

Training loop

☑ Trắc nghiệm 28 📧 Thực hành 2 🔗 Chạy trên Kaggle ⬇ PDF bài

Training loop là đoạn code lặp lại các bước: lấy một batch, tính dự đoán, tính loss, tính gradient, cập nhật trọng số. Viết cho loss giảm thì dễ; khó hơn là để kết quả dừng lại được. Các lỗi hay gặp: quên lưu mean/std dùng để chuẩn hóa, quên chuyển model sang chế độ eval nên dropout vẫn bật lúc dự đoán, chỉ lưu trọng số mà không lưu cấu hình model. Bài này viết một training loop đầy đủ, từ đọc dữ liệu tới lưu checkpoint, rồi nạp lại trong một chương trình mới và kiểm rằng dự đoán không đổi.

MỤC TIÊU

- Dùng Dataset, DataLoader; phân biệt batch, step, epoch.
- Viết training loop đúng thứ tự các bước.
- Chấm validation và lưu checkpoint tốt nhất.
- Nạp lại model trong chương trình mới và kiểm kết quả.

Một bài nhỏ để kiểm toàn bộ quy trình

Ta tự sinh 96 điểm hai đặc trưng; nhãn là 1 nếu $x_1 + 0.5x_2 > 0$, còn lại là 0. Quy tắc đã biết trước, nên nếu kết quả tệ thì lỗi nằm ở pipeline chứ không phải ở độ khó của dữ liệu. Chia train/validation trước; mean/std chỉ tính trên train và được dùng lại trên validation.

Python

```
from pathlib import Path
import json
import subprocess
import sys
import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader
```

```

torch.set_num_threads(1)
torch.manual_seed(21)
torch.use_deterministic_algorithms(True)
generator = torch.Generator().manual_seed(21)
raw_X = torch.randn(96, 2, generator=generator)
y = (raw_X[:, 0] + .5 * raw_X[:, 1] > 0).long()
order = torch.randperm(96, generator=generator)
train_id, valid_id = order[:72], order[72:]
mean = raw_X[train_id].mean(dim=0)
scale = raw_X[train_id].std(dim=0, unbiased=False).clamp_min(1e-6)
X = (raw_X - mean) / scale
train_data = TensorDataset(X[train_id], y[train_id])
valid_data = TensorDataset(X[valid_id], y[valid_id])
train_loader = DataLoader(train_data, batch_size=16, shuffle=True,
                           generator=torch.Generator().manual_seed(21), num_workers=0)
valid_loader = DataLoader(valid_data, batch_size=16, shuffle=False, num_workers=0)
print(len(train_data), len(valid_data), len(train_loader))
print(tuple(next(iter(valid_loader))[0]).shape)

```

Có 72 mẫu train, 24 validation và 5 batch mỗi epoch: bốn batch 16 mẫu, một batch 8 mẫu. Ba từ cần phân biệt:

- **Batch:** một nhóm mẫu dùng cho một lần tính gradient.
- **Step:** một lần optimizer cập nhật trọng số; mỗi batch một step.
- **Epoch:** một lần duyệt hết tập train.

Một epoch có $\lceil n/B \rceil$ step: 10 000 ảnh với batch 64 là 157 step; 1000 mẫu, batch 32, 5 epoch là $32 \cdot 5 = 160$ lần cập nhật.

TensorDataset gom các tensor cùng chiều đầu; dataset thực tế có thể tự đọc tệp ở `__getitem__` (mục cuối bài). DataLoader ghép các mẫu thành batch và quyết định thứ tự lấy. Với `shuffle=True`, thứ tự được xáo lại ở đầu mỗi epoch. Nếu tập train xếp sẵn hết mềo rồi mới tới chó mà không xáo, mỗi đoạn batch chỉ thấy một lớp, và mô hình bị kéo qua lại giữa hai lớp.

1 epoch = 72 mẫu train



batch cuối 8 mẫu vẫn là một lần cập nhật

72 mẫu, batch 16: 5 batch mỗi epoch, 20 epoch là 100 step.

Ở Windows, nhiều worker có thể tạo tiến trình con và yêu cầu bảo vệ code bằng `if __name__ == '__main__':`. Ví dụ đầu tiên dùng `num_workers=0` để dễ chạy nối tiếp trong notebook. Việc tăng worker nên dựa trên đo nút nghẽn, không phải mặc định “càng nhiều càng nhanh”.

HỎI NHANH

Train có 72 mẫu, batch size 16, không bỏ batch cuối. Train 20 epoch thì optimizer cập nhật bao nhiêu lần?

