

Fine-tuning

☑ Trắc nghiệm 19 📧 Thực hành 2 🔗 Chạy trên Kaggle ⬇ PDF bài

Khi dữ liệu mới khác nhiều so với dữ liệu model từng học, vector lấy từ feature extraction có thể chưa đủ tốt. Fine-tuning là huấn luyện tiếp chính model pretrained trên dữ liệu mới. Bài này làm theo hai bước: đầu tiên giữ nguyên thân model, chỉ huấn luyện lớp cuối mới thêm vào; sau đó mở thêm các lớp của thân với learning rate nhỏ. Mỗi bước được chọn bằng điểm validation, và cuối bài kiểm xem tham số nào thực sự đã thay đổi.

MỤC TIÊU

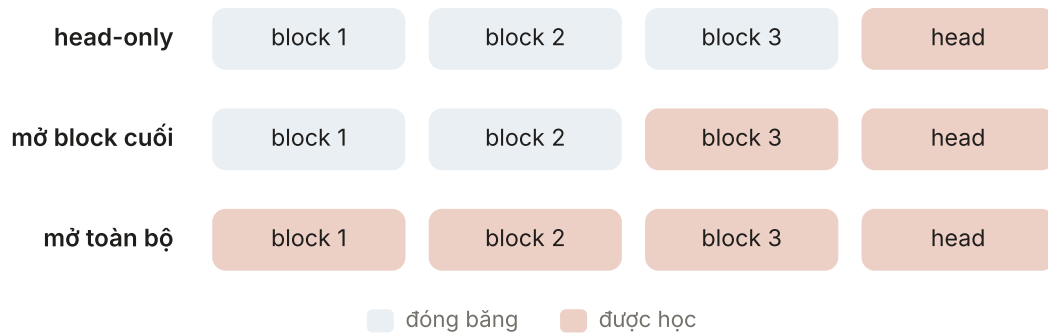
- Thay lớp cuối và chọn phần nào của model được huấn luyện.
- Phân biệt `requires_grad`, `train()`/`eval()` và `no_grad()`.
- Đặt learning rate riêng cho thân model và lớp cuối; chọn checkpoint bằng validation.
- Biết BatchNorm vẫn thay đổi dù trọng số đã bị khóa.

Chuyển kiến thức nào sang nhiệm vụ mới?

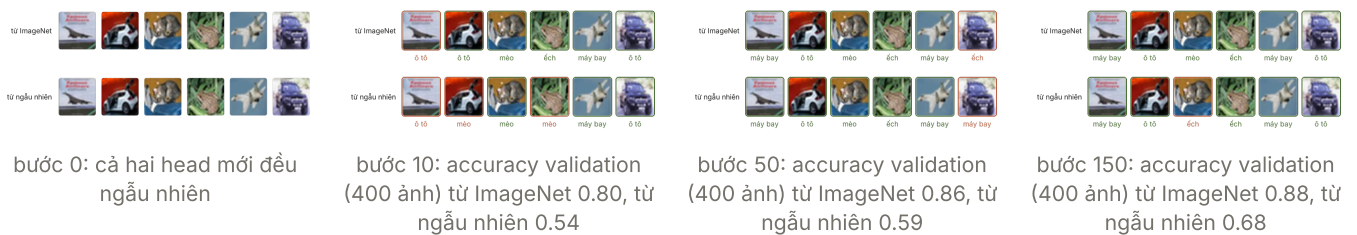
Model pretrained gồm encoder và thường có một head phục vụ nhiệm vụ cũ. Encoder ảnh có thể học nét, cấu trúc và những tổ hợp hình dạng có ích cho nhiều nhãn khác. Nhưng head phân loại nhãn cũ không có lý do biết các nhãn mới của câu lạc bộ. Ta giữ encoder và thay head có đúng số lớp K của bài mới. Cách làm này gọi là học chuyển giao (transfer learning).

Nó có ích nhất khi dữ liệu có nhãn ít mà nhiệm vụ nguồn đủ gần; ngay cả với 500 ảnh X-quang, bắt đầu từ trọng số pretrained rồi fine-tune vẫn thường tốt hơn học từ đầu.

Viết $\mathbf{z} = f_{\theta}(\mathbf{x})$ và logits $\ell = g_{\phi}(\mathbf{z})$. θ là tham số encoder, ϕ là tham số head. Head-only training chỉ cập nhật ϕ . Fine-tuning mở một phần hoặc toàn bộ θ cùng với ϕ . Phần mở nhiều hơn có thể thích nghi tốt hơn nhưng cũng có thể học thuộc dữ liệu nhỏ hoặc làm mất biểu diễn có ích.



Ba mức mở tham số: chỉ head, thêm block cuối, rồi toàn bộ encoder.



Cùng số bước, bắt đầu từ trọng số ImageNet đoán đúng sớm hơn hẳn từ ngẫu nhiên.

