optimizer.step()` ပုဂ္ဂနည်

SGDပုဂ္ဂနည်ပေါ်ပေါ်လေ့လာမှု

- ပုဂ္ဂနည်
 - `model.parameters()`ပုဂ္ဂနည်
 - `lr`ပုဂ္ဂနည်
 - `momentum`ပုဂ္ဂနည်
 - `dampening`ပုဂ္ဂနည် 0

- weight_decay L2 ဆုတေသန 0

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

Adam အကြောင်း

- အကြောင်းအရာများအတွက် အသုတေသန အသုတေသန
- ပို့မှု
 - model.parameters() ပို့မှုများ
 - lr ပို့မှု
 - betas ပို့မှု
 - β_1 ပို့မှုများ
 - β_2 ပို့မှုများ
- eps ပို့မှုများ 0
- weight_decay L2 ဆုတေသန

1. အကြောင်းအရာများ

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

2. ပို့မှု

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

3. ပို့မှု

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

RMSprop

- အကြောင်းအရာများ
- ပို့မှု
 - model.parameters() ပို့မှုများ
 - lr ပို့မှု
 - alpha ပို့မှုများ
 - eps ပို့မှုများ 0
 - weight_decay L2 ဆုတေသန

```
$$ v_t = \beta v_{t-1} + (1-\beta) (\frac{\partial J}{\partial w})^2 \backslash w = w - \alpha \frac{\partial J}{\partial w} \cdot \frac{1}{\sqrt{v_t + \epsilon}} 
```

အကြောင်း

အကြောင်းအရာများ

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

StepLR အကြောင်း

ပို့မှု	ပို့မှု
optimizer	ပို့မှုများ
step_size	အကြောင်း epoch အတွက်
gamma	0.1

```
last_epoch
```

```
epoch
```

```
scheduler.step() scheduler
```

ReduceLROnPlateau (调度器)

参数	说明
optimizer	优化器
mode	'min' 或 'max'
factor	缩放因子
patience	无进展 epoch 数量
threshold	停止阈值
threshold_mode	'rel' 或 'abs'
cooldown	冷却 epoch 数量
min_lr	最小学习率
verbose	输出信息

```
scheduler.step(val_loss) val_loss
```

CosineAnnealingLR (调度器)

参数	说明
optimizer	优化器
T_max	周期 epoch 数量
eta_min	最小学习率
last_epoch	epoch

```
scheduler.step()
```

Conv

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, dilation, groups, bias, padding_mode)
```