A 20

B 80

C 100

D 1440

Loop là một chu trình trạng thái

Mỗi batch đi qua năm bước theo đúng thứ tự:

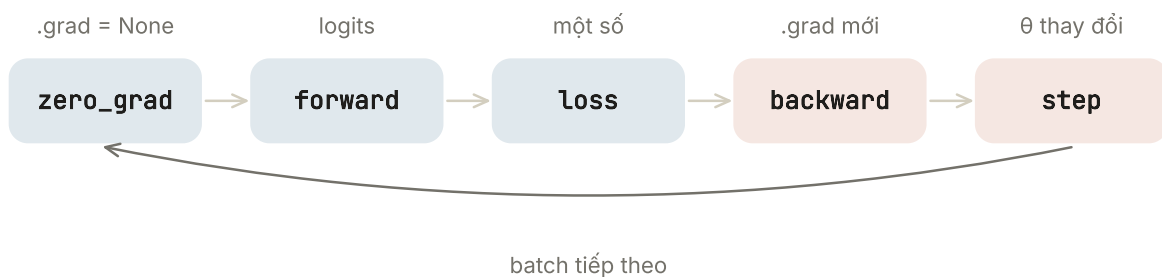
1. `zero_grad` xóa gradient cũ.
2. Forward dùng trọng số hiện tại để tạo logits.
3. Loss so logits với nhãn.

4. Backward tính gradient.

5. Step sửa trọng số.

Đảo step lên trước backward thì optimizer dùng gradient cũ hoặc không có gradient. Gọi `zero_grad` sau backward, trước step, thì xóa chính tín hiệu vừa tính. Nếu cần cắt gradient bằng `clip_grad_norm`, đặt nó sau backward và trước step.

Optimizer được tạo từ `model.parameters()`, tức các trọng số và bias sẽ được chỉnh; nó quyết định chỉnh theo cách nào (SGD, Adam ở bài 3.5).



đổi thứ tự backward / zero_grad / step là mất hoặc dùng sai gradient

Mỗi batch: `zero_grad` → `forward` → `loss` → `backward` → `step`.

Vì sao phải xóa? Backward không ghi đè `.grad` mà **cộng** gradient mới vào đó. Với hàm $\mathbf{x}^T \mathbf{w}$, gradient theo $\mathbf{w}$ chính là $\mathbf{x}$, nên dễ nhìn thấy điều này:

Python

```
w = torch.zeros(2, requires_grad=True)
for xi in [torch.tensor([1., 2.]), torch.tensor([3., -1.])]:
    (xi @ w).backward()
    print(w.grad.tolist())
```

Lần thứ hai `.grad` là $[1, 2] + [3, -1] = [4, 1]$, không phải $[3, -1]$. Thiếu `zero_grad`, mỗi step dùng tổng gradient của mọi batch đã qua, nên bước cập nhật lớn dần. Cộng dồn có chủ ý lại là một kỹ thuật: gradient accumulation (bài 3.7) gom gradient vài batch nhỏ rồi mới step một lần.

HỎI NHANH

Một vòng lặp gọi `loss.backward()`, rồi `optimizer.zero_grad(set_to_none=True)`, rồi `optimizer.step()`. Train loss đứng yên ở mức ban đầu qua mọi epoch, và không có lỗi nào. Vì sao?

- A Learning rate quá nhỏ nên mỗi bước gần như không đổi gì
- B Gradient bị xóa trước `step()`, nên không có gì để cập nhật
- C Gradient cộng dồn qua các batch nên các bước triệt tiêu nhau
- D Backward chưa chạy xong vì `zero_grad` được gọi ngay sau nó

Model có hai lớp linear xen ReLU. Loss nhận logits (B, 2) và nhãn long (B,). Ta gom code train một epoch thành một hàm. Loss báo cáo nhân batch size trước khi cộng, để batch cuối nhỏ không được tính ngang trọng lượng với batch đầy:

Python

```
def make_model():
    return nn.Sequential(nn.Linear(2, 8), nn.ReLU(), nn.Linear(8, 2))

model = make_model()
optimizer = torch.optim.Adam(model.parameters(), lr=.03)
criterion = nn.CrossEntropyLoss()

def train_epoch(model, loader, optimizer):
    model.train()
    loss_sum = 0.
    seen = 0
    for xb, yb in loader:
        optimizer.zero_grad(set_to_none=True)
        logits = model(xb)
        assert logits.shape == (len(yb), 2)
        loss = criterion(logits, yb)
        assert torch.isfinite(loss)
        loss.backward()
        optimizer.step()
        loss_sum += loss.item() * len(yb)
        seen += len(yb)
```

```

return loss_sum / seen

def evaluate(model, loader):
    model.eval()
    loss_sum = 0.
    correct = 0
    seen = 0
    with torch.no_grad():
        for xb, yb in loader:
            logits = model(xb)
            loss_sum += criterion(logits, yb).item() * len(yb)
            correct += (logits.argmax(dim=1) == yb).sum().item()
            seen += len(yb)
    return loss_sum / seen, correct / seen

print('initial validation:', tuple(round(v, 4) for v in evaluate(model, valid_loader)))

```

Trước khi học, accuracy $0.4583 = 11/24$, gần mức đoán bừa của hai lớp, và loss 0.7394 gần $\ln 2 \approx 0.693$, loss của một model chia đều xác suất cho hai lớp. Đó là điểm xuất phát đúng. Với K lớp, loss ban đầu lệch xa $\ln K$ là dấu hiệu nên kiểm khởi tạo hoặc nhẵn trước khi train.