HỎI NHANH

Nhiệm vụ cũ có 1000 lớp, bài mới có 3 lớp. Nạp checkpoint xong, phần nào chắc chắn phải thay hoặc kiểm trước khi train?

- A Batch size, vì đầu ra 3 lớp cần batch chia hết cho 3
- B Head cuối, và bảng đổi ba nhãn mới sang chỉ số 0, 1, 2
- C Seed, vì head cũ được khởi tạo bằng một seed khác
- D Learning rate của encoder, vì 1000 lớp cần bước nhỏ hơn

Trong mạng lớn, các block gần input thường bắt đầu hiệu chung như nét, cạnh, mảng màu; các block sau mang đặc trưng sát nhiệm vụ cũ hơn. Vì vậy thứ tự thử thực dụng là head trước, rồi mở thêm block cuối, rồi mới đến toàn bộ, đúng ba hàng của hình trên. Khi miền đích khác hẳn nguồn, như ảnh X-quang so với ảnh đời thường, đặc trưng ở các

block đầu cũng có thể cần đổi; mỗi mức mở khi đó là một thí nghiệm, và baseline head-only là mốc để biết thay đổi có đáng giá.

HỎI NHANH

Dữ liệu đích ít và khá giống miền nguồn, head-only đã có điểm validation. Ngân sách chỉ đủ thử thêm một cấu hình. Nên thử gì?

- A Mở toàn bộ mạng, với cùng learning rate của head
- B Khởi tạo lại encoder rồi học từ đầu trên dữ liệu đích
- C Mở block đầu gần input, vì đặc trưng nền quan trọng nhất
- D Mở thêm block cuối, giữ các block đầu đóng băng

Một mô hình nguồn tự pretrain

Để thấy mọi bước mà không cần tải trọng số, ta tự tạo một mô hình nguồn ngay trong notebook từ bộ chữ số 8×8 của scikit-learn. Nhiệm vụ nguồn phân loại các chữ số 0–7; nhiệm vụ đích phân biệt 8 và 9. Mọi ảnh 8 hoặc 9 đều bị loại khỏi việc học nguồn, nên encoder nguồn đóng đúng vai một checkpoint pretrained: đã học đặc trưng hữu ích trên dữ liệu khác và chưa từng thấy nhãn đích.

Với ResNet hay DistilBERT, các bước phía sau giống hệt; chỉ khác ở chỗ trọng số được nạp từ tệp checkpoint thay vì vừa học xong.

Một encoder $64 \rightarrow 32 \rightarrow \text{ReLU}$ đủ để thực nghiệm chạy trong vài giây. Pixel của bộ dữ liệu này nằm từ 0 đến 16, nên input được chia 16; mọi bước sau, kể cả nhiệm vụ đích, phải dùng đúng phép chia ấy, giống như fine-tune một checkpoint thật phải dùng đúng processor của nó (bài 4.1). Một phần ảnh nguồn được giữ làm validation để kiểm encoder thực sự đã học, thay vì gọi trọng số ngẫu nhiên là “đã học”.

Python

```

import copy
import numpy as np
import torch
from torch import nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

torch.set_num_threads(1)
torch.manual_seed(3)
digits = load_digits()
source_ids = np.flatnonzero(digits.target < 8)
target_ids = np.flatnonzero(digits.target >= 8)
source_train, source_valid = train_test_split(
    source_ids, test_size=0.25, stratify=digits.target[source_ids], random_state=3
)

def inputs(ids):
    return torch.tensor(digits.data[ids] / 16.0, dtype=torch.float32)

source_encoder = nn.Sequential(nn.Linear(64, 32), nn.ReLU())
source_head = nn.Linear(32, 8)
source_model = nn.Sequential(source_encoder, source_head)
source_optimizer = torch.optim.Adam(source_model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()
source_x = inputs(source_train)
source_y = torch.tensor(digits.target[source_train], dtype=torch.long)
for _ in range(80):
    source_model.train()
    source_optimizer.zero_grad(set_to_none=True)
    loss = criterion(source_model(source_x), source_y)
    loss.backward()
    source_optimizer.step()

source_model.eval()
with torch.no_grad():
    source_prediction = source_model(inputs(source_valid)).argmax(dim=1).numpy()
print("source train/valid:", len(source_train), len(source_valid))
print("source valid accuracy:", round(float((source_prediction ==
digits.target[source_valid]).mean()), 4))
print("source gap target:", len(set(source_ids) & set(target_ids)) == 0)

```

Accuracy 0.9806 trên 361 ảnh nguồn giữ lại cho thấy encoder đã học được đặc trưng chữ số có ích. Dòng thứ ba xác nhận không ảnh 8 hay 9 nào lọt vào việc học nguồn; nếu có,

điểm của nhiệm vụ đích sau này sẽ đo cả phần encoder đã thuộc lòng, không chỉ phần kiến thức được chuyển.

