

Tensor và autograd

☑ Trắc nghiệm 28 📧 Thực hành 1 ⬇ PDF bài

PyTorch là thư viện dùng để xây và huấn luyện mạng neural trong khóa học. Dữ liệu trong PyTorch là tensor: mảng nhiều chiều giống mảng NumPy, nhưng chạy được trên GPU. Điểm mới quan trọng hơn là autograd: PyTorch ghi lại các phép tính đã làm để tự tính đạo hàm của loss theo từng trọng số, thay cho việc tính tay như ở bài 0.6. Bài này tính một gradient bằng tay trước, rồi kiểm rằng autograd cho cùng kết quả.

MỤC TIÊU

- Đoán shape của tensor trước khi chạy code.
- Phân biệt nhân từng phần tử, nhân ma trận, reshape và đổi chiều.
- Tính gradient bằng tay và kiểm bằng autograd.
- Dùng detach, no_grad; chuyển tensor giữa CPU và GPU.

Từ mảng C++ tới tensor

Tensor là một khối số có shape, kiểu dữ liệu (dtype) và nơi lưu (device). Một số có shape `()`, một vector ba số có shape `(3,)`, bảng hai hàng ba cột có shape `(2, 3)`. Phép tính nào cũng đòi ba thứ này khớp nhau, giống kích thước mảng và kiểu số trong một lời giải CP. Tensor không tự biết chiều nào là học sinh, token hay đặc trưng; người viết code phải tự nhớ.

Trọng số và đặc trưng thường dùng `float32`; mã lớp hoặc token ID dùng `int64`, còn gọi là `torch.long`. Tensor số nguyên không thể đặt `requires_grad=True`, vì đạo hàm chỉ có nghĩa trên số thực. `torch.tensor` suy dtype từ dữ liệu, nên chỉ một dấu chấm thập phân cũng đổi kiểu của cả tensor. Ví dụ dưới dùng số nhỏ để tính tay được: một bảng hai mẫu, chiều sang hai đặc trưng mới.

Python

```
import torch

torch.set_num_threads(1)
torch.manual_seed(7)
X = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
W = torch.tensor([[1., 0.], [0., 1.], [1., 1.]])
b = torch.tensor([0.5, -0.5])
Y = X @ W + b
print(tuple(X.shape), X.dtype, X.device)
print(Y.tolist())
print(torch.tensor([1, 2, 3]).dtype, torch.tensor([1., 2., 3.]).dtype)
```

Dòng đầu cho cả shape, dtype và device. Dòng thứ hai gồm hai hàng vì ta vẫn có hai mẫu. Với hàng đầu, đầu ra thứ nhất là $1 + 3 + 0.5 = 4.5$; đầu ra thứ hai là $2 + 3 - 0.5 = 4.5$. Bias (2,) được cộng vào từng hàng. @ gộp chiều đặc trưng, còn $X * X$ chỉ bình phương từng ô. Dòng cuối cho thấy dtype đi theo dữ liệu: [1, 2, 3] thành int64, [1., 2., 3.] thành float32.

$$\begin{array}{c}
 \text{X } (2, 3) \\
 \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}
 \end{array}
 @
 \begin{array}{c}
 \text{W } (3, 2) \\
 \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}
 \end{array}
 +
 \begin{array}{c}
 \text{b } (2,) \\
 \begin{bmatrix} 0.5 & -0.5 \end{bmatrix}
 \end{array}
 =
 \begin{array}{c}
 \text{Y } (2, 2) \\
 \begin{bmatrix} 4.5 & 4.5 \\ 10.5 & 10.5 \end{bmatrix}
 \end{array}$$

$$1 \cdot 1 + 2 \cdot 0 + 3 \cdot 1 + 0.5 = 4.5$$

chiều 3 bị gộp; hai mẫu giữ nguyên

$X @ W + b$: chiều 3 bị gộp, hai mẫu giữ nguyên.