Hai assert trong `train_epoch` là hai phép kiểm rẻ: logits đúng shape $(B, 2)$ và loss hữu hạn. Loss thành nan thường do learning rate quá lớn làm trọng số phân kỳ, hoặc do phép log, phép chia không an toàn trong loss tự viết; assert dừng loop ngay ở batch đầu tiên gặp lỗi, trước khi nan lan vào trọng số.

`model.train()` và `model.eval()` điều khiển hành vi của các layer như dropout, BatchNorm (bài 3.6). `no_grad()` điều khiển việc ghi graph. Eval không tự tắt gradient; `no_grad` không tự tắt dropout. Mạng nhỏ này chưa có hai layer đó, nhưng vẫn giữ hai lời gọi để loop dùng được khi kiến trúc thay đổi.

Training loss trong hàm là trung bình loss quan sát **trong khi trọng số thay đổi** qua các batch. Nó không nhất thiết bằng loss đánh giá lại toàn bộ train ở checkpoint cuối epoch. Nếu cần so train/validation chặt chẽ, gọi cùng `evaluate` trên cả hai loader sau epoch, dùng cùng chế độ layer.

HỎI NHANH

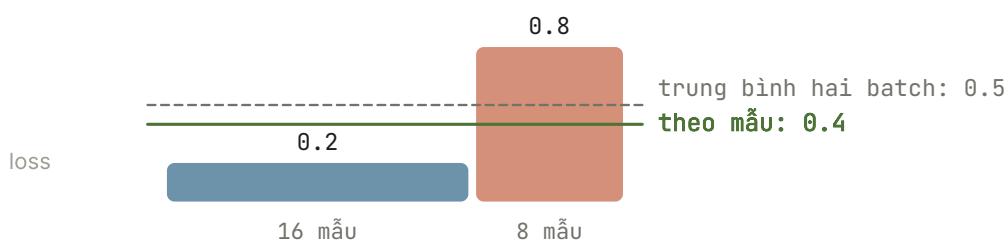
Model có lớp Dropout, tức lớp tắt ngẫu nhiên một số nơ-ron trong lúc train (bài 3.6 sẽ học kỹ). Hàm chấm validation có `with torch.no_grad()`: nhưng quên `model.eval()`. Chuyện gì xảy ra?

- A Dropout vẫn tắt ngẫu nhiên nơ-ron, nên điểm dao động
- B `no_grad` tự chuyển model sang eval, nên vẫn đúng
- C Graph vẫn được ghi lại, vì model còn ở chế độ train
- D Báo lỗi, vì Dropout cần gradient mới chạy được

Cộng loss qua các batch cũng phải tính theo mẫu. Giả sử batch đầu có 16 mẫu, loss mean 0.2; batch cuối có 8 mẫu, loss mean 0.8. Trung bình hai số bằng 0.5, nhưng trung bình theo 24 mẫu là $(16 \cdot 0.2 + 8 \cdot 0.8)/24 = 0.4$. Đó là lý do code nhân `len(yb)` trước khi cộng.

Cách này đúng với cross-entropy không trọng số của bài. Khi dùng class weight, `ignore_index` hoặc loss trung bình theo pixel, mẫu số đúng là tổng trọng số hoặc số phần tử thật sự được tính, nên cần đọc `reduction` của loss trước khi cộng.

`CrossEntropyLoss` mặc định bỏ qua nhãn `-100` (`ignore_index`) và chia cho số nhãn còn lại. Đổi `reduction='mean'` sang `'sum'` với batch 64 mà giữ learning rate của SGD là nhân bước đi lên 64 lần.



Trung bình theo mẫu là 0.4, không phải 0.5.

HỎI NHANH

Một epoch có bốn batch 16 mẫu, batch nào cũng có loss trung bình 0.1, và một batch cuối 8 mẫu có loss trung bình 0.7. Loss trung bình theo mẫu của cả epoch là bao nhiêu?

A 0.10

B 0.17

C 0.22

D 0.40

Chọn checkpoint theo validation

Nếu chỉ lưu epoch cuối, ta có thể bỏ mất model đã tổng quát hóa tốt hơn ở giữa quá trình. Ta chọn loss validation nhỏ nhất; metric lựa chọn trong bài thật có thể là F1, RMSE hoặc metric của scorer. Chốt hướng “nhỏ hơn tốt” hay “lớn hơn tốt” trước khi viết điều kiện so sánh.