Split ở đây chia theo từng ảnh; với dữ liệu có nhiều ảnh của cùng một người hay cùng một buổi, split phải chia theo nhóm ấy (bài 1.3).

HỎI NHANH

Một đội pretrain encoder trên toàn bộ 1797 ảnh chữ số từ 0 đến 9 cùng nhãn của chúng, rồi báo accuracy của nhiệm vụ phân biệt 8 với 9 trên validation lấy từ chính bộ ảnh ấy. Con số này đo gì?

- A Đúng khả năng chuyển học, vì head đích vẫn là head mới
- B Đúng khả năng chuyển học, vì phần nguồn không dùng head đích
- C Một con số quá lạc quan, vì encoder đã học cả ảnh validation
- D Một con số quá bi quan, vì nhãn 0 đến 7 làm nhiễu encoder

Bắt đầu bằng một head mới

Ta dành 48 ảnh đích làm train có nhãn; các ảnh đích còn lại làm validation. Nhãn 8 đổi thành 0, nhãn 9 đổi thành 1 để khớp head hai lớp. Việc đổi nhãn phải lưu lại để trả output đúng tên lớp sau khi dự đoán.

Python

```
target_train, target_valid = train_test_split(
    target_ids, train_size=48, random_state=5, stratify=digits.target[target_ids]
)
target_x = inputs(target_train)
target_y = torch.tensor(digits.target[target_train] - 8, dtype=torch.long)
valid_x = inputs(target_valid)
valid_y = torch.tensor(digits.target[target_valid] - 8, dtype=torch.long)

torch.manual_seed(10)
```

```

initial_head = nn.Linear(32, 2)
head_only = nn.Sequential(copy.deepcopy(source_encoder), copy.deepcopy(initial_head))
for parameter in head_only[0].parameters():
    parameter.requires_grad_(False)
optimizer = torch.optim.SGD(head_only[1].parameters(), lr=0.05)
before = {name: value.detach().clone() for name, value in head_only[0].named_parameters()}

head_only.train()
head_only[0].eval()
optimizer.zero_grad(set_to_none=True)
loss = criterion(head_only(target_x), target_y)
loss.backward()
print("encoder grad None:", all(p.grad is None for p in head_only[0].parameters()))
print("head có gradient:", head_only[1].weight.grad.norm().item() > 0)
optimizer.step()
print("encoder không đổi:", all(torch.equal(before[n], p) for n, p in
head_only[0].named_parameters()))
print("target train/valid:", len(target_train), len(target_valid))

```

Ở mạng này, encoder chỉ có Linear và ReLU; eval() chưa đổi phép tính của nó. Ta vẫn gọi để tạo thói quen đúng khi sau này dùng encoder có Dropout hoặc BatchNorm.

Dòng encoder grad None kiểm việc không tạo gradient cho tham số đã đóng băng. Dòng thứ ba so từng giá trị tensor trước và sau một bước, kiểm việc optimizer không sửa encoder. Nếu chỉ nhìn loss giảm, ta không biết phần nào đã được cập nhật.

Đóng băng nghĩa là đặt requires_grad_(False) cho tham số encoder: chúng không nhận gradient và optimizer không sửa. Số tham số được học khi đó chỉ là của head, đếm bằng `sum(p.numel() for p in model.parameters() if p.requires_grad)`: ResNet-18 đóng băng hết rồi thay fc bằng `Linear(512, 10)` còn $512 \cdot 10 + 10 = 5130$; ResNet-50 với head 2048 $\rightarrow K$ là $2048K + K$.

HỎI NHANH

Một tham số có `requires_grad=False` nhưng vẫn nằm trong forward. Khi dự đoán, nó đóng vai trò gì?

- A Vẫn tính forward như cũ, chỉ không được học
- B Bị bỏ qua, nên output của tầng ấy bằng 0
- C Bị thay bằng giá trị trung bình của nó
- D Bị tách khỏi mạng cho tới khi gọi lại `train()`

HỎI NHANH

Head hai lớp được học với nhãn `digit - 8`. Validation accuracy đạt 0.94, nhưng file nộp ghi thẳng `argmax` của logits và được chấm 0 điểm. Lỗi khả dĩ nhất là gì?

