Contents

[Google Colab: 1](#_Toc160534853)

[Torch: 3](#_Toc160534854)

[Processing Unit – Bộ Xử Lí: 7](#_Toc160534855)

[Computation Graph – Sơ Đồ Tính Toán: 9](#_Toc160534856)

[Tensor: 10](#_Toc160534857)

[Quick Codes – Mã Nhanh: 16](#_Toc160534858)

[Neural Network – Mạng Thần Kinh: 18](#_Toc160534859)

[Architecture Optimization – Tối Ưu Mô Hình: 39](#_Toc160534860)

[Functional – Hàm: 41](#_Toc160534861)

[Optimizer – Tối Ưu Hóa: 42](#_Toc160534862)

[Torch Vision: 42](#_Toc160534863)

[Data – Dữ Liệu: 47](#_Toc160534864)

Google Colab:

1. Google Colab Dùng Để Làm Gì?

* Trang Web dùng để lập trình và Train Model trên máy của Google thay vì máy của mình
* Mọi thư viện đều được cài đặt trên máy của Google

1. Cách Truy Cập?

* <https://colab.research.google.com/>
* Đăng nhập = tài khoảng Google + tạo New Notebook + tiến hành lập trình

1. Đặt Tên Và Lưu File?

* Click vào tiêu đề + đổi tên + vào File + chọn Save

1. Cách Hiện Số Dòng?

* Vào Tab “Tools” + chọn “Settings” + vào thẻ “Editor” +

Check “Show line numbers” + Click “Save”

1. Có Phải Sau Khi Chạy Xong 1 Đoạn Code Trong Google Colab Thì Tất Cả Thư Viện, Biến Sẽ Được Lưu Lại?

* Phải, sau khi chạy xong, thì bạn có thể sử dụng thư viện và biến đó ở bất kì ô Code nào bạn muốn mà không cần khai báo lại
* Muốn xem các biến đã lưu thì chọn mục “Variables” ở bên trái, biểu tượng “{x}”, cửa sổ này có thể dùng để Debug, xem giá trị cuối cùng của biến mà không cần in ra

1. RAM Và Disk Trên Google Colab?

* Muốn xem dung lượng RAM và Disk tối đa và đã dùng thì di chuột lên biểu tượng của RAM và Disk góc phải trên
* Dung lượng RAM chủ yếu đến từ các biến đã lưu, gán lại giá trị hoặc xóa chúng đi để giảm dung lượng
* Dung lượng Disk đến từ các File bạn lưu, xóa chúng đi để giảm dung lượng

1. Cách Xóa Biến Đã Lưu?

* Để xóa tất cả

%reset -f

* Để xóa 1 biến

%reset\_selective -f <Tên Biến>

* Ví dụ

%reset\_selective -f foo

1. Dùng Terminal Trên Google Colab?

* Đéo dùng được, chỉ tài khoản Pro mới dùng được

1. Cấu Trúc Đường Dẫn Trong Google Colab?

* Ban đầu ở cửa sổ “Files”, bạn đang ở trong thư mục “content”, và bạn thấy có 1 thư mục “sample\_data” chứa sẵn 1 số Dataset cơ bản
* Trong thư mục “content” bạn có thể tạo sửa xóa thư mục hoặc File tùy ý, đây cũng là thư mục làm việc hiện tại, và sẽ không thay đổi cho đến khi bạn nhập lệnh thay đổi
* Code sẽ chạy trong thư mục làm việc hiện tại
* Click Up One Level để lên thư mục cha, đây là thư mục chứa tất cả mọi thứ, không có tên
* Trong thư mục cha bạn sẽ thấy thư mục “root”
* Tất cả các thư mục và File chưa bị mất đi khi thoát khỏi Google Colab, đợi 1 lúc lâu mà không làm việc trên đó nữa thì mới bị mất

1. Cách Lưu Vĩnh Viễn File Trên 1 Note Book?

* Click “Mount Drive” + chờ Restart Runtime
* Khi này, sẽ có 1 thư mục “drive” được tạo trong thư mục “content”, trong đó có thư mục “MyDrive”, trong này sẽ có tất cả File trên Drive của bạn
* Vì Drive tồn tại vĩnh viễn nên bạn có thể lưu File vào thư mục “MyDrive” để nó cũng tồn tại vĩnh viễn
* Khi sắp kết thúc làm việc thì mới đưa dữ liệu vào Drive, còn trong khi làm việc thì cứ để nó ở ngoài