Đại lượng	Ý nghĩa	Shape
X	B mẫu, mỗi mẫu d đặc trưng	(B, d)
W	d đầu vào thành h đầu ra	(d, h)
b	Bias cho từng đầu ra	(h,)
$X @ W + b$	Đặc trưng mới của từng mẫu	(B, h)

HỎI NHANH

X có shape (8, 3), W có shape (3, 5). X @ W có shape gì?

A (3, 3)

B (5, 8)

C (8, 5)

D (8, 3, 5)

Shape đúng phải đi cùng ý nghĩa đúng

Khi xử lý chuỗi, tensor thường là (B, L, d): batch, độ dài, đặc trưng. reshape chia lại cách nhìn một dãy phần tử; nó không đổi thứ tự logic của dãy. transpose hoặc permute mới hoán đổi các trục. Đây giống khác biệt giữa đánh lại chỉ số của mảng một chiều và chuyển vị thật một bảng hai chiều.

Python

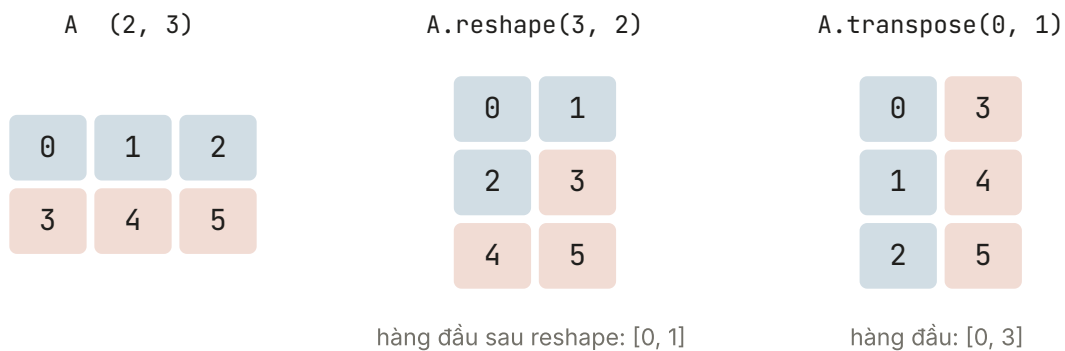
```
A = torch.arange(6).reshape(2, 3)
print(A.tolist())
print(A.reshape(3, 2).tolist())
print(A.transpose(0, 1).tolist())
print(A.unsqueeze(0).shape, A.sum(dim=1).shape)
```

So hai bảng (3, 2): reshape gom hai phần tử liên tiếp thành một hàng; transpose lấy các cột cũ làm hàng mới. Shape bằng nhau không bảo đảm dữ liệu mang cùng nghĩa.

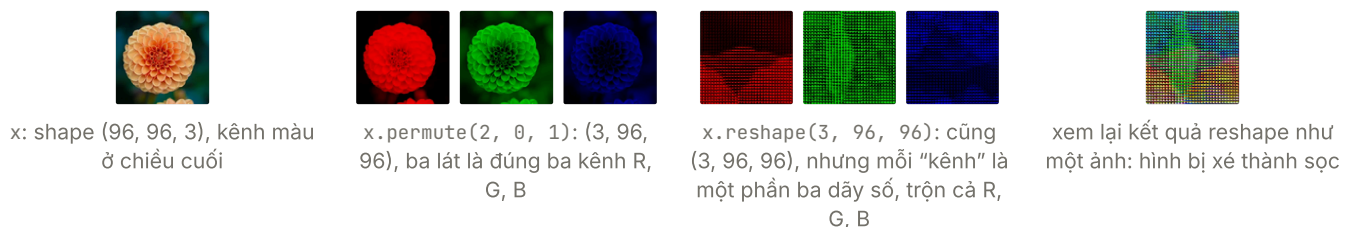
unsqueeze(0) thêm chiều batch kích thước 1; sum(dim=1) gộp ba cột của mỗi hàng. squeeze(0) chỉ xóa chiều 0 nếu nó bằng 1. Tránh dùng squeeze() không chỉ định chiều khi batch có thể bằng 1: chiều batch có thể mất ngoài ý muốn.