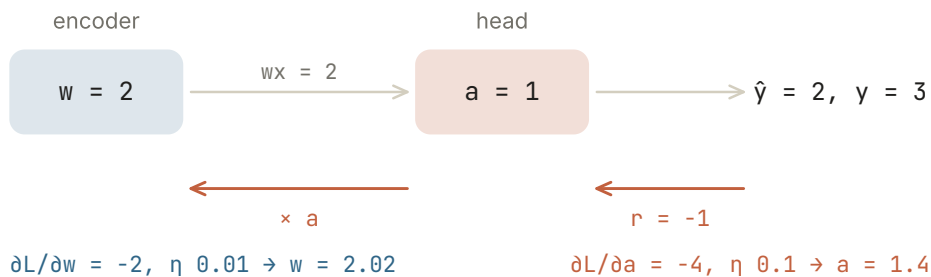
- A Head hai lớp không đủ sức cho dữ liệu test
- B Validation bị rò, nên 0.94 là con số ảo
- C File nộp ghi chỉ số 0, 1 thay vì 8, 9
- D Encoder ở chế độ `eval()` khi dự đoán test

Vì sao encoder dùng bước nhỏ hơn

Xét mô hình một chiều $\hat{y} = a(wx)$, encoder có tham số w , head có tham số a , loss $L = (\hat{y} - y)^2$. Với $x = 1$, $w = 2$, $a = 1$, $y = 3$, ta có $\hat{y} = 2$, phần dư -1 , nên

$$\frac{\partial L}{\partial a} = 2(-1) \cdot 2 = -4, \quad \frac{\partial L}{\partial w} = 2(-1) \cdot 1 = -2.$$

Nếu learning rate head là 0.1, encoder là 0.01, cập nhật cho $a = 1.4$ và $w = 2.02$. Nếu encoder đóng băng thì w vẫn bằng 2.



Learning rate 0.1 cho head, 0.01 cho encoder: a lên 1.4, w chỉ nhích tới 2.02.

Python

```
x, w, a, y = 1.0, 2.0, 1.0, 3.0
residual = a * w * x - y
grad_a = 2 * residual * w * x
grad_w = 2 * residual * a * x
print("gradient:", grad_a, grad_w)
print("sau buoc:", a - 0.1 * grad_a, w - 0.01 * grad_w)
```

Để ý $\partial L / \partial w$ chứa a : gradient về encoder phải đi qua head. Head mới khởi tạo ngẫu nhiên, nên trong những bước đầu, hướng gradient mà encoder nhận được cũng mang theo sự ngẫu nhiên đó. Nếu encoder đi bước lớn như head, nó có thể bị kéo xa chỉ để bù cho một head còn sai, làm hỏng những đặc trưng tốt đã học. Việc mô hình mất dần kiến thức nguồn khi được huấn luyện tiếp cho nhiệm vụ mới gọi là quên thảm khốc (catastrophic forgetting); bước lớn ở phần pretrained làm hiện tượng này đến nhanh hơn.

Hai cách hạn chế: learning rate nhỏ cho encoder, hoặc học head-only vài epoch rồi mới mở encoder. Tỷ lệ 10 lần ở ví dụ là một lựa chọn, không phải hằng số. Với BERT, bài báo gốc fine-tune toàn bộ mô hình với learning rate từ $2 \cdot 10^{-5}$ đến $5 \cdot 10^{-5}$ trong 2-4 epoch, nhỏ hơn mức 10^{-4} dùng khi pretrain chính BERT.

HỎI NHANH

Vẫn ví dụ ấy, với $x = 1$, $w = 2$, $y = 3$, nhưng head mới được khởi tạo $a = -1$. Gradient theo w bằng bao nhiêu, và gradient descent đẩy w theo hướng nào?

- A -2 , nên w tăng, như ví dụ có $a = 1$
- B -10 , nên w tăng để $\hat{y}$ lớn lên
- C 0 , nên w đứng yên chờ head học xong
- D 10 , nên w giảm

So sánh head-only và mở encoder

Hai thực nghiệm bắt đầu từ cùng encoder nguồn và cùng head mới, nên khác biệt chỉ nằm ở việc mở encoder. Ta chạy 60 epoch full-batch, chấm validation sau mỗi epoch và giữ checkpoint có validation loss thấp nhất. Head dùng learning rate 0.05; khi mở, encoder có nhóm riêng với 0.003, nhỏ hơn khoảng 17 lần. Vì validation vừa dùng để chọn epoch vừa để so hai mode, con số báo cáo cuối cùng nên lấy từ một test giữ kín.

