

PyTorch クイックリファレンス

テンソル、自動微分、ニューラルネットワーク、学習

テンソル

テンソルの作成

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

テンソルコンストラクタ

torch.zeros(m, n)	全ゼロ、形状(m, n)
torch.ones(m, n)	全1、形状(m, n)
torch.randn(m, n)	標準正規分布のランダム値
torch.arange(start, end, step)	等間隔の値
torch.linspace(start, end, steps)	固定数の点
torch.eye(n)	単位行列
torch.empty(m, n)	未初期化メモリ

NumPy との相互変換

```
t = torch.from_numpy(np_array)
arr = tensor.numpy() # shares memory
t = torch.as_tensor(np_array)
```

自動微分

勾配の追跡

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

勾配追跡の無効化

```
with torch.no_grad():
    pred = model(x) # inference only
x_det = x.detach() # detach from graph
```

勾配の制御

x.requires_grad_(True)	インプレースで勾配追跡を有効化
x.grad.zero_()	累積勾配をリセット
x.detach()	勾配履歴なしの新しいテンソル
x.grad	保存された勾配にアクセス

ニューラルネットワーク

モデルの定義

```
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)
```

シーケンシャルモデル

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

よく使うレイヤー

nn.Linear(in, out)	全結合層
nn.Conv2d(c_in, c_out, k)	2D 畳み込み、カーネルサイズ k
nn.BatchNorm2d(n)	バッチ正規化
nn.LSTM(in, hidden)	LSTM リカレント層
nn.Dropout(p)	確率 p のドロップアウト
nn.Embedding(vocab, dim)	埋め込みルックアップテーブル

データのロード

カスタムデータセット

```
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)
```

組み込みデータセット

```
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)
```

学習ループ

標準的な学習ループ

```
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()
```

評価

```
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()
```

学習チェックリスト

model.train()	ドロップアウト/バッチ正規化の学習モードを有効化
model.eval()	推論モードに切り替え
optimizer.zero_grad()	逆伝播の前に勾配をクリア
loss.backward()	勾配を計算
optimizer.step()	パラメータを更新

オプティマイザ

よく使うオプティマイザ

```
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)
```

学習率スケジューラ

```
sched = optim.lr_scheduler.StepLR(
    opt, step_size=10, gamma=0.1)
# in loop: sched.step() after each epoch
```

オプティマイザの比較

SGD	シンプル、チューニングが必要、モメンタムで効果的
Adam	適応的学習率、高速収束、デフォルト
AdamW	重み減衰を分離した Adam
RMSprop	適応的、RNN に有効

損失関数

よく使う損失関数

nn.CrossEntropyLoss()	分類 (ロジット、ソフトマックスなし)
nn.BCEWithLogitsLoss()	二値分類 (ロジット)
nn.MSELoss()	回帰 (平均二乗誤差)
nn.L1Loss()	回帰 (平均絶対誤差)
nn.NLLLoss()	負の対数尤度 (log_softmax の後)
nn.HuberLoss()	ロバスト回帰 (外れ値に対して頑健)

使用例

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

カスタム損失

```
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()
```

保存とロード

状態辞書の保存/ロード (推奨)

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

完全なチェックポイントの保存

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

チェックポイントのロード

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

PyTorch クイックリファレンス

GPU

デバイスの管理

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

GPU ユーティリティ

torch.cuda.is_available()	CUDA が利用可能か確認
torch.cuda.device_count()	GPU の数
torch.cuda.memory_allocated()	現在の GPU メモリ使用量 (バイト)
torch.cuda.empty_cache()	未使用のキャッシュメモリを解放

マルチ GPU

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

よくあるパターン

重みの初期化

```
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)
```

勾配のクリッピング

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

レイヤーのフリーズ

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

モデルのサマリー

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