Vài quy ước gặp liên tục:

- Một batch ảnh màu có shape (N, C, H, W): 32 ảnh RGB 64×64 là (32, 3, 64, 64). Thư viện ảnh thường trả (H, W, 3), nên phải `permute(2, 0, 1)` trước khi thêm chiều batch.
- Gộp theo chiều nào thì chiều đó biến mất: logits (32, 10) lấy nhãn bằng `argmax(dim=1)`, còn `dim=0` cho 10 số, mỗi số là ảnh có điểm cao nhất của một lớp.
- `torch.cat` nối theo một chiều có sẵn: hai tensor (3, 4) thành (6, 4). `torch.stack` tạo chiều mới: thành (2, 3, 4).



Cùng shape (3, 2): reshape giữ thứ tự, transpose đưa cột thành hàng.



Đổi (H, W, 3) sang (3, H, W): permute giữ đúng kênh màu, reshape trộn lẫn.

HỎI NHANH

Bảng A có shape (2, 3). Muốn đổi thành (3, 2) sao cho các cột cũ thành hàng mới, ta dùng gì?

A `A.reshape(3, 2)`

B `A.view(3, 2)`

C `A.unsqueeze(0)`

D `A.transpose(0, 1)`

Broadcasting cũng cần kiểm. Phép cộng (B, h) với (h,) có ý nghĩa bias; phép trừ dự đoán (B, 1) với nhãn (B,) lại tạo (B, B). Máy không hiểu ta định so từng mẫu. Đặt `assert prediction.shape == target.shape` trước loss hồi quy là một kiểm tra rẻ nhưng hữu ích.

$$\begin{array}{c} \text{pred (2, 1)} \\ \begin{bmatrix} 1 \\ 2 \end{bmatrix} \end{array} - \begin{array}{c} \text{target (2,)} \\ \begin{bmatrix} 1 & 2 \end{bmatrix} \end{array} = \begin{array}{c} (2, 2) \\ \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \end{array}$$

$\text{mean}((\cdot)^2) = 0.5$
MSE đúng = 0

mỗi dự đoán bị trừ cho mọi nhãn; chỉ đường chéo là cặp đúng

(2, 1) trừ (2,) thành bảng (2, 2), nên MSE ra 0.5 thay vì 0.

HỎI NHANH

Dự đoán `pred = [[1.], [2.]]`, `shape (2, 1)`, khớp đúng nhãn `target = [1., 2.]`, `shape (2,)`. Vậy mà `((pred - target) ** 2).mean()` lại in ra `0.5`. Vì sao?

- A `float32` làm tròn 1 và 2 nên hiệu khác 0
- B Phép trừ bị broadcast thành `(2, 2)`, so mọi cặp với nhau
- C Nhãn phải có kiểu `long` thì phép trừ mới đúng
- D `mean()` chia thêm cho chiều thứ hai

Autograd lưu đường đi của phép tính

Trong bài linear regression, ta tự viết gradient. Với mạng nhiều lớp, biểu thức dài hơn nhưng mỗi phép cục bộ vẫn nhỏ: nhân, cộng, bình phương, ReLU. Autograd ghép đạo hàm cục bộ bằng quy tắc hàm hợp. Nó tính đạo hàm của phép tính đã thực hiện, không tìm công thức tối ưu và không tự sửa trọng số. So với sai phân hữu hạn ở bài 2.1, autograd cho đạo hàm chính xác và chỉ cần một lượt ngược cho mọi tham số, thay vì mỗi tham số một lượt chạy.

