

RÉFÉRENCE RAPIDE PYTORCH

Tenseurs, autograd, réseaux de neurones et entraînement

Tenseurs

Créer des tenseurs

```
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
```

Constructeurs de tenseurs

torch.zeros(m, n) Tous zéros, forme (m, n)
torch.ones(m, n) Tous uns, forme (m, n)
torch.randn(m, n) Normal standard aléatoire

torch.arange(start, end, step) Valeurs régulièrement espacées

torch.linspace(start, end, steps) Nombre fixe de points
torch.eye(n) Matrice identité
torch.empty(m, n) Mémoire non initialisée

Interopérabilité NumPy

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

Autograd

Suivi des gradients

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

Désactiver le suivi des gradients

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

Contrôle des gradients

x.requires_grad_(True) Activer le suivi de gradient en place
x.grad.zero_() Réinitialiser les gradients accumulés
x.detach_() Nouveau tenseur sans historique de gradient
x.grad Accéder aux gradients stockés

Réseaux de neurones

Définir un modèle

```
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)
```

Modèle séquentiel

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

Couches courantes

nn.Linear(in, out) Couche entièrement connectée
nn.Conv2d(c_in, c_out, k) Convolution 2D, taille de nouveau k
nn.BatchNorm2d(n) Normalisation par lots
nn.LSTM(in, hidden) Couche récurrente LSTM
nn.Dropout(p) Dropout avec probabilité p
nn.Embedding(vocab, dim) Table de recherche d'embeddings

Chargement des données

Dataset personnalisé

```
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 intégrés

```
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)
```

Boucle d'entraînement

Boucle d'entraînement 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()
```

Évaluation

```
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()
```

Liste de contrôle d'entraînement

model.train() Activer dropout / entraînement batch norm
model.eval() Passer en mode inférence
optimizer.zero_grad() Effacer les gradients avant rétropropagation
loss.backward() Calculer les gradients
optimizer.step() Mettre à jour les paramètres

Optimiseurs

Optimiseurs courants

```
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)
```

Planificateur de taux d'apprentissage

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

Comparaison des optimiseurs

SGD Simple, nécessite réglage, bon avec momentum
Adam LR adaptatif, convergence rapide, par défaut
AdamW Adam avec décroissance de poids découplée
RMSprop Adaptatif, bon pour les RNN

Fonctions de perte

Fonctions de perte courantes

nn.CrossEntropyLoss() Classification (logits, sans softmax)
nn.BCEWithLogitsLoss() Classification binaire (logits)
nn.MSELoss() Régression (erreur quadratique moyenne)
nn.L1Loss() Régression (erreur absolue moyenne)
nn.NLLLoss() Log-vraisemblance négative (après log_softmax)
nn.HuberLoss() Régression robuste (moins sensible aux valeurs aberrantes)

Utilisation

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

Perte personnalisée

```
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()
```

Sauvegarde et chargement

Sauvegarder / Charger le state dict (recommandé)

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

Sauvegarder un checkpoint complet

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

Charger un checkpoint

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

GPU

Gestion des appareils

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

Utilitaires GPU

torch.cuda.is_available() Vérifier si CUDA est disponible
torch.cuda.device_count() Nombre de GPU
torch.cuda.memory_allocated() Utilisation mémoire GPU actuelle (octets)
torch.cuda.empty_cache() Libérer la mémoire cache inutilisée

Multi-GPU

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

Motifs courants

Initialisation des poids

```
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)
```

Écrêtage du gradient

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

Geler des couches

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

Résumé du modèle

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