Lưu bản tốt nhất phải lưu bản sao. Các tensor trong `model.state_dict()` dùng chung bộ nhớ với tham số; chỉ gán `best_state = model.state_dict()` rồi tiếp tục học thì “bản tốt nhất” đổi theo model. Dùng clone hoặc deepcopy, hoặc ghi tệp ngay khi gặp bản tốt hơn.

Python

```
best_loss = float('inf')
best_epoch = -1
best_state = None
history = []
for epoch in range(35):
    training_loss = train_epoch(model, train_loader, optimizer)
    valid_loss, valid_accuracy = evaluate(model, valid_loader)
    history.append((epoch + 1, training_loss, valid_loss, valid_accuracy))
    if valid_loss < best_loss:
        best_loss = valid_loss
```

```

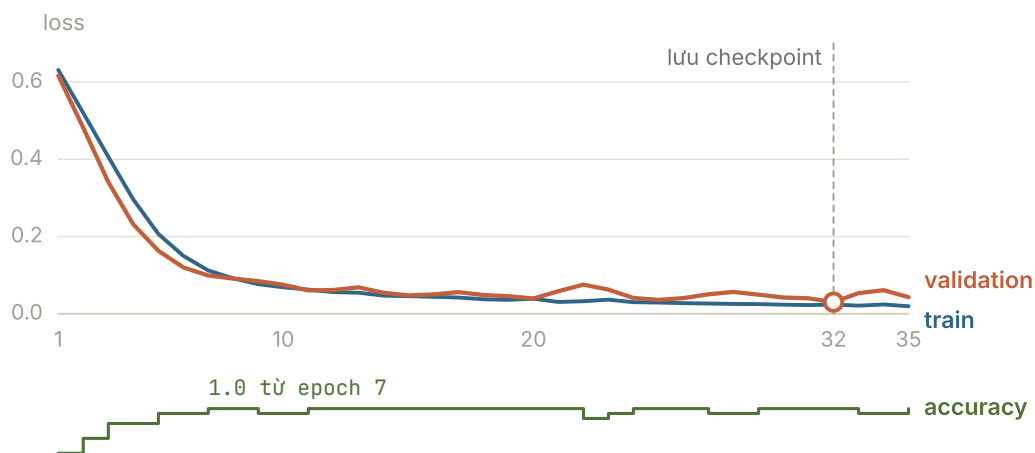
best_epoch = epoch + 1
best_state = {name: value.detach().cpu().clone()
               for name, value in model.state_dict().items()}
if epoch in [0, 9, 19, 34]:
    print(epoch + 1, round(training_loss, 4),
          round(valid_loss, 4), round(valid_accuracy, 4))

perfect = [e for e, _, _, accuracy in history if accuracy == 1.0]
print('accuracy 1.0:', len(perfect), 'epochs, first', perfect[0])
model.load_state_dict(best_state)
model.eval()
print('best epoch:', best_epoch)
print('best validation:', tuple(round(v, 4) for v in evaluate(model, valid_loader)))

```

Mỗi dòng epoch có train loss, validation loss và validation accuracy. Train loss giảm gần như đều từ 0.6312 xuống 0.0197, còn validation loss thấp nhất ở epoch 32 (0.03) rồi lên 0.0429 ở epoch 35: epoch cuối không phải epoch tốt nhất.

Validation chỉ 24 mẫu nên accuracy đi theo nấc 1/24; 21 trong 35 epoch cùng đạt 1.0, accuracy không phân biệt được chúng, còn loss thì phân biệt được. Điểm (0.03, 1.0) đã được dùng để chọn epoch, nên nó hơi lạc quan so với dữ liệu mới.



Validation loss thấp nhất ở epoch 32, nên lưu checkpoint ở đó.



bước 0 (epoch 0.0): accuracy
validation 0.142, loss 2.313



bước 10 (epoch 0.1): accuracy
validation 0.577, loss 1.658



bước 80 (epoch 0.9): accuracy
validation 0.757, loss 0.719



bước 1880 (epoch 20.0):
accuracy validation 0.852, loss
0.420

Vòng train trên Fashion-MNIST: viền xanh đoán đúng, viền cam đoán sai.

HỎI NHANH

Code chọn checkpoint ghi `best_state = model.state_dict()`, không clone, mỗi khi validation loss tốt hơn. Train hết 35 epoch rồi `model.load_state_dict(best_state)`. Model có trọng số của epoch nào?

- A Epoch 32, epoch có validation loss nhỏ nhất
- B Epoch 1, lần đầu điều kiện được thỏa
- C Không xác định, vì tensor cũ đã bị giải phóng
- D Epoch 35, vì `best_state` thay đổi theo model

HỎI NHANH

Giả sử vòng lặp trên chọn checkpoint theo validation accuracy, với điều kiện `valid_accuracy > best_accuracy` và `best_accuracy` khởi tạo bằng 0, thay vì theo loss. Checkpoint là epoch nào?