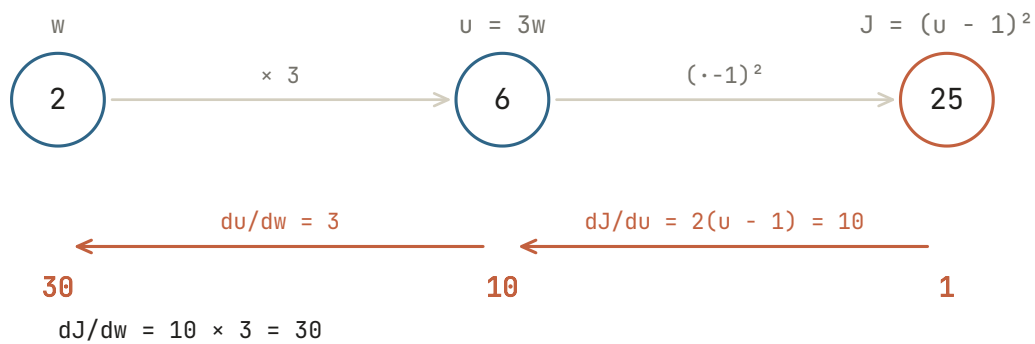
Hình dung phép tính như một DAG trong CP. Forward tính các đỉnh theo thứ tự topo, lưu giá trị cần thiết. Backward đi ngược DAG, truyền độ nhạy của loss về các đầu vào. Nếu một biến góp mặt ở nhiều nhánh, các đóng góp gradient được **cộng**, giống cộng số đường đi từ các nhánh tới một đỉnh.

PyTorch dựng DAG ngay trong lúc forward chạy và dựng lại ở mỗi lần forward (đồ thị động). Vì vậy `if` hay vòng lặp Python trong forward không cản autograd: backward đi theo đúng nhánh đã thực sự chạy.

Mỗi phép lưu đúng thứ nó cần cho backward. Sigmoid lưu chính đầu ra y vì đạo hàm là $y(1 - y)$, nên sửa y tại chỗ như `y.add_(1)` sau forward sẽ làm backward báo lỗi.

Thử trên một phép tính nhỏ. Đặt $w = 2$, $u = 3w$, $J = (u - 1)^2$. Forward cho $u = 6$, $J = 25$. Nếu tăng w một lượng rất nhỏ, u tăng ba lần lượng đó. Còn J nhảy với u một lượng $2(u - 1) = 10$. Vì vậy

$$\frac{dJ}{dw} = \frac{dJ}{du} \frac{du}{dw} = 2(u - 1) \cdot 3 = 30.$$



Forward $2 \rightarrow 6 \rightarrow 25$; backward nhân đạo hàm cục bộ: $10 \times 3 = 30$.

Gradient dương nói rằng tăng w một chút sẽ làm loss tăng tại điểm đang xét. Muốn đi xuống, gradient descent dùng $w \leftarrow w - \eta 30$. Với $\eta = 0.01$, một bước cho $w = 1.7$; đó là phép cập nhật riêng, không thuộc `backward()`.

Với nhiều biến, gradient gom các đạo hàm riêng: $f = x^2y + 3y$ tại $(1, 2)$ có gradient $(2xy, x^2 + 3) = (4, 4)$.

Python

```
w = torch.tensor(2., requires_grad=True)
u = 3 * w
J = (u - 1) ** 2
J.backward()
print(round(J.item(), 2), round(w.grad.item(), 2))
with torch.no_grad():
    w -= 0.01 * w.grad
print(round(w.item(), 2))
```

Hai số ở dòng đầu khớp 25 và 30 đã tính tay; dòng sau kiểm phép cập nhật 1.7. Vài điều đọc từ đoạn code:

- `requires_grad=True` yêu cầu PyTorch ghi lại các phép tính dùng tới `w`.
- `w` là tensor lá do ta tạo trực tiếp; gradient được cộng vào `w.grad`. Tensor trung gian như `u` không giữ `.grad`, nhưng đạo hàm vẫn đi qua nó bình thường.
- `J` là một số (scalar) nên `backward()` biết bắt đầu từ độ nhạy 1. Với đầu ra là vector, phải gộp thành một số trước, ví dụ lấy tổng hoặc trung bình.

HỎI NHANH

Vấn $J = (3w - 1)^2$, nhưng tại $w = 0$. Gradient bằng bao nhiêu, và một bước gradient descent đẩy w theo hướng nào?

