PyTorch 快速参考

张量、自动微分、神经网络与训练流程

张量

创建张量

```
import torch
a = torch.tensor([1, 2, 3])
b = torch.zeros(2, 3)
c = torch.ones(3, 3)
d = torch.randn(2, 4)  # 正态分布
```

张量构造函数

torch.zeros(m, n)	全零张量，形状 (m, n)
torch.ones(m, n)	全一张量，形状 (m, n)
torch.randn(m, n)	标准正态分布随机张量
torch.arange(start, end, step)	等差数列
torch.linspace(start, end, steps)	固定数量的均匀点
torch.eye(n)	单位矩阵
torch.empty(m, n)	未初始化内存

与 NumPy 互操作

```
t = torch.from_numpy(np_array)
arr = tensor.numpy()  # 共享内存
t = torch.as_tensor(np_array)
```

自动微分

追踪梯度

```
x = torch.tensor([2.0, 3.0],
    requires_grad=True)
y = (x ** 2).sum()
y.backward()
print(x.grad)  # tensor([4., 6.])
```

禁用梯度追踪

```
with torch.no_grad():
    pred = model(x)  # 仅用于推理
x_det = x.detach()  # 从计算图中分离
```

梯度控制

x.requires_grad_(True)	原地启用梯度追踪
x.grad.zero_()	清零累积梯度
x.detach()	返回无梯度历史的新张量
x.grad	访问存储的梯度

神经网络

定义模型

```
import torch.nn as nn
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

Sequential 模型

```
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, 10))
```

常用层

nn.Linear(in, out)	全连接层
nn.Conv2d(c_in, c_out, k)	2D 卷积，卷积核大小 k
nn.BatchNorm2d(n)	批归一化
nn.LSTM(in, hidden)	LSTM 循环层
nn.Dropout(p)	概率为 p 的 Dropout
nn.Embedding(vocab, dim)	嵌入查找表

数据加载

自定义 Dataset

```
from torch.utils.data import Dataset, DataLoader
class MyData(Dataset):
    def __init__(self, X, y):
        self.X, self.y = X, y
    def __len__(self): return len(self.X)
    def __getitem__(self, i):
        return self.X[i], self.y[i]
```

DataLoader

```
loader = DataLoader(dataset, batch_size=32,
    shuffle=True, num_workers=2)
for batch_x, batch_y in loader:
    output = model(batch_x)
```

内置数据集

```
from torchvision import datasets, transforms
t = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))])
data = datasets.MNIST("data", train=True,
    download=True, transform=t)
```

训练循环

标准训练循环

```
model.train()
for epoch in range(num_epochs):
    for X, y in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(X), y)
        loss.backward()
        optimizer.step()
```

评估

```
model.eval()
with torch.no_grad():
    correct = 0
    for X, y in test_loader:
        pred = model(X).argmax(dim=1)
        correct += (pred == y).sum().item()
```

训练检查清单

model.train()	启用 Dropout / BatchNorm 训练模式
model.eval()	切换到推理模式
optimizer.zero_grad()	反向传播前清零梯度
loss.backward()	计算梯度
optimizer.step()	更新参数

优化器

常用优化器

```
import torch.optim as optim
opt = optim.SGD(model.parameters(), lr=0.01,
    momentum=0.9)
opt = optim.Adam(model.parameters(), lr=1e-3)
opt = optim.AdamW(model.parameters(), lr=1e-3,
    weight_decay=0.01)
```

学习率调度器

```
sched = optim.lr_scheduler.StepLR(
    opt, step_size=10, gamma=0.1)
# 训练循环中：每个 epoch 后调用 sched.step()
```

优化器对比

SGD	简单，需调参，与 momentum 配合效果好
Adam	自适应学习率，收敛快，默认首选
AdamW	Adam + 解耦权重衰减
RMSprop	自适应，适合 RNN

损失函数

常用损失函数

nn.CrossEntropyLoss()	分类（输入 logits，无需 softmax）
nn.BCEWithLogitsLoss()	二分类（输入 logits）
nn.MSELoss()	回归（均方误差）
nn.L1Loss()	回归（平均绝对误差）
nn.NLLLoss()	负对数似然（log_softmax 后使用）
nn.HuberLoss()	鲁棒回归（对异常值不敏感）

使用示例

```
criterion = nn.CrossEntropyLoss()
loss = criterion(logits, targets)
# logits: (batch, classes), targets: (batch,)
```

自定义损失函数

```
def focal_loss(pred, target, gamma=2.0):
    ce = nn.functional.cross_entropy(
        pred, target, reduction="none")
    pt = torch.exp(-ce)
    return ((1 - pt) ** gamma * ce).mean()
```

保存与加载

保存/加载 state dict（推荐）

```
torch.save(model.state_dict(), "model.pt")
model = Net()
model.load_state_dict(
    torch.load("model.pt", weights_only=True))
```

保存完整 checkpoint

```
torch.save({
    "epoch": epoch,
    "model": model.state_dict(),
    "optimizer": opt.state_dict(),
    "loss": loss}, "checkpoint.pt")
```

加载 checkpoint

```
ckpt = torch.load("checkpoint.pt",
    weights_only=False)
model.load_state_dict(ckpt["model"])
opt.load_state_dict(ckpt["optimizer"])
```

PyTorch 快速参考

GPU

设备管理

```
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "cpu")
model = model.to(device)
x = x.to(device)
```

GPU 工具

torch.cuda.is_available()	检查 CUDA 是否可用
torch.cuda.device_count()	GPU 数量
torch.cuda.memory_allocated()	当前 GPU 显存占用（字节）
torch.cuda.empty_cache()	释放未使用的缓存显存

多 GPU

```
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
model = model.to(device)
```

常用模式

权重初始化

```
def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)
model.apply(init_weights)
```

梯度裁剪

```
torch.nn.utils.clip_grad_norm_(
    model.parameters(), max_norm=1.0)
```

冻结层

```
for param in model.fc1.parameters():
    param.requires_grad = False
```

模型参数统计

```
total = sum(p.numel()
    for p in model.parameters())
trainable = sum(p.numel()
    for p in model.parameters()
    if p.requires_grad)
```