- A Epoch 7, lần đầu accuracy đạt 1.0
- B Epoch 20, mốc log đầu tiên có accuracy 1.0
- C Epoch 32, cũng là epoch có loss nhỏ nhất
- D Epoch 35, lần cuối accuracy đạt 1.0

Lưu đủ để suy luận lại

Đưa đầu vào mới qua model đã học để lấy dự đoán gọi là suy luận (inference), khác với huấn luyện là lúc trọng số còn được chỉnh. Để suy luận lại ở một chương trình khác, ta lưu một gói (packet) gồm `state_dict`, `mean/std` và vài đầu vào kiểm tra kèm logits mong

đội, vào thư mục receipts/phase-3 cạnh notebook. Muốn tiếp tục training chính xác thì còn cần trạng thái optimizer, scheduler, số epoch và trạng thái bộ sinh số ngẫu nhiên.

`torch.load` một tệp lạ có thể chạy code tùy ý, nên chỉ tải checkpoint đáng tin cậy; `weights_only=True` chỉ cho phép tensor và kiểu dữ liệu cơ bản, đủ cho packet này.

Python

```
artifact_dir = Path('receipts/phase-3')
artifact_dir.mkdir(parents=True, exist_ok=True)
checkpoint_path = artifact_dir / '3-4-best.pt'
query = torch.tensor([[1., 0.], [-1., 0.]])
with torch.no_grad():
    expected = model((query - mean) / scale)
torch.save({'state': best_state, 'mean': mean, 'scale': scale,
            'query': query, 'expected': expected, 'epoch': best_epoch}, checkpoint_path)
fresh = make_model()
packet = torch.load(checkpoint_path, map_location='cpu', weights_only=True)
fresh.load_state_dict(packet['state'])
fresh.eval()
with torch.no_grad():
    reloaded = fresh((packet['query'] - packet['mean']) / packet['scale'])
print('same logits:', torch.allclose(reloaded, expected, atol=1e-7))
print('classes:', reloaded.argmax(dim=1).tolist())
```

True kiểm khôi phục trong cùng tiến trình; hai lớp $[1, 0]$ khớp quy tắc ban đầu: $(1, 0)$ cho $x_1 + 0.5x_2 = 1 > 0$, $(-1, 0)$ cho $-1 < 0$.

HỎI NHANH

Phiên Colab bị ngắt ở epoch 20, và checkpoint chỉ có `state_dict` của model. Ta tạo một Adam mới rồi train tiếp 15 epoch. So với chạy liền 35 epoch thì sao?

- A Giống hệt, vì trọng số model đã mô tả đủ trạng thái
- B Giống hệt, nếu đặt lại seed như lúc đầu
- C Khác, vì các trung bình động của Adam và thứ tự xáo dữ liệu đều bắt đầu lại
- D Báo lỗi, vì `load_state_dict` đòi cả state của optimizer

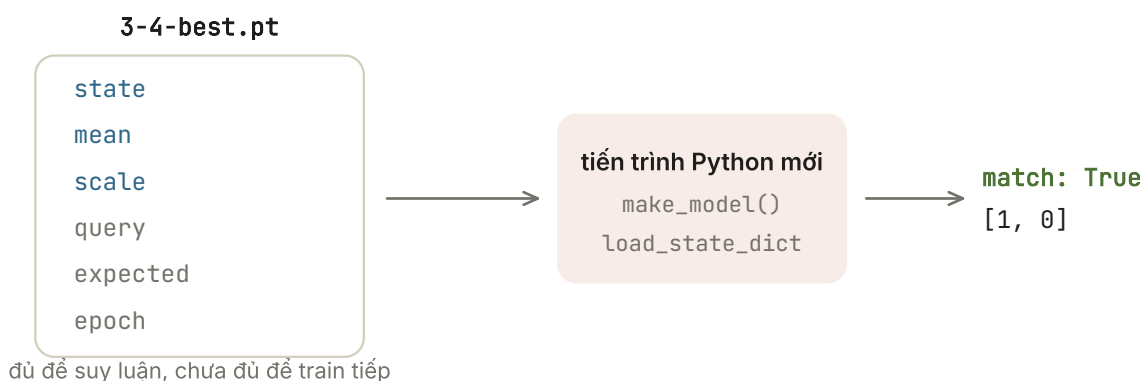
Phép kiểm trong cùng tiến trình chưa cho biết notebook có dựa vào biến cũ hay không, nên ta tạo một script nhỏ chạy bằng Python mới. Script tự tạo kiến trúc, đọc checkpoint và chỉ dùng thông tin trong packet:

Python

```
inference_source = '''import sys
import torch
from torch import nn
torch.set_num_threads(1)
packet = torch.load(sys.argv[1], map_location="cpu", weights_only=True)
model = nn.Sequential(nn.Linear(2, 8), nn.ReLU(), nn.Linear(8, 2))
model.load_state_dict(packet["state"])
model.eval()
with torch.no_grad():
    x = (packet["query"] - packet["mean"]) / packet["scale"]
    logits = model(x)
print("new process match:", torch.allclose(logits, packet["expected"], atol=1e-7))
print("new process classes:", logits.argmax(dim=1).tolist())
'''

inference_path = artifact_dir / '3-4-infer.py'
inference_path.write_text(inference_source, encoding='utf-8')
result = subprocess.check_output([sys.executable, '-B', '-X', 'utf8',
    str(inference_path), str(checkpoint_path)], text=True)
print(result.rstrip())
```

Phép kiểm này giống chạy lời giải CP trên input mới qua stdin: kết quả không được phụ thuộc vào biến còn sót trong môi trường thử. Ở bài thi thật, script suy luận còn phải theo đúng quy định nộp bài của đề (format file, thứ tự ID, tệp đề cho), vì vậy phép kiểm ở phiên mới phải chạy đúng script sẽ nộp.



Checkpoint lưu trọng số, mean và scale, kèm đầu vào kiểm và logits mong đợi.

HỎI NHANH

Trong notebook, model mới nạp checkpoint đã cho `same_logits: True`. Vì sao bài vẫn chạy thêm một script ở tiến trình Python mới?

- A `torch.load` chỉ đọc đúng tệp khi chạy ở tiến trình khác
- B Tiến trình mới đổi seed nên phép so khớp khắt khe hơn
- C `allclose` trong cùng tiến trình luôn trả về True
- D Notebook có thể đang dựa vào biến cũ không có trong packet

Tái lập

Seed của khởi tạo model, chia dữ liệu và DataLoader cần được cố định. Bài đặt seed, dùng CPU một thread và chế độ deterministic, nên chạy lại trong cùng môi trường cho cùng số. PyTorch không hứa kết quả giống từng bit giữa mọi phiên bản, nền tảng và GPU; vì vậy ghi lại phiên bản thư viện, cách chia và cấu hình cùng điểm số.

Tái lập được một lần chạy chưa có nghĩa là một lần chạy đủ để so hai cấu hình. Ngay trong log trên, validation loss đi từ 0.0396 (epoch 20) xuống 0.03 (epoch 32) rồi lên 0.0429 (epoch 35). Chênh lệch giữa hai cấu hình chỉ đáng tin khi lớn hơn rõ mức dao động giữa các seed.

HỎI NHANH

Thử learning rate 0.03 và 0.01, mỗi cấu hình chạy một lần với cùng seed 21. Validation loss tốt nhất lần lượt là 0.030 và 0.028. Kết luận nào hợp lý?

- A Chọn 0.01, vì cùng seed nên chênh lệch chỉ do learning rate
- B Chọn 0.03, vì chênh 0.002 quá nhỏ nên giữ cấu hình cũ
- C Chưa chọn được, cần chạy thêm vài seed rồi so trung bình
- D Chọn 0.01, vì validation loss thấp hơn là đủ để quyết định

Tự viết Dataset và collate_fn

TensorDataset cần cả bộ dữ liệu nằm sẵn trong một tensor. Thư mục hàng nghìn ảnh, mỗi ảnh một cỡ, thì nên viết một lớp con của Dataset với hai hàm ở bài 0.1: `__len__` trả số mẫu, `__getitem__(i)` đọc **một** mẫu từ đĩa khi cần.

Python

```
import numpy as np
import pandas as pd
from PIL import Image
from tempfile import TemporaryDirectory
from torch.utils.data import Dataset

class FolderDataset(Dataset):
    def __init__(self, root, table, size=8):
        self.root, self.table, self.size = Path(root), table.reset_index(drop=True), size

    def __len__(self):
        return len(self.table)

    def __getitem__(self, i):
        row = self.table.iloc[i]
        image = Image.open(self.root / row["file"]).convert("L").resize((self.size,
self.size))
        x = torch.from_numpy(np.asarray(image, dtype=np.float32) / 255).unsqueeze(0) # (1,
```

```

H, W)
    return x, int(row["label"])

with TemporaryDirectory() as directory:
    root = Path(directory)
    rng = np.random.default_rng(0)
    rows = []
    for i in range(10):
        w, h = rng.integers(10, 30, size=2)  # mỗi ảnh một kích thước
        label = i % 2  # nhãn 1: ảnh sáng hơn
        pixels = rng.integers(0, 120, size=(h, w)) + 120 * label
        Image.fromarray(pixels.astype(np.uint8)).save(root / f"{i}.png")
        rows.append({"file": f"{i}.png", "label": label})
    data = FolderDataset(root, pd.DataFrame(rows))
    loader = DataLoader(data, batch_size=4, shuffle=False, num_workers=0)
    xb, yb = next(iter(loader))
    print(len(data), len(loader), tuple(data[0][0].shape), tuple(xb.shape), yb.tolist())
    print(round(float(xb[yb == 0].mean()), 3), round(float(xb[yb == 1].mean()), 3))

```

`__getitem__` đưa mọi ảnh về (1, 8, 8), nên `DataLoader` xếp bốn mẫu thành batch (4, 1, 8, 8); 10 mẫu, batch 4 cho 3 batch. Độ sáng 0.236 và 0.709 khớp cách sinh: ảnh nhãn 1 sáng hơn 120 mức. Đổi cỡ, chuẩn hóa, augmentation (chương 5) đều đặt trong `__getitem__`.