- A 6, giảm w , tức đi ngược dấu gradient
- B 6, tăng w , tức đi theo dấu gradient
- C -6, giảm w , tức đi theo dấu gradient
- D -6, tăng w , tức đi ngược dấu gradient

HỎI NHANH

Cho `x = torch.tensor(-3., requires_grad=True)`, rồi `y = 2 * x if x > 0 else x ** 2` và `y.backward()`. `x.grad` bằng bao nhiêu?

- A -6.0, đạo hàm của nhánh `x ** 2`
- B 2.0, đạo hàm của nhánh `2 * x`
- C -2.0, trung bình đạo hàm của hai nhánh
- D Báo lỗi, vì graph phải cố định trước khi forward

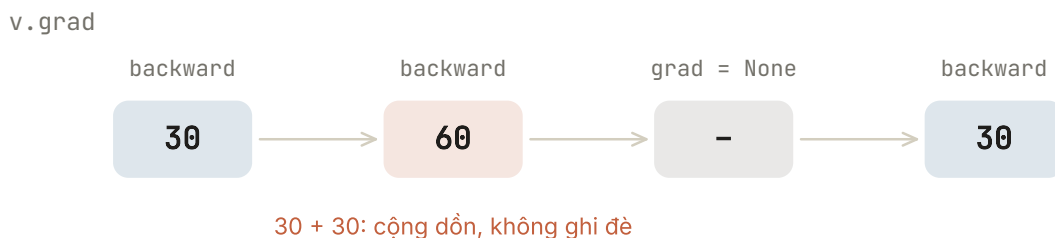
Gradient được cộng dồn

PyTorch cộng gradient mới vào `.grad`. Cách này cho phép tích lũy nhiều microbatch, nhưng trong training loop bình thường ta phải xóa trước mỗi bước. Sau `backward()`, PyTorch giải phóng các giá trị trung gian của graph, nên mỗi lượt dưới đây có forward riêng:

Python

```
v = torch.tensor(2., requires_grad=True)
((3 * v - 1) ** 2).backward()
print(v.grad.item())
((3 * v - 1) ** 2).backward()
print(v.grad.item())
v.grad = None
((3 * v - 1) ** 2).backward()
print(v.grad.item())
```

Dãy 30, 60, 30 cho thấy gradient cộng dồn và phép xóa đã hoạt động. Trong training loop, việc xóa do `optimizer.zero_grad(set_to_none=True)` đảm nhận. Gọi `backward()` hai lần trên cùng một graph sẽ báo lỗi; đừng sửa bằng `retain_graph=True`: giữ graph tốn bộ nhớ, không xóa gradient cũ, và graph cũ vẫn mô tả trọng số cũ. Mỗi bước huấn luyện cần forward mới rồi backward mới.



Mỗi lần backward cộng thêm: 30 rồi 60; xóa grad thì lại 30.

HỎI NHANH

Một bạn tính loss một lần trước vòng lặp, rồi mỗi epoch gọi `loss.backward()` và `optimizer.step()`. Sang epoch thứ hai thì báo `RuntimeError: Trying to backward through the graph a second time. Sửa thế nào?`

- A Thêm `retain_graph=True` để graph không bị giải phóng
- B Gọi `optimizer.zero_grad()` trước lần `backward()` thứ hai
- C Đưa forward và phép tính loss vào trong vòng lặp
- D Đổi thành `loss.detach().backward()` để tách khỏi graph cũ

Detach, no_grad và bản sao

Khi lưu dự đoán để tính metric, ta không muốn giữ cả đường tính toán. Có ba công cụ:

- `detach()` tạo tensor không nối gradient về nguồn. Nó vẫn dùng chung vùng nhớ với tensor gốc.
- `detach().clone()` tạo bản sao độc lập, dùng khi muốn lưu lại giá trị lúc đó.
- `with torch.no_grad():` tạm ngừng ghi graph cho các phép trong khối, dùng khi suy luận hoặc cập nhật trọng số bằng tay.