Python

```
def run_transfer(open_encoder):
    transferred = nn.Sequential(copy.deepcopy(source_encoder), copy.deepcopy(initial_head))
    for parameter in transferred[0].parameters():
        parameter.requires_grad_(open_encoder)
    groups = [{"params": transferred[1].parameters(), "lr": 0.05}]
    if open_encoder:
        groups.append({"params": transferred[0].parameters(), "lr": 0.003})
    optimizer = torch.optim.Adam(groups)
    best_loss = float("inf")
    best_state, best_epoch = None, None
    history = []
    for epoch in range(1, 61):
        transferred.train()
        if not open_encoder:
            transferred[0].eval()
```

```

optimizer.zero_grad(set_to_none=True)
train_loss = criterion(transferred(target_x), target_y)
train_loss.backward()
optimizer.step()
transferred.eval()
with torch.no_grad():
    val_loss = criterion(transferred(valid_x), valid_y).item()
history.append((train_loss.item(), val_loss))
if val_loss < best_loss:
    best_loss = val_loss
    best_epoch = epoch
    best_state = copy.deepcopy(transferred.state_dict())
transferred.load_state_dict(best_state)
with torch.no_grad():
    accuracy = (transferred(valid_x).argmax(dim=1) == valid_y).float().mean().item()
trainable = sum(p.numel() for p in transferred.parameters() if p.requires_grad)
return transferred, history, best_epoch, best_loss, accuracy, trainable

for opened in [False, True]:
    net, history, epoch, val_loss, accuracy, count = run_transfer(opened)
    print("open encoder =", opened, "| trainable =", count, "| best epoch =", epoch)
    print("best val loss =", round(val_loss, 4),
          "| epoch 60 val loss =", round(history[-1][1], 4),
          "| val accuracy =", round(accuracy, 4))

```

Đọc loss và accuracy ở checkpoint được chọn, không chỉ ở epoch cuối:

- **Head-only** học 66 tham số ($32 \cdot 2 + 2$); validation loss thấp nhất 0.2247 ở epoch 36, đến epoch 60 là 0.2366 trong khi train loss vẫn giảm. Giữ checkpoint epoch 36 nghĩa là không phải trả giá cho 24 epoch học thêm vô ích.
- **Mở encoder** học 2146 tham số (thêm $64 \cdot 32 + 32 = 2080$ của encoder), loss 0.1570 và accuracy 0.9379, so với 0.9020 của head-only: đúng thêm 11 trong 306 ảnh validation. Checkpoint tốt nhất của nó rơi đúng epoch cuối, tức validation còn đang giảm; ngân sách 60 epoch có thể chưa đủ, và đó là tín hiệu để chạy dài hơn, không phải bằng chứng đã hội tụ.

Trên một split nhỏ, chênh lệch 11 ảnh là bằng chứng có ích, chưa phải định luật “mở encoder luôn thắng”. Loss và accuracy cũng có thể chọn hai epoch khác nhau: cross-entropy còn phạt sự tự tin sai, còn accuracy chỉ nhìn argmax.

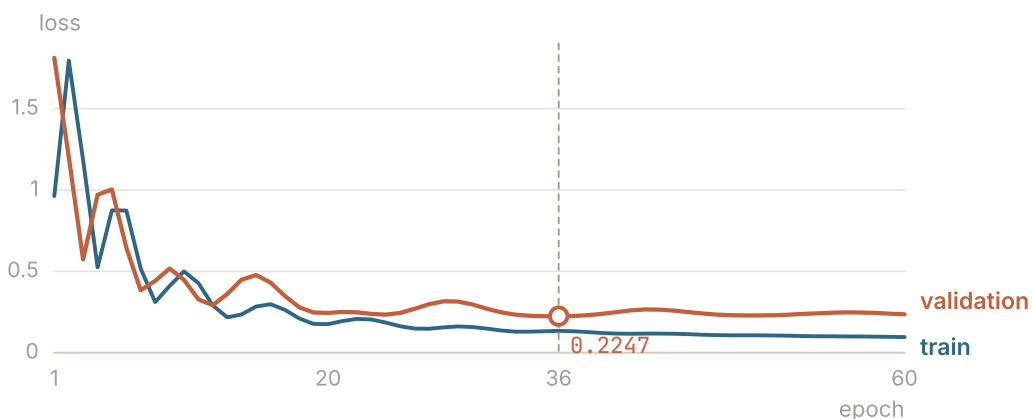
Ở đây encoder chỉ có một tầng Linear nên mở nó là mở toàn bộ; với ResNet, “mở block cuối” là một tập tham số riêng, có số lượng và bộ nhớ riêng phải đo lại. Mỗi tham số được mở kéo theo một gradient và, với Adam, hai moment m , v : thêm ba số float32 cho mỗi tham số, chưa kể activation phải giữ để backward qua phần mở (bài 3.7). Với 66 tham số, phần này không đáng kể; với hàng chục triệu tham số, nó lên tới hàng trăm MB.