Văn bản và âm thanh mỗi mẫu một độ dài. `DataLoader` mặc định ghép bằng `torch.stack` nên báo lỗi; `collate_fn` tự viết đệm chuỗi ngắn bằng 0 và giữ độ dài thật.

Python

```

from torch.nn.utils.rnn import pad_sequence

pairs = [(torch.tensor([5, 2, 9]), 1), (torch.tensor([7]), 0), (torch.tensor([3, 3, 1, 4]),
1)]
try:
    next(iter(DataLoader(pairs, batch_size=3)))
except RuntimeError as error:
    print("không có collate_fn:", str(error).split(" at entry")[0])

def pad_collate(batch):
    seqs, labels = zip(*batch)
    lengths = torch.tensor([len(s) for s in seqs])
    padded = pad_sequence(seqs, batch_first=True, padding_value=0)
    return padded, lengths, torch.tensor(labels)

```

```
for tokens, lengths, labels in DataLoader(pairs, batch_size=3, collate_fn=pad_collate):
    print(tokens)
    print(lengths.tolist(), labels.tolist())
```



`collate_fn` đệm 0 cho bằng chuỗi dài nhất và giữ độ dài thật.

`collate_fn` nhận list các mẫu do `__getitem__` trả về và trả về một batch. Số 0 là token đệm; tính loss trên cả ma trận mà không dùng `lengths` hay mặt nạ thì model học cả phần đệm. Bài 3.8 và chương 6 dùng khuôn này kèm `mask`.

`collate_fn` cũng là chỗ gọn để bỏ mẫu hỏng: nếu `__getitem__` trả `None` cho ảnh không đọc được, lọc các `None` trước khi gom batch; tốt hơn nữa là lọc danh sách tệp hỏng trước khi huấn luyện.

HỎI NHANH

Dataset trả về ảnh khác cỡ nhau, `DataLoader batch_size=32`, không có `collate_fn`. Lấy batch đầu tiên thì sao?

- A Mọi ảnh được đổi về cỡ của ảnh đầu tiên
- B Mọi ảnh được cắt về cỡ nhỏ nhất trong batch
- C Báo lỗi vì không xếp được các tensor khác shape
- D Batch là một list ảnh rời

Khi đi thi

- Chạy thử vài batch, đo thời gian một epoch rồi nhân với số epoch để biết có kịp giờ không.
- Lưu checkpoint ra tệp ngay khi validation tốt hơn, theo đúng metric của đề.
- Checkpoint phải đi kèm scaler, thứ tự feature và bảng tên lớp để suy luận lại được.

Tóm tắt

- Mỗi batch: `zero_grad` → `forward` → `loss` → `backward` → `step`. Epoch là một lần duyệt hết train, gồm nhiều step.
- Chuẩn hóa bằng thống kê của train; validation chạy trong `eval()` và `no_grad()`.
- Chọn checkpoint theo validation, lưu bản sao; kiểm suy luận lại ở một tiến trình Python mới.
- Dữ liệu thật vào loop qua Dataset tự viết; mẫu dài ngắn khác nhau thì cần `collate_fn`.

Kiểm tra cuối bài

CÂU 1

Tập train có N mẫu, mỗi mẫu d đặc trưng, mô hình tuyến tính. So chi phí tính toán của **một epoch** SGD, mỗi bước dùng một mẫu, với **một vòng** gradient descent dùng cả tập. Câu nào đúng?

- A Cùng bậc $O(Nd)$, vì cả hai đều đi qua N mẫu một lần
- B Epoch SGD đắt hơn N lần, vì có N lần cập nhật
- C Epoch SGD rẻ hơn N lần, vì mỗi bước chỉ nhìn một mẫu
- D Không so được, vì thứ tự mẫu trong SGD là ngẫu nhiên

PHÒNG THEO CMU 10-601 · Exam 2 Practice · Q1.4

CÂU 2

Một vòng lặp quên hẳn `optimizer.zero_grad()`. Tham số w bắt đầu bằng 1, optimizer là SGD với learning rate 0.1, và ở batch nào gradient của loss theo w cũng bằng 2. Sau 3 batch, w và $w.grad$ bằng bao nhiêu?

- A $w = 0.4$, $w.grad$ bằng 2
- B $w = 0.4$, $w.grad$ bằng 6
- C $w = -0.2$, $w.grad$ bằng 2
- D $w = -0.2$, $w.grad$ bằng 6

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 126

CÂU 3

Loss train in ra sau mỗi 50 bước là 0.69, 0.48, 1.9, 27.3, rồi nan. Dữ liệu đã chuẩn hóa, loss là `nn.CrossEntropyLoss`, optimizer là Adam với learning rate 1.0. Nên kiểm gì trước?

