

PyTorch: Tensoren & Modellbausteine

PyTorch 2.x • N = Batchgröße, K = Klassen, D = Merkmale • Seite 1: unabhängige Beispiele mit Platzhaltern

01 Einstieg & Gerät

```
import torch
from torch import nn

print(torch.__version__)
use_cuda = torch.cuda.is_available()
device = torch.device(
    "cuda" if use_cuda else "cpu")
print(device)
```

Tensor: mehrdimensionales Array mit dtype und device. PyTorch ergänzt automatische Ableitung und neuronale Netze. Shapes beschreiben Struktur, nicht automatisch Bedeutung.

Geräte: CUDA für compatible GPUs; CPU immer als Rückfall. ROCm nutzt ebenfalls torch.cuda. Apple MPS ggf. mit torch.backends.mps.is_available() prüfen.

Installation: passenden Befehl aus pytorch.org/get-started/locally wählen. CUDA-Unterstützung hängt von PyTorch-Build, GPU und Treiber ab, nicht nur vom installierten Toolkit.

02 Tensoren erzeugen & prüfen

```
a = torch.tensor([1., 2., 3.])
x = torch.rand(4, 1, 28, 28)
z = torch.zeros(2, 3)
o = torch.ones_like(z)
r = torch.randn(2, 3)
ids = torch.arange(5)
y = torch.tensor([2, 7, 0, 9],
                 dtype=torch.long)
print(x.shape, x.ndim, x.numel())
print(x.dtype, x.device)
```

rand: gleichverteilt in [0, 1); randn: Standardnormalverteilung. empty reserviert uninitialisierten Speicher. tensor(data) kopiert; as_tensor kann Speicher teilen.

Typen: float32 üblich für Eingaben und Gewichte; int64 / long für Klassenindizes; bool für Masken. x.float(), x.long(), x.to(dtype=...) konvertieren.

```
x = x.to(device)
y = y.to(device)
x_cpu = x.cpu()
scalar = a.sum().item()
values = a.detach().cpu().tolist()
```

Tensor.to(...) gibt den übertragenen Tensor zurück; zuweisen! item() nur für ein Element. detach() trennt Autograd; cpu() überträgt; tolist() erzeugt Python-Werte.

03 Shapes, Achsen & Auswahl

```
Bildkonvention NCHW: N Batch, C Kanäle, H
Höhe, W Breite. (4, 1, 28, 28): vier Graustufenbilder.
(4, 784): dieselben Pixel als Merkmale.

x = torch.rand(4, 1, 28, 28)
single = x[0]      # (1, 28, 28)
one = x[0:1]       # (1, 1, 28, 28)
flat = x.flatten(start_dim=1)
# flat: (4, 784), Batch bleibt
v = torch.rand(5)  # (5,)
m = torch.rand(1, 5) # (1, 5)
```

(5,) ≠ (1, 5): Vektor gegenüber Matrix mit einer Zeile; andere Anzahl Achsen. Indizierung mit Ganzzahl entfernt eine Achse, Slice erhält sie.

```
x[0, 0, :,5, :5]
x[x > 0.5]      # flache Auswahl
u = v.unsqueeze(0) # (1, 5)
v2 = u.squeeze(0)  # (5,)
hwc = x.permute(0, 2, 3, 1)
t = m.transpose(0, 1) # (5, 1)
```

squeeze() ohne dim entfernt **alle** Einserachsen, eventuell auch den Batch. Negative dim zählen von hinten; dim=-1 ist die letzte Achse.

04 Umformen & Zusammenfügen

```
x.reshape(4, -1) # -1 berechnen
x.view(4, -1)
a = torch.zeros(2, 3)
b = torch.ones(2, 3)
torch.cat([a, b], dim=0) # (4, 3)
torch.stack([a, b], dim=0) # (2, 2, 3)
```

reshape: View wenn möglich, sonst Kopie. **view:** benötigt passende Speicher-Strides; nach permute ggf. contiguous().view(...). Anzahl Elemente muss passen.

cat verbindet auf vorhandener Achse; **stack** erzeugt eine neue. repeat kopiert Daten; expand erweitert Einserachsen als View. Views können Speicher mit dem Ursprung teilen.