head-only 66 tham số · val acc 0.902

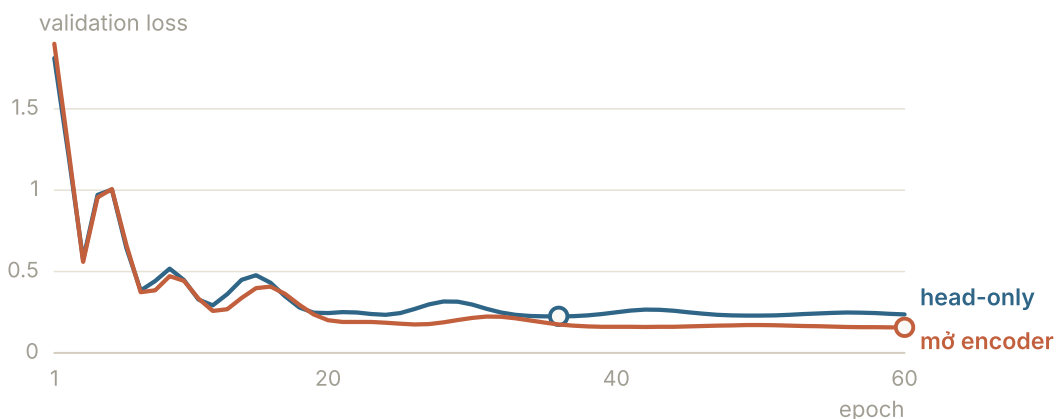
mở encoder 2146 tham số · val acc 0.938

đúng thêm 11 trên 306 ảnh validation; mỗi tham số mở kèm gradient và hai moment Adam

Mở encoder: tham số học từ 66 lên 2146, accuracy từ 0.902 lên 0.938.



Chỉ học head: validation loss thấp nhất ở epoch 36, nên lưu checkpoint ở đó.



Mở encoder với learning rate nhỏ cho validation loss thấp hơn chỉ học head.

HỎI NHANH

Một lần fine-tune 20 epoch có checkpoint tốt nhất ở epoch 20, và validation loss vẫn đang giảm. Hiểu kết quả này thế nào?

- A Có thể chưa đủ epoch, nên chạy thêm rồi xem tiếp
- B Đã hội tụ đúng ở epoch 20, nên dừng tại đó
- C Đã overfit, vì điểm tốt nhất rơi vào epoch cuối
- D Chọn lại theo train loss, vì validation chưa ổn

Script của chương đưa đúng thí nghiệm này lên DistilBERT với model và dữ liệu lưu sẵn trên máy. Mode head đóng bằng backbone và giữ nó ở eval; mode `full` học backbone với learning rate 10^{-5} và head với 10^{-3} . Cả hai chọn checkpoint theo macro-F1 validation và ghi số tham số học, lịch sử validation, thời gian vào `RESULT.json`; so hai tệp ấy giống như so hai cặp dòng output trên.

Freeze, eval và BatchNorm là ba chuyện

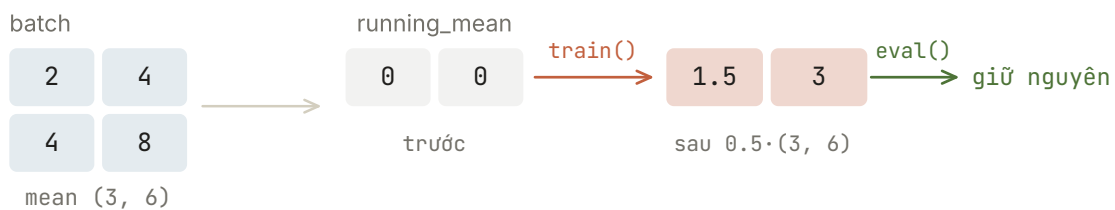
BatchNorm (bài 3.6) có tham số affine như weight và bias, đồng thời có buffer `running_mean`, `running_var`. Trong train mode, batch mới có thể cập nhật buffer dù không tính gradient; chuyển sang eval thì dùng buffer đã tích lũy. Với encoder cần giữ cố định, ta giữ cả buffer bằng eval.

Khi batch fine-tune nhỏ, như vài ảnh độ phân giải cao mỗi batch, thống kê batch rất nhiều, nên nhiều hệ thống phát hiện vật thể giữ BatchNorm của backbone cố định ngay cả khi các trọng số khác được học.

Thí nghiệm sau đặt mọi tham số BatchNorm thành không học, nhưng vẫn cho một batch qua train mode. Trung bình batch là $(3, 6)$, `momentum=0.5`, trung bình cũ là $(0, 0)$, nên trung bình mới là $(1.5, 3)$. Trong BatchNorm, momentum này là hệ số cập nhật thống kê, khác nghĩa với momentum của optimizer.

Python

```
bn = nn.BatchNorm1d(2, momentum=0.5)
for parameter in bn.parameters():
    parameter.requires_grad_(False)
batch = torch.tensor([[2., 4.], [4., 8.]])
bn.train()
with torch.no_grad():
    bn(batch)
after_train = bn.running_mean.clone()
bn.eval()
with torch.no_grad():
    bn(batch + 100)
print("running mean sau train:", after_train.tolist())
print("eval giu buffer:", torch.equal(after_train, bn.running_mean))
print("khop tinh tay:", torch.allclose(after_train, batch.mean(dim=0) * 0.5))
```



mọi tham số đã `requires_grad=False`, đang trong `no_grad()`, mà buffer vẫn đổi

Đóng băng tham số nhưng ở `train()`, BatchNorm vẫn cập nhật `running_mean`.