- A Batch size, tăng lên cho loss bớt nhiều
- B Learning rate, giảm 10 lần rồi xem độ lớn của gradient
- C Số epoch, dừng sớm trước khi loss thành nan
- D Kiến trúc, thêm lớp để mạng học tiếp được

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 141

CÂU 4

Dòng log epoch 1 của bài có train loss 0.6312 và validation loss 0.6167. Validation thấp hơn train có phải dấu hiệu lỗi không?

- A Có, validation thấp hơn nghĩa là dữ liệu đã lọt vào train
- B Có, phải gọi `model.eval()` khi train để hai số so được với nhau
- C Không, vì train loss được lấy trong lúc trọng số còn đang học
- D Không, vì ở epoch đầu validation luôn thấp hơn train

CÂU 5

Hàm chấm validation có `model.eval()` nhưng không có `no_grad`, và gom `all_logits.append(model(xb))` để tính F1 sau vòng lặp. Bộ nhớ tăng theo từng batch, rồi báo hết bộ nhớ giữa chừng. Sửa thế nào?

- A Gọi thêm `model.eval()` để tắt hẳn autograd
- B Bọc vòng chấm trong `with torch.no_grad():`
- C Gọi `optimizer.zero_grad()` trước mỗi batch chấm
- D Giảm batch size khi train để dành chỗ cho validation

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 127

CÂU 6

Lưu MLP của bài, gồm `Linear(2, 8)`, `ReLU`, `Linear(8, 2)`, bằng `torch.save(model.state_dict(), 'best.pt')`. Tập này chứa gì?

- A 4 tensor, tổng cộng 42 số: weight và bias của hai lớp Linear
- B 6 tensor, thêm một mục cho ReLU và một mục cho kiến trúc
- C 4 tensor trọng số kèm `.grad` để train tiếp được
- D Cả object model, nạp lại không cần dựng kiến trúc

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 134

CÂU 7

Đổi tiêu chí chọn checkpoint từ validation loss sang macro-F1, nhưng vẫn giữ `best = float('inf')` và điều kiện `if f1 < best:`. Checkpoint được giữ là gì?

- A Epoch cuối, vì F1 luôn nhỏ hơn `inf`
- B Không có, vì F1 không so được với `inf`
- C Epoch có F1 cao nhất, đúng như mong muốn
- D Epoch có F1 thấp nhất trong lúc train

CÂU 8

Script nộp bài đọc `test.csv` có cùng tên cột với `train` nhưng khác thứ tự cột, rồi chuẩn hóa bằng `mean` và `std` tính lại trên chính `test`. Model là MLP như trong bài. Phải sửa thế nào?

- A Theo thứ tự cột của `test`, `mean` và `std` tính lại trên `test`
- B Theo thứ tự cột của `test`, `mean` và `std` đã lưu từ `train`
- C Theo thứ tự cột đã lưu, `mean` và `std` đã lưu từ `train`
- D Theo thứ tự cột đã lưu, `mean` và `std` tính lại trên `test`

CÂU 9

Phân loại 10 lớp cân bằng, validation accuracy trong notebook là 0.91. Script nộp chạy ở phiên mới, nạp đúng checkpoint, chuẩn hóa bằng mean và std đã lưu, số hàng và thứ tự ID khớp file mẫu. Vậy mà điểm chấm chỉ khoảng 0.10. Giả thuyết nào khớp nhất?

- A Model overfit validation, cần thêm regularization rồi train lại
- B Bảng đổi chỉ số ra tên lớp ở script nộp khác với lúc train
- C Test lệch phân phối nên scaler cần fit lại trên test
- D Phiên mới có seed khác nên dự đoán thành ngẫu nhiên

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 284

CÂU 10

Bài đặt seed, chạy trên CPU với một thread, và bật `use_deterministic_algorithms`. Như vậy bảo đảm được điều gì?

- A Chạy lại trong cùng môi trường thì cho cùng kết quả
- B Kết quả giống từng bit trên mọi máy và mọi phiên bản
- C Dựng lại được model kể cả khi không lưu cách chia dữ liệu
- D Không cần validation nữa, vì kết quả đã cố định

Đọc thêm

- [PyTorch — Datasets & DataLoaders](#): đọc cách chia trách nhiệm dataset, sampler và ghép batch.
- [PyTorch — Optimizing Model Parameters](#): đối chiếu train/test loop và thứ tự gradient.

- [PyTorch — Saving and Loading Models](#): xem state_dict, checkpoint và lưu best model bằng bản sao.
- [PyTorch 2.9 — Reproducibility](#): đọc giới hạn của seed và deterministic algorithms.