1. Giải Nén Toàn Bộ File Zip Trong Thư Mục Làm Việc Hiện Tại?

* Chạy lệnh sau

!unzip \\*.zip && rm \*.zip

* Các File Zip ban đầu cũng được xóa

1. Cách Di Chuyển 1 File Hoặc Thư Mục Sang Thư Mục Khác?

* Nhấn giữ File hoặc thư mục rồi kéo thả sang thư mục khác

1. Thay Đổi Thư Mục Làm Việc Hiện Tại Sang Thư Mục Khác?

* Chuyển sang thư mục “root” từ thư mục bất kì

%cd

* Chuyển sang thư mục khác cùng cấp

%cd /<Tên Thư Mục Muốn Chuyển Sang>

* Ví dụ
* Thư mục làm việc hiện tại là /content/foo

%cd /bar

* Thư mục làm việc hiện tại là /content/bar
* Chuyển sang thư mục con

%cd <Đường Dẫn Từ Thư Mục Hiện Tại Đến Thư Mục Con>

* Ví dụ
* Thư mục làm việc hiện tại là /root/foo

%cd bar/barfoo

* Thư mục làm việc hiện tại là /root/foo/bar/barfoo

1. Xóa 1 Thư Mục Không Rỗng Trong Google Colab?

* Xóa thư mục trong thư mục làm việc hiện tại

!rm -r <Tên Thư Mục Muốn Xóa>

1. Lỡ Click Up One Level Rồi Thì Muốn Trở Về Thì Sao?

* Đéo trở về được

1. Phím Tắt Để Chạy Đoạn Code Hiện Tại?

* Ctrl + Enter

1. Không Kết Nối Được Runtime?

* Đổi từ GPU hay TPU sang CPU

1. Cú Pháp Tải Thư Viện?

* !pip install <Tên Thư Viện>
* Ví dụ
* !pip install pygame

1. Dãy Lệnh Tải Các Game Để Test AI Học Tăng Cường?

!pip install gym gym[atari] gym[accept-rom-license] pyvirtualdisplay pygame

!apt-get update

!apt-get install xvfb

1. Những Thư Viện Đã Có Sẵn Trong Google Colab?

keras, torch, tensorflow, jax

* Những thư viện này đều là bản mới nhất

1. Tải Nhanh Những Thư Viện Xử Lí Hình Ảnh Chưa Có?

!pip install mediapipe

1. Cách Chọn GPU Và TPU?

* Vào Tab Runtime + chọn Change Runtime Type + tại Hardware Accelerator chọn GPU hoặc TPU + Click Save
* Việc làm này sẽ xóa toàn bộ dữ liệu đã tải xuống trước đó

1. Câu Lệnh Dừng Code Trên Google Colab?

* Trước tiên khai báo thư viện
* import sys
* Sau đó muốn dừng chỗ nào thì đặt câu lệnh này ngay chỗ đó
* sys.exit()

1. Muốn Đo Thời Gian Trung Bình Thực Thi 1 Câu Lệnh Trên Google Colab?

* Đặt trước câu lệnh dòng Code sau
* %timeit –r <Số Lần Chạy> –n <Số Lần Thực Thi Câu Lệnh Lệnh Mỗi Lần Chạy>
* Ví dụ, câu lệnh sum sẽ chạy tổng cộng 1000 lần và đo thời gian trung bình thực hiện câu lệnh này
* %timeit –r 10 –n 100 sum(range(20))
* Màn hình Output
* 475 ns ± 43.5 ns per loop (mean ± std. dev. of 10 runs, 100 loops each)

1. Chỉ Muốn Đo 1 Lần Thời Gian Thực Hiện Câu Lệnh Tại Thời Điểm Hiện Tại?

* %time <Câu Lệnh>
* Ví dụ
* %time sum(range(100))
* Màn hình Output
* CPU times: user 8 µs, sys: 1 µs, total: 9 µs
* Wall time: 12.4 µs
* User chỉ bao gồm thời gian CPU thực thi câu lệnh
* Sys là thời gian CPU xử lí mấy cái tác vụ hệ thống râu ria không liên quan đến câu lệnh
* Wall Time là thời gian thực tế của người quan sát khi chạy câu lệnh từ đầu đến cuối, bao gồm luôn thời gian chờ đợi Input nhập vào, Output xuất ra

Torch:

1. Cách Cài Đặt?

pip3 install torch

1. Cách Import?

import torch

1. Dấu “@” Có Ý Nghĩa Gì?

* Phép nhân Batch ma trận hoặc tích vô hướng của 1D Tensor
* Ví dụ

foo = bar @ boo

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| bar | (2, 3, 4, 5, 6) |
| boo | (2, 1, 4, 6, 8) |
| foo | (2, 3, 4, 5, 8) |

* Cách Để Hiện Số Thập Phân Thay Vì Số Khoa Học?

torch.set\_printoptions(

precision = <Số Chữ Số Thập Phân Sau Dấu Phẩy>,

sci\_mode = False

)

1. Thứ tự ưu tiên các toán tử?

|  |  |
| --- | --- |
| 1 | \*\* |
| 2 | \*, @ |
|  |  |
|  |  |
|  |  |
|  |  |

1. Cách Hiện Full Tensor Thay Vì Bị Rút Gọn Thành Dấu “…”?

torch.set\_printoptions(threshold = <Số Phần Tử Tối Đa Được Hiện>)

1. Trả Về Tensor Là Cấp Số Cộng?

torch.arange(<Start>, <End>, <Step>)

* Cấp số cộng bắt đầu từ <Start> và kết thúc trước <End>, công sai = <Step>
* Mặc định

|  |  |
| --- | --- |
| <Start> | 0 |
| <Step> | 1 |

* Ví dụ

foo = torch.arange(2, 5, 0.5)

* Ta có

|  |  |
| --- | --- |
| foo | [2, 2.5, 3, 3.5, 4, 4.5] |

1. Trả Về Tensor Có Giá Trị Ngẫu Nhiên Thuộc Phân Phối Chuẩn Tắc?

torch.randn(<Shape>)

* Ví dụ

torch.randn(2, 3, 4)

1. Trả Về Tensor Có Giá Trị Ngẫu Nhiên Là Số Nguyên?

torch.randint(<Min>, <Max>, <Shape>)

* Mặc định

|  |  |
| --- | --- |
| <Min> | 0 |

* Tensor được trả về sẽ có giá trị nguyên từ <Min> đến <Max> – 1
* Ví dụ

foo = torch.randint(8, (2, 4))

* Ta có

|  |  |
| --- | --- |
| foo | [[1, 0, 2, 7],  [5, 4, 1, 3]] |

1. Trả Về Tensor Có Giá Trị Ngẫu Nhiên Từ 0 Đến 1?

torch.rand(<Shape>)

* Ví dụ

torch.rand(4, 5)

* Để tạo Tensor có giá trị ngẫu nhiên là True hoặc False

torch.rand(<Shape>) > 0.5

1. Cách Để Mỗi Lần Chạy Lại Chương Trình Thì Giá Trị Ngẫu Nhiên Không Thay Đổi?

torch.manual\_seed(<Seed>)

* <Seed> là 1 số nguyên, mỗi <Seed> khác nhau sẽ tạo chuỗi ngẫu nhiên khác nhau và không thay đổi khi chạy lại chương trình

1. Trả Về Tensor Chỉ Chứa 1 Giá Trị Nào Đó?

torch.full(<Shape>, <Giá trị>)

* Ví dụ

torch.full((2, 3), 1.5)

* Lệnh này còn có thể dùng để Fill giá trị cho 1 lớp trong Tensor
* Ví dụ

x = torch.zeros(2, 3, 4)

x[0] = torch.full((3, 4), 1.5)

* Tấm ảnh trên cùng sẽ chứa toàn giá trị 1.5, tấm ảnh phía dưới vẫn chứa toàn giá trị 0

1. Trả Về Phiên Bản Ma Trận Tam Giác Trên Của 1 Ma Trận Nào Đó?

torch.triu(<Batch Ma Trận>, <Offset Đường Chéo Chính>)

* <Offset Đường Chéo Chính> mặc định = 0
* Ví dụ

foo = torch.triu(bar, 2)

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 2, 3, 4, 5, 6],  [7, 8, 9, 1, 2, 3],  [4, 5, 6, 7, 8, 9]  [1, 2, 3, 4, 5, 6]] |
| foo | [[0, 0, 3, 4, 5, 6],  [0, 0, 0, 1, 2, 3],  [0, 0, 0, 0, 8, 9]  [0, 0, 0, 0, 0, 6]] |

1. Trả Về Tensor Được Ghép Từ Nhiều Tensor?

<Tensor Ghép> = torch.cat(<Các Tensor>, dim = <Chiều Ghép>)