`no_grad()` không tự chuyển model sang chế độ đánh giá `eval()` (bài 3.4 và 3.6); ngược lại, `eval()` cũng không tắt gradient. Lúc đánh giá nên dùng cả hai.

Ta kiểm ý nghĩa “cắt đường gradient” bằng biểu thức v^3 . Nếu giữ cả hai nhánh, đạo hàm tại $v = 2$ là $3v^2 = 12$. Nếu tính v^2 rồi detach, hệ số 4 ấy được coi như hằng số:

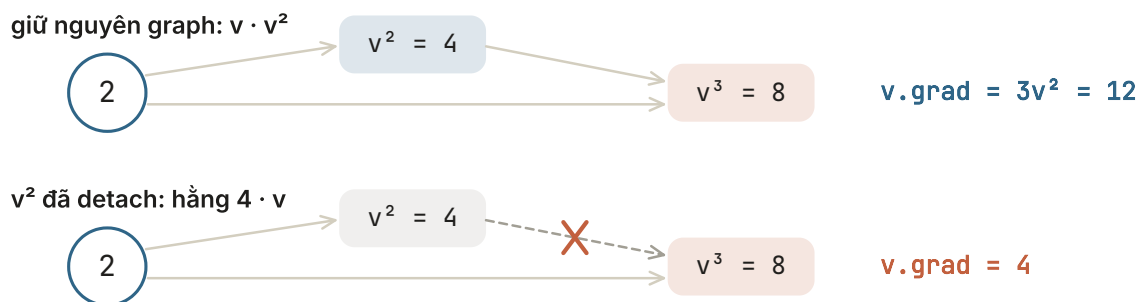
Python

```
v.grad = None
square = v ** 2
frozen_square = square.detach()
(frozen_square * v).backward()
```

```
print(v.grad.item(), frozen_square.requires_grad)
snapshot = square.detach().clone()
print(snapshot.item())
```

Gradient 4 là đúng với **graph đã bị cắt**, dù giá trị forward vẫn bằng 8. Vì vậy detach không phải cách sửa lỗi backward: nó đổi bài toán, và tham số nằm trước điểm cắt sẽ không học được. Chuyển tensor cần gradient sang NumPy cũng nên viết `tensor.detach().cpu().numpy()` khi chỉ phục vụ phân tích. Với một số đơn lẻ như loss, `loss.item()` trả về số Python; ghi log bằng số này không giữ graph.

Cập nhật tham số bằng tay cũng đặt trong `no_grad()`: `with torch.no_grad(): w -= lr * w.grad`; viết thẳng thì PyTorch báo lỗi sửa tại chỗ một tensor là cần gradient. Ngược lại, chạy forward trong `no_grad()` rồi gọi `loss.backward()` sẽ báo loss không có `grad_fn`. Detach còn được dùng có chủ đích: mang trạng thái ẩn của RNN sang đoạn văn bản sau bằng `hidden.detach()` (bài 3.8).



Giữ graph: gradient 12. Detach v^2 : chỉ còn 4.

HỎI NHANH

w bằng 2 và đã có w.grad bằng 30. Chạy `snap = w.detach()`, rồi `with torch.no_grad(): w -= 0.1 * w.grad.snap.item()` in ra gì?

- A 2.0, vì snap đã tách khỏi graph
- B 2.0, vì no_grad không làm đổi giá trị của w
- C -1.0, vì snap dùng chung vùng nhớ với w
- D Báo lỗi, vì tensor đã detach thì không đọc được

Thiết bị và nơi lưu dữ liệu

GPU thực hiện nhiều phép số song song, nhưng mọi tensor trong một phép tính phải cùng device:

- Với tensor, `.to(device)` trả về **tensor mới** trên device đã chọn, nên phải gán lại: `X = X.to(device)`.
- Với model, `model.to(device)` chuyển tham số của model tại chỗ.
- Tensor tạo mới trong forward, như `torch.ones(...)`, mặc định nằm trên CPU; viết `device=x.device` hoặc `torch.ones_like` để nó theo dữ liệu, đừng ghi cứng 'cuda'.