Dòng đầu trực tiếp phản bác hiểu lầm “không có gradient thì không có gì thay đổi”. Dropout lại không có running mean; eval làm nó bỏ thao tác che ngẫu nhiên. Khi gọi `model.train()` ở đầu epoch, các module con cũng chuyển train; nếu encoder phải đứng yên, gọi `encoder.eval()` sau đó. Còn `no_grad()` không được bọc phần đang học (head, adapter): autograd phải ghi lại các phép tính ấy thì backward mới tới được tham số của chúng.

HỎI NHANH

Đóng băng weight và bias của BatchNorm, rồi cho một batch mới đi qua ở chế độ `train()`, bên trong `no_grad()`. Running mean có đổi không?

- A Không, vì mọi tham số đều có `requires_grad=False`
- B Không, vì `no_grad()` chặn mọi thay đổi trong forward
- C Có, vì buffer tự cập nhật ngay trong forward
- D Có, nhưng chỉ khi gọi `optimizer.step()` sau đó

HỎI NHANH

`model = nn.Sequential(encoder, head)`, và `encoder` có BatchNorm cần giữ nguyên. Đoạn sau có giữ được running mean của `encoder` không?

```
encoder.eval()          # gọi một lần trước vòng lặp
for epoch in range(10):
    model.train()
    for x, y in loader:
        ...               # forward, backward, step cho head
```

- A Có, vì `encoder.eval()` đã được gọi từ trước
- B Không, vì `model.train()` lan xuống cả `encoder`
- C Có, nếu tham số của `encoder` đã `requires_grad=False`
- D Không, vì `optimizer` cập nhật buffer ở mỗi `step()`

Khi đi thi

- Học head trên `encoder` đóng băng trước, để có mốc so sánh và một bài nộp hợp lệ.

- Phần pretrained dùng learning rate nhỏ; model cũ BERT thường từ $2 \cdot 10^{-5}$ đến $5 \cdot 10^{-5}$.
- Dùng đúng processor của checkpoint: kích thước ảnh, mean/std, tokenizer.

Tóm tắt

- Fine-tuning học tiếp model pretrained trên nhiệm vụ mới, với head mới đúng số lớp.
- Bắt đầu bằng head-only trên encoder đóng băng; rồi mở thêm encoder với learning rate nhỏ hơn head, chọn checkpoint bằng validation.
- Đóng băng tham số, chế độ train/eval, `no_grad()` và tập tham số của optimizer là bốn công tắc khác nhau; BatchNorm còn có buffer đổi ngay trong forward ở chế độ train.
- Trên ví dụ chữ số, mở encoder nâng validation accuracy từ 0.9020 lên 0.9379.

Kiểm tra cuối bài

CÂU 1

Một đội có 3000 bình luận tiếng Anh gán ba nhãn cảm xúc, và một GPU dùng trong hai giờ. Nên xuất phát từ đâu?

- A BERT pretrained, thay head ba lớp rồi fine-tune nhẹ
- B Transformer cũ BERT khởi tạo ngẫu nhiên, học từ đầu trên 3000 câu
- C ResNet-50 pretrained trên ImageNet, thay head ba lớp rồi fine-tune
- D AlexNet học từ đầu trên vector đếm từ của 3000 bình luận

PHÒNG THEO [NOAI 2025 Singapore](#) · Phần 1 · Câu 17

CÂU 2

Ta muốn giữ cố định encoder khi học head, đồng thời tiết kiệm bộ nhớ và thời gian của backward. Cách nào đạt được cả hai?

- A Đặt learning rate 0 cho nhóm tham số của encoder
- B Gọi `requires_grad_(False)` cho các tham số của encoder
- C Gọi `encoder.eval()` trước mỗi epoch train
- D Gọi `model.freeze()` một lần trước vòng lặp train

PHÒNG THEO [NOAI 2026 Singapore](#) · Vòng sơ loại · Câu 132

CÂU 3

Ba đội cùng dùng một BERT pretrained cho bài phân loại. Đội A viết prompt kèm 5 ví dụ có nhãn. Đội B đóng băng BERT, học một head mới trên vector [CLS]. Đội C cập nhật cả trọng số BERT lẫn head trên 2000 câu có nhãn. Đội nào đã làm thay đổi trọng số pretrained?

- A Cả ba đội, vì đều dùng dữ liệu có nhãn của bài
- B Chỉ đội C
- C Đội B và đội C, vì cả hai đều huấn luyện
- D Đội A và đội C, vì cả hai đều đưa ví dụ vào mô hình

PHÒNG THEO [NOAI 2026 Singapore](#) · Vòng sơ loại · Câu 245

CÂU 4

Fine-tune ResNet-50 pretrained cho 10 lớp:

```
model = resnet50(weights="IMAGENET1K_V2")
for p in model.parameters():
    p.requires_grad = False
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
model.fc = nn.Linear(2048, 10)
# vòng train: zero_grad, forward, loss.backward(), optimizer.step()
```

Sau khi train, validation accuracy chỉ khoảng 10%, và `model.fc.weight` giống hệt lúc khởi tạo. Lỗi ở đâu?