```
in_channels$(in_channels,H,W)$
```

- out_channels
- kernel_size

参数

- in_channels
- out_channels
- kernel_size
- stride
- padding

- bias ဗို့
- dilation ဗို့
- groups ဗို့

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

Transposed Conv

`torch.nn.ConvTranspose2d`

ဗို့

ဗို့

ဗို့

1. ဗို့ s-1 ဗို့
2. ဗို့ k-p-1 ဗို့
3. ဗို့
4. ဗို့

ဗို့

- in_channels ဗို့RGB ဗို့3
- out_channels ဗို့
- kernel_size ဗို့
- stride ဗို့0 ဗို့
- padding ဗို့
- bias ဗို့
- out_padding ဗို့



ဗို့ \$ H_{out} = (H_{in} - 1) \times \text{stride} + \text{kernel_size} + \text{output_padding}

Pool

`torch.nn`

ဗို့

- `nn.MaxPool2d`
- `nn.AvgPool2d`
- `nn.AdaptiveMaxPool2d`

ဗို့

- kernel_size ဗို့
- stride ဗို့ kernel_size ဗို့
- padding ဗို့
- dilation ဗို့

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

Linear

`torch.nn.Linear` $y = x \cdot W^T + b$

- x: $\text{Tensor}^{in_features \times 1}$

参数

- in_feature: $\text{int}^{}$
- out_feature: $\text{int}^{}$
- bias: $\text{bool}^{}$

`torch.nn.Linear` $y = x \cdot W^T + b$

Flatten

`torch.nn.Flatten`

- 输入: $(batch_size, channels, height, width)$
- 输出: 2D $(batch_size, channels * height * width)$

Normalization

`torch.nn`

Batch Normalization

`torch.nn.BatchNorm1d`

- 参数

`torch.nn.BatchNorm2d` `torch.nn.BatchNorm3d`

- 输入: $[Batch_size, Channels, Height, Width]$
- 输出: $Batch\text{-normalized}_i$ **Batch** i
- 参数: $Channels$

参数

- 参数
- batch: $batch$
- 参数

Layer Normalization

`torch.nn.LayerNorm`

参数

- 参数
- 参数: **RNN/Transformer**

Group Normalization

nn.GroupNorm

TensorNorm

参数

- num_groups 通道数
- num_channels 通道数
- affine 是否有偏置

Upsample

nn.Upsample

TensorNorm

参数

参数	类型	描述
size	int[tuple]	输出尺寸(256, 256)或None[None]scale_factor
scale_factor	float[tuple]	缩放因子2或3或2或(2, 3)或2或3
mode	str	'nearest'或'bilinear'或'bicubic'或'nearest'
align_corners	bool	bilinear/bicubic或False

Dropout

torch.nn.Dropout

.train() 或 .eval()

Interpolate

torch.nn.functional.interpolate

TensorNorm/张量插值

参数

参数	描述	类型
size	(target_h, target_w)	输出尺寸
scale_factor	float [float, float]	缩放因子 2.0 或 2
mode	'nearest', 'bilinear', 'bicubic', 'area'	插值模式
align_corners	bool	True 或 False

张量插值

TensorNorm/张量插值

- torch.utils.checkpoint
 - 张量插值

- 1. `checkpoint` `fn`
 - 2. `checkpoint` `fn`
 - `fn`
 - `fn` `fn`
 - `**args` `fn`

2

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

## ...
model = BigModel()
inputs = torch.randn(32, 1000)
outputs = model(inputs)
loss = outputs.sum()
loss.backward()  # ...layer1layer2...
```

- `torch.utils.checkpoint_sequential` **Sequential**
 - **Sequential**
 - `functions` **Sequential**
 - `input` **Sequential**
 - `segments` **Sequential**
 - `nn.Sequential` **Sequential** segments `nn.Sequential`
 - `nn.Sequential` segments `nn.Sequential`

2

```
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)
## 亂數 chunk 2 forward checkpoint
output = checkpoint_sequential(model, chunks=2, input=input_tensor)
```

1

10

```
## 亂數种子
accumulation_steps = 4
## 亂數种子
optimizer.zero_grad()

for i ,(inputs,labels) in enumerate(datalaoder):
    outputs = model(inputs)
    # 亂數种子
    loss = criterion(outputs,labels)
    # 亂數种子
    loss = loss / accumulation_steps
    # 亂數种子
    loss.backward() # 亂數种子

    if (i + 1) % accumulation_steps == 0:
        # 亂數种子
        optimizer.step()
        # 亂數种子
        optimizer.zero_grad()
```

三

- 亂子ループ内 \$batch_size\$ = 36
$$\nabla L_{\text{true}} = \frac{1}{32} \sum_{i=1}^{32} \nabla \text{loss}_i$$
 - 乱子ループ内 \$mini_batch\$ = 8
$$\nabla L_{\text{accu}} = \sum_{j=1}^4 \left(\frac{1}{4} \cdot \frac{1}{8} \sum_{k=1}^8 \nabla \text{loss}_{jk} \right) = \frac{1}{32} \sum_{i=1}^{32} \nabla \text{loss}_i$$

1

torch.nn.utils

clip grad value

- `clip_grad_norm_`
 - `torch.nn.utils.clip_grad_value_`
 - `model.parameters()` `grad_fn`

- clip_value 亂數範圍
- clip_grad_norm_ ◦
- 計算各參數L2范數並依此規範化梯度
 - torch.nn.utils.clip_grad_norm_
 - model.parameters() 亂數範圍
 - max_norm 亂數範圍L2范數

