Referencia Rápida de PyTorch

Tensores, autograd, redes neuronales y entrenamiento

Tensores

Crear 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
```

Constructores de Tensores

`torch.zeros(m, n)`	Todo ceros, forma (m, n)
`torch.ones(m, n)`	Todo unos, forma (m, n)
`torch.randn(m, n)`	Aleatorio normal estándar
`torch.arange(start, end, step)`	Valores equiespaciados
`torch.linspace(start, end, steps)`	Número fijo de puntos
`torch.eye(n)`	Matriz identidad
`torch.empty(m, n)`	Memoria no inicializada

Interoperabilidad con NumPy

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

Autograd

Seguimiento de Gradientes

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

Deshabilitar Seguimiento de Gradientes

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

Control de Gradientes

`x.requires_grad_(True)`	Habilitar seguimiento de grad en sitio
`x.grad.zero_()`	Restablecer gradientes acumulados
`x.detach()`	Nuevo tensor sin historial de grad
`x.grad`	Acceder a gradientes almacenados

Redes Neuronales

Definir un 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 Secuencial

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

Capas Comunes

`nn.Linear(in, out)`	Capa totalmente conectada
`nn.Conv2d(c_in, c_out, k)`	Convolución 2D, tamaño de kernel k
`nn.BatchNorm2d(n)`	Normalización por lotes
`nn.LSTM(in, hidden)`	Capa recurrente LSTM
`nn.Dropout(p)`	Dropout con probabilidad p
`nn.Embedding(vocab, dim)`	Tabla de búsqueda de embeddings

Carga de Datos

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)
```

Ciclo de Entrenamiento

Ciclo de Entrenamiento Estándar

```
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()
```

Evaluación

```
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()
```

Lista de Verificación de Entrenamiento

`model.train()`	Habilitar dropout / batch norm de entrenamiento
`model.eval()`	Cambiar a modo de inferencia
`optimizer.zero_grad()`	Limpiar gradientes antes de backward
`loss.backward()`	Calcular gradientes
`optimizer.step()`	Actualizar parámetros

Optimizadores

Optimizadores Comunes

```
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)
```

Planificador de Tasa de Aprendizaje

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

Comparativa de Optimizadores

SGD	Simple, requiere ajuste, bueno con momentum
Adam	LR adaptativa, convergencia rápida, predeterminado
AdamW	Adam con decaimiento de peso desacoplado
RMSprop	Adaptativo, bueno para RNNs

Funciones de Pérdida

Funciones de Pérdida Comunes

`nn.CrossEntropyLoss()`	Clasificación (logits, sin softmax)
`nn.BCEWithLogitsLoss()`	Clasificación binaria (logits)
`nn.MSELoss()`	Regresión (error cuadrático medio)
`nn.L1Loss()`	Regresión (error absoluto medio)
`nn.NLLLoss()`	Log-verosimilitud negativa (tras log_softmax)
`nn.HuberLoss()`	Regresión robusta (menos sensible a valores atípicos)

Uso

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

Pérdida 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()
```

Guardar y Cargar

Guardar / Cargar State Dict (Recomendado)

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

Guardar Checkpoint Completo

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

Cargar Checkpoint

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

Referencia Rápida de PyTorch

GPU

Gestión de Dispositivo

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

Utilidades de GPU

`torch.cuda.is_available()`	Verificar si CUDA está disponible
`torch.cuda.device_count()`	Número de GPUs
`torch.cuda.memory_allocated()`	Uso actual de memoria GPU (bytes)
`torch.cuda.empty_cache()`	Liberar memoria caché no utilizada

Multi-GPU

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

Patrones Comunes

Inicialización 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 Gradientes

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

Congelar Capas

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

Resumen del Modelo

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