- A Optimizer được tạo trước khi thay `fc`, nên không chứa `fc` mới
- B Backbone đóng băng chặn gradient, không cho đi tới `fc`
- C Learning rate $1e-3$ quá nhỏ cho một head khởi tạo ngẫu nhiên
- D Optimizer nhận tham số đóng băng nên bỏ qua cả nhóm

PHÒNG THEO [NOAI 2026 Singapore](#) · [Chung kết](#) · [Câu 15](#)

CÂU 5

Fine-tune BERT-base phân loại 5 lớp trên vài nghìn câu: batch 32, max_length=128, AdamW với learning rate 10^{-3} cho toàn bộ mô hình, 3 epoch. Validation loss giảm sau epoch 1 rồi tăng mạnh, còn train loss vẫn giảm đều. Nguyên nhân khả dĩ nhất là gì?

- A Batch 32 quá nhỏ, cần gradient accumulation lên 256
- B max_length=128 cắt mất chữ, cần tăng lên 512 token
- C Thiếu warmup, thêm warmup là đủ, giữ learning rate cũ
- D Learning rate quá lớn đối với trọng số pretrained

PHÒNG THEO NOAI 2026 Singapore · Chung kết · Câu 21

CÂU 6

Từ head-only, đội mở thêm layer4 của ResNet-50, khoảng 15 triệu tham số, rồi train bằng Adam ở float32. Phần bộ nhớ tăng thêm cho gradient và trạng thái optimizer của những tham số vừa mở, chưa tính activation, là khoảng bao nhiêu MB, với 1 MB bằng 10^6 byte?

- A 60
- B 120
- C 180
- D 240

CÂU 7

BatchNorm có running mean cũ là 2, batch mean là 8, và $\text{momentum}=0.25$ theo quy ước của PyTorch. Running mean mới bằng bao nhiêu?

A 3.5

B 5

C 6.5

D 8

CÂU 8

Hai checkpoint A và B có validation loss lần lượt là 0.30 và 0.35, accuracy là 0.88 và 0.90. Quy trình đã định từ trước là chọn theo loss nhỏ nhất. Chọn checkpoint nào, và báo accuracy nào?

A Chọn A, báo accuracy 0.90 của B

B Chọn A, báo accuracy 0.88

C Chọn B, báo accuracy 0.90

D Trộn trọng số của A và B, báo 0.90

CÂU 9

Học head, còn encoder được giữ cố định chỉ bằng cách không đưa vào optimizer; tham số của nó vẫn `requires_grad=True`. Muốn khỏi tốn bộ nhớ cho graph của encoder bằng `torch.no_grad()`. Đặt khối ấy ở đâu?

- A Bọc cả dòng `loss = criterion(head(encoder(x)), y)` trong khối
- B Chỉ bọc `head(z)`, còn `z = encoder(x)` tính bên ngoài khối
- C Không cần khối, gọi `encoder.eval()` là đủ tiết kiệm
- D Chỉ bọc `z = encoder(x)`, còn head và loss tính bên ngoài khối

CÂU 10

Head-only đạt accuracy 0.86 trên train và 0.84 trên validation. Mở toàn bộ encoder thì đạt 0.99 và 0.79, trên cùng cách chia. Bước tiếp theo hợp lý là gì?

- A Chọn full fine-tuning, vì nó học được nhiều tham số hơn
- B Mở thêm tham số, vì train accuracy vẫn chưa bằng 1
- C Mở ít hơn hoặc giảm learning rate của encoder, rồi so lại với head-only
- D Chọn full fine-tuning, nếu test công khai cho điểm cao hơn

Đọc thêm

- [PyTorch — transfer learning tutorial](#): hai nhánh fixed feature extractor và fine-tuning cùng vòng train/validation.
- [Dive into Deep Learning — fine-tuning](#): cách thay head và dùng bước cập nhật khác nhau cho trọng số cũ/mới.
- [PyTorch — BatchNorm1d](#): running statistics, affine parameters và nghĩa của momentum.
- [PyTorch — autograd mechanics](#): quan hệ giữa freeze, grad mode và eval mode.

- Transformers — sequence classification: cùng quy trình với model văn bản: tokenizer, head phân loại mới, learning rate nhỏ và chấm validation.
- Devlin và cộng sự — BERT, phụ lục A.3: dải learning rate, batch size và số epoch khi fine-tune.
- Howard, Ruder — ULMFiT: mở dần từng tầng và learning rate khác nhau theo tầng khi fine-tune model ngôn ngữ.

Tải gói recipe của chương — gồm các module phụ và hướng dẫn chuẩn bị asset.