亂數範圍

pytorch亂數範圍

```
class EarlyStopping:
    def __init__(self, patience=3):
        self.patience = patience
        self.best_loss = float('inf')
        self.counter = 0
    # 亂數範圍
    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
```

亂數範圍

亂數

亂數範圍

int8 亂數範圍 float32

- torch.quantization ◦ from torch.quantization import quantize_dynamic
 - model 亂數範圍
 - {nn.Linear, nn.LSTM} 亂數範圍
 - dtype 亂數範圍

int8 亂數範圍

- 亂數
 - 亂數範圍
 - 亂數範圍 fuse_modules ◦
 - torch.quantization.fuse_modules
 - 亂數範圍 qconfig ◦
 - torch.quantization.get_default_qconfig
 - fbgemm ◦x86◦ qnnpack ◦ARM

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

- torch.quantization.prepare()
 - inplace 量化

5. 量化前向传播 forward()

6. 量化转换 convert()

 - torch.quantization.convert

1

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

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

## 2. 配置
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)
```

██████████(QAT)██████████“████████”██████████ FakeQuantize ██████████

- 量化
 1. 准备阶段
 - `torch.quantization.fuse_modules`
 - 2. 构建 QAT 配置
 - `torch.quantization.get_default_qat_qconfig`
 - 3. 准备 QAT
 - `torch.quantization.prepare_qat()`
 - 4. 完成 epoch
 - 5. 转换模型
 - `torch.quantization.convert()`

1

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

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

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

## 3. QNN
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. QNN
model_fused.eval()
torch.quantization.convert(model_fused, inplace=True)
```

2

torch.nn.utils.prune

5

mask[0]

TensorBoard

TensorBoard

from torch.utils import from torch.utils.tensorboard import SummaryWriter

- `SummaryWriter` は `tensorboard` モジュール

```
from torch.utils.tensorboard import SummaryWriter
## SummaryWriter は SummaryWriter

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

## SummaryWriter

writer.close()
```

1. `add_scalar`

- `writer.add_scalar()`
 - `tag` は
 - `train/loss` や `eval/loss` など
 - `scalar_value` はスケーラル
 - `global_step` はグローバル

2. `add_graph`

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

3. `add_hparams`

- `writer.add_hparams()`
 - `name`
 - `value`

tqdm

進捗表示

```
from tqdm import tqdm
```

```
tqdm()
```

- `Dataloader` は
- `desc` は
- `total` は
- `leave` は

- unit 旣定の it/s 旣定の batch/s
- ncols 旣定の

`bar.set_postfix() 旣定の`

- 旣定の `train_bar.set_postfix(loss=f'{loss.item():.4f}')`
- 旣定の

```
train_bar.set_postfix({
    'loss': f'{loss.item():.4f}',
    'acc': f'{acc.item():.2%}' # 旣定の
})
```

モデルの保存

- 旣定の

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

- 旣定の

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

- 旣定の

`torch.save(model, path)`

- 旣定の

`model = torch.load(path)`

並列化

並列化

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

並列化

1. 旣定の

並列化するGPU

2. 旣定の

並列化するプロセス

3. `DDP` 旣定の `DistributedSampler` 旣定の
4. `torch.multiprocessing.spawn` 旣定の

逆伝播

- `loss.backward()` 旣定の `.grad` 旣定の

- `optimizer.step()` 丟失函數
 - `optimizer.zero_grad()` 清空梯度
 - `torch.no_grad()` 禁用梯度計算
 - `.requires_grad_` 設定梯度計算
 - `loss.item()` 將損失轉換為 Python 數據型別
 - `optim.zero_grad()` 清空梯度
 - `optimizer.step()` 丟失函數
 - `loss.backward()` 反向傳播
 - `self.training` `nn.Module` 類別的訓練狀態，`True` 表示訓練，`False` 表示評估
 - `grad_fn`
 - `.permute()` 轉置 `.reshape()` 重新整理