05 Python / NumPy verbinden

NumPy: torch.from_numpy(array) teilt CPU-Speicher; array = x.detach().cpu().numpy() kann ihn ebenfalls teilen. Für unabhängige Daten .copy() bzw. .clone() verwenden.

06 Rechnen, Reduzieren, Broadcast

```
a = torch.rand(2, 3)
b = torch.rand(2, 3)
a + b; a * b      # elementweise
w = torch.rand(3, 4)
a @ w             # (2, 4)
a.sum(); a.mean()
a.mean(dim=1)     # (2,)
a.mean(dim=1, keepdim=True) # (2, 1)
a.argmax(dim=1)   # Indizes (2,)
```

*** ≠ @:** elementweise Produkt gegenüber Matrixmultiplikation. Für (N, D) @ (D, K) ist das Ergebnis (N, K).

Broadcasting: Achsen von rechts vergleichen: gleich groß oder eine davon 1; fehlende Achsen gelten als 1. (4, 3) + (3,) → (4, 3).

Falle: (N, 1) - (N,) ergibt (N, N), nicht (N, 1). Vor Loss-Berechnung Shapes ausdrücklich prüfen!

```
torch.allclose(a, b)
torch.isfinite(a).all()
torch.clamp(a, min=0, max=1)
torch.where(a > 0.5, a, b)
```

07 Autograd & Speicher

```
v = torch.tensor([2., 3.],
                 requires_grad=True)
loss = (v * v).sum()
loss.backward()
print(v.grad) # tensor([4., 6.] )
```

Autograd baut bei der Vorwärtsrechnung einen Graphen. backward() berechnet Ableitungen; .grad wird standardmäßig für Leaf-Tensoren mit requires_grad gefüllt.

Gradienten **addieren sich** bei erneutem backward(). Vor jedem gewöhnlichen Optimierungsschritt zero_grad() verwenden. Der Graph wird gewöhnlich nach backward freigegeben.

clone() kopiert Daten und bleibt ableitbar; **detach()** trennt den Graphen, teilt Speicher. Unabhängige graphfreie Kopie: x.detach().clone().

In-place-Operationen (z. B. add_) können Autograd stören. Kein .data zum Umgehen der Ableitung. Für reine Auswertung no_grad() oder inference_mode().

08 Netzwerk-Bausteine

```
nn.Linear(D, K): letzte Achse D → K.
nn.Flatten(1): Batch erhalten. ReLU:
Nichtlinearität. Dropout: zufälliges Maskieren im
Training.

layers = nn.Sequential(
    nn.Flatten(start_dim=1),
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 10),
)
```

Conv2d(Cin, Cout, k): verarbeitet NCHW. Mit stride=1, padding=1, k=3 bleibt H/W gleich. MaxPool2d(2) halbiert üblicherweise H/W.

BatchNorm: lernt Skalierung und verwaltet Statistik. **ModuleList:** registriert mehrere Teilmodule. Eine normale Python-Liste registriert enthaltene Layer nicht automatisch.

Modellaufruf: model(x) statt model.forward(x), damit Hooks greifen. model.parameters() für Optimizer, state_dict() für Gewichte und persistente Puffer.

09 Loss passend zur Aufgabe

Mehrklassen: CrossEntropyLoss. Für Klassifikation logits (N, K), targets (N,) mit long und Werten 0...K-1. Klassenachse dim=1; bei räumlichen Aufgaben (N, K, H, W).

Kein Softmax davor! CrossEntropyLoss erwartet rohe Logits und kombiniert LogSoftmax mit NLLLoss. Wahrscheinlichkeiten nur für Anzeige: logits.softmax(dim=1).

Binär / Multi-Label: BCEWithLogitsLoss; rohe Logits und float-Ziele mit gleicher Shape. Mehrere Labels können gleichzeitig richtig sein.