Đoạn sau chọn CUDA khi có, nếu không thì dùng CPU; các dòng còn lại không phụ thuộc máy:

Python

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
X_device = X.to(device)
linear = torch.nn.Linear(3, 2).to(device)
with torch.no_grad():
    result = linear(X_device)
print(str(device), tuple(result.shape))
print(result.detach().cpu().numpy().shape)
```

Máy chạy bài không có CUDA nên dòng đầu in cpu; trên máy có CUDA, cùng đoạn code chọn cuda mà không phải sửa dòng nào. Dòng hai là đường thường dùng khi lấy kết quả ra phân tích: ngắt graph, chép về CPU, rồi mới sang NumPy.

GPU cũng không tự nhanh hơn: với dữ liệu rất nhỏ, thời gian chép dữ liệu và khởi động có thể lớn hơn phần tính. Bài 3.7 có cách đo thời gian mỗi bước trên máy CUDA.

HỎI NHANH

Model đã gọi `model.to('cuda')`. Trong vòng lặp có dòng `X.to('cuda')`, không gán lại, rồi `model(X)`, và PyTorch báo các tensor nằm trên hai device khác nhau. Vì sao?

- A `X.to` trả về một bản mới, còn `X` cũ vẫn ở CPU
- B `model.to` cũng phải gán lại thì model mới chuyển
- C Phải dùng `X.cuda()`, vì `.to` chỉ đổi dtype
- D Batch quá nhỏ nên PyTorch giữ `X` ở CPU

Khi đi thi

- In shape ở ranh giới dataset, model và loss; đặt `assert pred.shape == target.shape`.
- Nhãn lớp cho cross-entropy là `long`; nhãn hồi quy và nhãn BCE là `float`.
- Đặt device ở một chỗ và chuyển bằng `.to(device)`, đừng gõ cứng `.cuda()`.

Tóm tắt

- Tensor là mảng kèm shape, dtype và device. `reshape` chia lại dãy phần tử, `transpose` đổi trục; broadcasting có thể lạng lã tạo shape sai.
- Autograd ghi DAG của forward, rồi cộng gradient theo các đường ở backward.

- `backward()` chưa cập nhật trọng số; `.grad` cộng dồn nên phải xóa trước mỗi bước.
- `detach()` cắt graph, `clone()` tạo bản sao, `.item()` lấy ra số Python, `no_grad()` ngừng ghi graph cho một khối.

Kiểm tra cuối bài

CÂU 1

Batch embedding `E` có shape `(4, 6, 8)`: 4 câu, mỗi câu 6 token, mỗi token 8 đặc trưng. Muốn mỗi câu có một vector bằng trung bình các token, ta gọi `E.mean(dim=?)`, và nhận được shape gì?

A `dim=0`, shape `(6, 8)`

B `dim=1`, shape `(4, 8)`

C `dim=1`, shape `(4, 1, 8)`

D `dim=2`, shape `(4, 6)`

CÂU 2

`a = torch.tensor([1, 2, 3])` và `b = torch.Tensor([1, 2, 3])`. Hai tensor này có dtype gì?

A a là float32, b là float32

B a là int64, b là int64

C a là int64, b là float32

D a là float32, b là int64

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 129

CÂU 3

`x = torch.tensor([1., -2., 3.], requires_grad=True)`, `loss = (x ** 2).mean()`, rồi `loss.backward()`. `x.grad` bằng bao nhiêu, làm tròn hai chữ số?

A [0.67, -1.33, 2.00]

B [2.00, -4.00, 6.00]

C [0.33, 1.33, 3.00]

D 4.67

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 53

CÂU 4

Một model khai báo ba lớp `fc1`, `fc2`, `fc3` trong `__init__`, nhưng `forward` chỉ gọi `fc1` rồi `fc2`. Sau `loss.backward()`, `fc3.weight.grad` là gì?

- A Tensor toàn 0, vì `fc3` không ảnh hưởng tới `loss`
- B None, vì `fc3` không có trong đồ thị của `forward`
