

Referência Rápida de PyTorch

Tensores, autograd, redes neurais e treinamento

Tensores

Criando Tensores

```
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
```

Construtores de Tensor

torch.zeros(m, n)	Todos zeros, forma (m, n)
torch.ones(m, n)	Todos uns, forma (m, n)
torch.randn(m, n)	Normal padrão aleatório
torch.arange(start, end, step)	Valores igualmente espaçados
torch.linspace(start, end, steps)	Número fixo de pontos
torch.eye(n)	Matriz identidade
torch.empty(m, n)	Memória não inicializada

Interoperabilidade com NumPy

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

Autograd

Rastreamento de Gradientes

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

Desabilitar Rastreamento de Gradiente

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

Controle de Gradiente

x.requires_grad_(True)	Habilitar rastreamento de grad in-place
x.grad.zero_()	Zerar gradientes acumulados
x.detach_()	Novo tensor sem histórico de gradiente
x.grad	Acessar gradientes armazenados

Redes Neurais

Definir um Modelo

```
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)
```

Modelo Sequencial

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

Camadas Comuns

nn.Linear(in, out)	Camada totalmente conectada
nn.Conv2d(c_in, c_out, k)	Convolação 2D, tamanho de kernel k
nn.BatchNorm2d(n)	Normalização em lote
nn.LSTM(in, hidden)	Camada recorrente LSTM
nn.Dropout(p)	Dropout com probabilidade p
nn.Embedding(vocab, dim)	Tabela de lookup de embeddings

Carregamento de Dados

Dataset Personalizado

```
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)
```

Datasets Integrados

```
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)
```

Loop de Treinamento

Loop de Treinamento Padrão

```
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()
```

Avaliação

```
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()
```

Checklist de Treinamento

model.train()	Habilitar dropout / batch norm de treinamento
model.eval()	Mudar para modo de inferência
optimizer.zero_grad()	Limpar gradientes antes do backward
loss.backward()	Calcular gradientes
optimizer.step()	Atualizar parâmetros

Otimizadores

Otimizadores Comuns

```
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)
```

Scheduler de Taxa de Aprendizado

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

Comparação de Otimizadores

SGD	Simple, requer ajuste, bom com momentum
Adam	LR adaptativo, convergência rápida, padrão
AdamW	Adam com decaimento de peso desacoplado
RMSprop	Adaptativo, bom para RNNs

Funções de Perda

Funções de Perda Comuns

nn.CrossEntropyLoss()	Classificação (logits, sem softmax)
nn.BCEWithLogitsLoss()	Classificação binária (logits)
nn.MSELoss()	Regressão (erro quadrático médio)
nn.L1Loss()	Regressão (erro absoluto médio)
nn.NLLLoss()	Log-verossimilhança negativa (após log_softmax)
nn.HuberLoss()	Regressão robusta (menos sensível a outliers)

Uso

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

Perda Personalizada

```
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()
```

Salvar e Carregar

Salvar / Carregar State Dict (Recomendado)

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

Salvar Checkpoint Completo

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

Carregar Checkpoint

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

GPU

Gerenciamento de Dispositivo

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

Referência Rápida de PyTorch

Utilitários de GPU

<code>torch.cuda.is_available()</code>	Verificar se CUDA está disponível
<code>torch.cuda.device_count()</code>	Número de GPUs
<code>torch.cuda.memory_allocated()</code>	Uso atual de memória GPU (bytes)
<code>torch.cuda.empty_cache()</code>	Liberar memória em cache não utilizada

Multi-GPU

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

Padrões Comuns

Inicialização de Pesos

```
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)
```

Recorte de Gradiente

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

Congelar Camadas

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

Resumo do Modelo

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