Regression: MSELoss oder L1Loss; Vorhersage und Ziel sollen dieselbe Shape haben. mean reduziert standardmäßig über alle Loss-Elemente.

10 Training ≠ Auswertung

model.train(): Training für Dropout / BatchNorm. **model.eval():** Auswertungsverhalten. eval() deaktiviert **nicht** die Gradientenberechnung.

no_grad(): ohne Gradientenaufzeichnung. **inference_mode():** zusätzlich weniger Verwaltungsaufwand; darin erzeugte Tensoren sind für spätere Autograd-Rechnungen eingeschränkt.

PyTorch: Training, Inferenz & Praxis

Seite 2: Abschnitte 11–15 bilden zusammen ein Demo-Programm; 16–18 sind ergänzende Muster und Hinweise

11 Daten & Setup (gemeinsam)

```
import torch
from torch import nn
from torch.utils.data import (
    TensorDataset, DataLoader,
)

torch.manual_seed(42)
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "cpu")
images = torch.rand(128, 1, 28, 28)
labels = torch.randint(0, 10, (128,))
dataset = TensorDataset(images, labels)
loader = DataLoader(
    dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(
    dataset, batch_size=32, shuffle=False)
```

Nur Demo-Daten: zufällige Pixel und Klassen zeigen den Ablauf, keine lernbare Aufgabe. Hier nutzt Validation denselben Datensatz; für echte Bewertung separat aufteilen!

Eigener Dataset: `__len__()` und `__getitem__()` (index) liefern (input, target). `DataLoader` bildet Batches. Training meist `shuffle=True`; Validation ohne zufällige Datenaugmentation.

12 Eigenes Modell & Optimizer

```
class DigitClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(1),
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 10),
        )

    def forward(self, x):
        return self.net(x)

model = DigitClassifier().to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(
    model.parameters(), lr=1e-3)
```

Zuerst Modell auf das Gerät verschieben, dann Optimizer anlegen. Optimizer hält Parameterreferenzen. Lernrate und `weight_decay` passend zur Aufgabe wählen; SGD ist eine Alternative.

13 Eine Trainingsepoche

```
def train_epoch(loader):
    model.train()
    loss_sum, count = 0.0, 0
    for inputs, targets in loader:
        inputs = inputs.to(device)
        targets = targets.to(device)
        optimizer.zero_grad(
            set_to_none=True)
        logits = model(inputs)
        loss = loss_fn(logits, targets)
        loss.backward()
        optimizer.step()
        n = targets.size(0)
        loss_sum += loss.item() * n
        count += n
    return loss_sum / count
```

Reihenfolge: Gradienten leeren → Vorwärtsrechnung → Loss → backward → step. Voraussetzung: nichtleerer Loader, ein Klassenindex je Beispiel.

`loss.item()` erzeugt einen Python-Skalar ohne Graph. Gewichtung nach Batchgröße liefert hier den Mittelwert je Beispiel, auch bei kleinerem letztem Batch.

14 Validation & Klassen-Treffer

```
@torch.inference_mode()
def evaluate(loader):
    model.eval()
    correct, count = 0, 0
    for inputs, targets in loader:
        inputs = inputs.to(device)
        targets = targets.to(device)
        logits = model(inputs)
        predicted = logits.argmax(1)
        correct += (predicted == targets
                    ).sum().item()
        count += targets.size(0)
    return correct / count

for epoch in range(3):
    loss = train_epoch(loader)
    accuracy = evaluate(val_loader)
    print(epoch, loss, accuracy)
```

`argmax` liefert Klassenindizes; `Softmax` ist dafür unnötig. `eval()` und `inference_mode()` ergänzen einander. `train_epoch()` schaltet danach wieder auf Training.

15 Inferenz & Gewichte speichern

```
model.eval()
with torch.inference_mode():
    batch = images[:1].to(device)
    logits = model(batch)
    probabilities = logits.softmax(1)
    prediction = logits.argmax(1)
    print(prediction.cpu().tolist())

torch.save(model.state_dict(),
            "digit_weights.pt")
restored = DigitClassifier().to(device)
state = torch.load(
    "digit_weights.pt",
    map_location=device,
    weights_only=True)
restored.load_state_dict(state)
restored.eval()
```

Gewichte brauchen dieselbe Modellarchitektur sowie passende Vorverarbeitung und Klassenreihenfolge. `map_location` erlaubt CPU/GPU-Wechsel. Nur vertrauenswürdige Dateien laden.