- C Gradient bình thường, vì `fc3` có trong `model.parameters()`
- D Báo lỗi lúc `backward`, vì có tham số không được dùng

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 54

CÂU 5

Huấn luyện full-batch trên dữ liệu cố định. Mỗi epoch gọi `optimizer.zero_grad()`, tính `loss`, gọi `loss.backward()`, nhưng quên `optimizer.step()`. `Loss` in ra qua các epoch sẽ thế nào?

- A Giảm chậm, vì `backward` tự cập nhật với learning rate mặc định
- B Tăng dần, vì gradient cộng dồn qua các epoch
- C Không đổi, vì `backward` chỉ tính gradient
- D Thành NaN, vì graph cũ không được giải phóng

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 120

CÂU 6

Một bạn khởi tạo `w = torch.randn(3, requires_grad=True) * 0.01`, tính loss rồi gọi `backward()`. `w.grad` là `None`, và PyTorch đưa ra cảnh báo. Vì sao?

- A `w` được sinh ra từ một phép nhân, không phải tensor lá
- B Nhân với 0.01 làm mất `requires_grad` của `w`
- C `torch.randn` tạo tensor không hỗ trợ đạo hàm
- D Gradient quá nhỏ nên bị làm tròn thành `None`

CÂU 7

loss là tensor 0 chiều, logits là tensor (32, 10), cả hai còn nối với graph. Ta muốn (a) ghi loss vào log dưới dạng số Python, và (b) giữ logits ở dạng tensor nhưng không giữ graph. Gọi gì?

- A (a) `loss.detach()`, (b) `logits.detach()`
- B (a) `loss.detach()`, (b) `logits.item()`
- C (a) `loss.item()`, (b) `logits.item()`
- D (a) `loss.item()`, (b) `logits.detach()`

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 145

CÂU 8

Hàm dự đoán tập test chạy `preds.append(model(xb))` cho 2000 batch trên GPU. Bộ nhớ tăng dần rồi báo out of memory, dù batch nào cũng rất nhỏ. Sửa thế nào?

- A Gọi `optimizer.zero_grad()` sau mỗi batch dự đoán
- B Giảm batch size xuống một nửa rồi chạy lại
- C Gọi `torch.cuda.empty_cache()` sau mỗi batch
- D Chạy trong `torch.no_grad()`, và lưu kết quả bằng `.cpu()`

CÂU 9

X có shape (64, 10): 64 mẫu, mỗi mẫu 10 đặc trưng. Để trừ mỗi hàng cho trung bình của chính hàng đó, một bạn viết `m = X.mean(dim=1)` rồi `X - m`. Kết quả thế nào?

- A Báo lỗi, vì (64, 10) và (64,) không broadcast được
- B Chạy đúng, mỗi hàng trừ trung bình của chính nó
- C Chạy được, nhưng ra (64, 64), so mọi cặp mẫu
- D Chạy được, nhưng trừ trung bình theo từng cột

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 104

CÂU 10

Model có tham số trên CUDA, x đang ở CPU, và y là nhãn lớp kiểu long dùng cho cross-entropy. Muốn train một batch trên CUDA, ta làm gì?

- A Chỉ chuyển x , vì hàm loss tự chuyển y theo input
- B Chuyển cả x lẫn y tới device của model
- C Gọi `model.eval()` để tự đồng bộ device của input
- D Đổi y sang float để loss nhận nhãn ở CPU

Đọc thêm

- [PyTorch — Automatic Differentiation](#): đọc về graph động, gradient cộng dồn và cách tắt theo dõi.
- [D2L — Forward/Backward Propagation](#): xem đồ thị tính toán như cầu nối từ đạo hàm một biến sang mạng.
- [PyTorch 2.9 — Tensor](#): tra shape, dtype, device và các thao tác tensor trước khi ghép model lớn.