PyTorch Riferimento Rapido

Tensori, autograd, reti neurali e addestramento

Tensori

Creazione di Tensori

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

Costruttori di Tensori

`torch.zeros(m, n)`	Tutti zeri, forma (m, n)
`torch.ones(m, n)`	Tutti uno, forma (m, n)
`torch.randn(m, n)`	Casuale normale standard
`torch.arange(start, end, step)`	Valori equidistanti
`torch.linspace(start, end, steps)`	Numero fisso di punti
`torch.eye(n)`	Matrice identità
`torch.empty(m, n)`	Memoria non inizializzata

Interoperabilità con NumPy

```
t = torch.from_numpy(np_array)
arr = tensor.numpy()  # condivide memoria
t = torch.as_tensor(np_array)
```

Autograd

Tracciamento dei Gradienti

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

Disabilitare il Tracciamento dei Gradienti

```
with torch.no_grad():
    pred = model(x)  # solo inferenza
x_det = x.detach()  # stacca dal grafo
```

Controllo dei Gradienti

`x.requires_grad_(True)`	Abilita il tracciamento in-place
`x.grad.zero_()`	Azzera i gradienti accumulati
`x.detach()`	Nuovo tensore senza storia del gradiente
`x.grad`	Accede ai gradienti memorizzati

Reti Neurali

Definire un Modello

```
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)
```

Modello Sequenziale

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

Layer Comuni

`nn.Linear(in, out)`	Layer completamente connesso
`nn.Conv2d(c_in, c_out, k)`	Convoluzione 2D, kernel di dimensione k
`nn.BatchNorm2d(n)`	Normalizzazione batch
`nn.LSTM(in, hidden)`	Layer ricorrente LSTM
`nn.Dropout(p)`	Dropout con probabilità p
`nn.Embedding(vocab, dim)`	Tabella di lookup per embedding

Caricamento Dati

Dataset Personalizzato

```
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)
```

Dataset Integrati

```
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 di Addestramento

Ciclo di Addestramento Standard

```
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()
```

Valutazione

```
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 di Addestramento

`model.train()`	Abilita dropout / batch norm in addestramento
`model.eval()`	Passa alla modalità inferenza
`optimizer.zero_grad()`	Azzera i gradienti prima del backward
`loss.backward()`	Calcola i gradienti
`optimizer.step()`	Aggiorna i parametri

Ottimizzatori

Ottimizzatori Comuni

```
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 del Learning Rate

```
sched = optim.lr_scheduler.StepLR(
    opt, step_size=10, gamma=0.1)
# nel ciclo: sched.step() dopo ogni epoca
```

Confronto Ottimizzatori

SGD	Semplice, richiede tuning, buono con momentum
Adam	LR adattivo, convergenza rapida, default
AdamW	Adam con weight decay disaccoppiato
RMSprop	Adattivo, ottimo per RNN

Funzioni di Perdita

Funzioni di Perdita Comuni

`nn.CrossEntropyLoss()`	Classificazione (logit, senza softmax)
`nn.BCEWithLogitsLoss()`	Classificazione binaria (logit)
`nn.MSELoss()`	Regressione (errore quadratico medio)
`nn.L1Loss()`	Regressione (errore assoluto medio)
`nn.NLLLoss()`	Log-verosimiglianza negativa (dopo log_softmax)
`nn.HuberLoss()`	Regressione robusta (meno sensibile agli outlier)

Utilizzo

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

Perdita Personalizzata

```
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()
```

Salvataggio e Caricamento

Salva / Carica State Dict (Raccomandato)

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

Salva Checkpoint Completo

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

Carica Checkpoint

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

GPU

Gestione del Dispositivo

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

PyTorch Riferimento Rapido

Utilità GPU

`torch.cuda.is_available()`	Verifica se CUDA è disponibile
`torch.cuda.device_count()`	Numero di GPU
`torch.cuda.memory_allocated()`	Utilizzo corrente memoria GPU (byte)
`torch.cuda.empty_cache()`	Libera memoria cache inutilizzata

Multi-GPU

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

Pattern Comuni

Inizializzazione dei Pesi

```
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)
```

Clipping del Gradiente

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

Congela Layer

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

Sommario del Modello

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