Training fortsetzen: auch `optimizer.state_dict()`, Epoche, Scheduler und ggf. AMP-Scaler sichern. Bestes Modell: Zustand direkt speichern oder `deepcopy` verwenden, nicht nur Referenz halten.

16 Debugging & Diagnose

```
print(f"{inputs.shape=}")
print(inputs.dtype, inputs.device)
assert inputs.ndim == 4
assert targets.dtype == torch.long
assert logits.shape[0] == len(targets)
assert logits.shape[1] == 10
assert torch.isfinite(loss).item()
parameter_count = sum(
    p.numel() for p in model.parameters()
    if p.requires_grad)
```

Diagnosefragment nutzt `inputs`, `targets`, `logits` und `loss` aus einem Trainingsschritt; bei Funktionen z. B. am Breakpoint prüfen.

Typische Fehler: `mat1/mat2` → Merkmalsdimensionen; `expected Long` → Klassenziel falsch; `same device` → CPU/GPU gemischt. `NaN/Inf`: Daten, Loss, Lernrate und Gradienten prüfen.

Kein Lernen: `backward` / `step` vergessen? Training versehentlich ohne Gradienten? Optimizer mit falschen Parametern? Loss mit `item()` / `detach()` vom Graph getrennt?

17 Leistung & Stabilität

GPU-Speicher: Batchgröße senken; unnötige Graphen vermeiden. Loss-Verlauf mit `loss.item()` speichern, nicht mit graphtragenden Tensoren. `empty_cache()` gibt keine belegten Tensoren frei.

Datentransfer: bei CUDA ggf. `DataLoader(pin_memory=True)` und `.to(device, non_blocking=True)`. `num_workers` messen; bei Multiprocessing Programmeinstieg mit `__main__` schützen.

Mixed Precision (CUDA): `autocast` plus `GradScaler` für `float16`. Erst eine korrekte `float32`-Baseline herstellen; AMP-Unterstützung hängt von Hardware und Operationen ab.

```
scaler = torch.amp.GradScaler("cuda")
# Pro Schritt, CUDA-Voraussetzung:
optimizer.zero_grad(set_to_none=True)
with torch.autocast("cuda",
                    dtype=torch.float16):
    logits = model(inputs)
    loss = loss_fn(logits, targets)
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Optional bei instabilen Gradienten: `clip_grad_norm_`. Mit AMP vorher `scaler.unscale_(optimizer)`. `torch.compile(model)` kann beschleunigen; Kompilierkosten und Kompatibilität prüfen.

18 Qualität, Design & Namen

Datenqualität: Train / Validation / Test trennen; Test erst am Ende. Normalisierung nur anhand Trainingsdaten bestimmen. Datenleckage und Klassenungleichgewicht beachten.

Reproduzierbarkeit: Seeds für verwendete Bibliotheken setzen. Gleicher Seed garantiert keine identischen Ergebnisse über Geräte und Versionen; deterministische Optionen separat prüfen.

Design: Daten, Modell und Trainingslogik trennen. Funktionen im echten Projekt mit `model`, `loader`, `optimizer`, `loss_fn` und `device` als Parametern statt globalen Abhängigkeiten schreiben.

Namen: `DigitClassifier`, `train_epoch()`, `evaluate()`, `predict()`. `inputs` / `targets` für Batchdaten; `logits` vor `Softmax`, `probabilities` danach; `predictions` für Klassen; `batch_size`, `learning_rate`, `num_classes`.

Dokumentieren: erwartete Shape, dtype, device und Wertebereich an Funktionsgrenzen. Bei Kennzahlen korrekt durch Beispielzahl teilen; Loss-Reduktion und Gewichte berücksichtigen.