* <Các Tensor> là 1 Iterable có các phần tử là các Tensor cùng số chiều
* Tưởng tượng 2 Tensor là 2 xấp ảnh, bây giờ ta ghép 2 xấp ảnh này với nhau theo chiều ngang, dọc, … như thế nào đó để mặt ghép vừa khít với nhau, tạo ra xấp ảnh mới lớn hơn
* Ví dụ

z = torch.cat((x, y), dim = 1)

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| x | (3, 4, 5) |
| y | (3, 8, 5) |
| z | (3, 12, 5) |

1. Chồng Tensor Này Lên Tensor Khác?

<Tensor Chồng> = torch.stack(<Các Tensor>, <Chiều Quay Mặt>)

* Mặc định

|  |  |
| --- | --- |
| <Chiều Quay Mặt> | 0 |

* <Các Tensor> là 1 Iterable có các phần tử là các Tensor cùng Shape
* Tưởng tượng 2 Tensor là 2 tấm ảnh trong không gian 2D, đưa chúng vào chiều không gian cao hơn 1 bậc là 3D, sau đó chồng 2 tấm ảnh lên nhau, tấm ảnh thứ 2 đặt ở dưới, rồi xoay xấp ảnh 90 độ sao cho mặt của chúng hướng vào <Chiều Quay Mặt> nếu chưa, cuối cùng là Flip để phần tử đầu tiên được đọc là phần tử đầu tiên của tấm ảnh thứ 1
* Ví dụ

z = torch.stack((x, y), 1)

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| x | (4, 5) |
| y | (4, 5) |
| z | (4, 2, 5) |

1. Sử Dụng Hàm Softmax?

<Tensor Xác Suất> = torch.softmax(<Tensor>, dim = <Chiều Tính Softmax>)

1. Sử Dụng Hàm Tanh?

* <Kết Quả> = torch.tanh(<Tensor>)

1. Sampling Từ 1 Phân Phối Rời Rạc?

<Sample> = torch.multinomial(

<Tensor Trọng Số>, <Kích Thước Sample>, replacement = True

)

* <Sample> bản chất = <Tensor Trọng Số> nhưng mỗi phần tử là mảng 1 chiều trong đó, gọi là A, bị thay thế bởi mảng 1 chiều khác, gọi là B, B có số phần tử = <Kích Thước Sample>
* A ban đầu được chuẩn hóa để tổng các phần tử của nó = 1, gọi X là biến ngẫu nhiên sao cho P(X = i) = A[i], khi đó B chứa các phần tử được lấy từ phân phối của X
* Ví dụ

foo = torch.multinomial(bar, 7, replacement = True)

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 2, 9, 4],  [0, 0, 5, 0],  [4, 0, 0, 4]] |
| foo | [[3, 2, 1, 2, 1, 3, 2],  [2, 2, 2, 2, 2, 2, 2],  [0, 3, 0, 3, 3, 0, 0]] |

1. Lượng Tử Hóa Nhiều Tensor?

<Các Tensor Lượng Tử> = torch.quantize\_per\_tensor(

<Các Tensor>, <Các Tham Số Scale>, <Các Tham Số Offset>, <Kiểu Lượng Tử>

)

* Các giá trị cho <Kiểu Lượng Tử>
* Số nguyên không dấu 8 Bit

torch.quint8

* Số nguyên có dấu 8 Bit

torch.qint8

* Số nguyên có dấu 32 Bit

torch.qint32

* Nếu <Các Tensor> là 1 mảng có các phần tử là Tensor, thì <Các Tensor Lượng Tử> cũng vậy, còn nếu chỉ là 1 Tensor, thì <Các Tensor Lượng Tử> cũng vậy
* Mỗi phần tử Scale trong <Các Tham Số Scale> và mỗi phần tử Offset trong

<Các Tham Số Offset> sẽ tương ứng với 1 Tensor trong <Các Tensor>

* Mỗi Tensor lượng tử trong <Các Tensor Lượng Tử> = Tensor tương ứng trong

<Các Tensor> / Scale tương ứng + Offset tương ứng, sau đó làm tròn đến số nguyên gần nhất, rồi quy đổi sang nhị phân tương ứng với <Kiểu Lượng Tử>, nếu giá trị nhị phân vượt quá giới hạn của

<Kiểu Dữ Liệu> thì sẽ bị Clip, ví dụ 111011110 thành 11111111

* Để chuyển từ nhị phân sang giá trị thực tương ứng, ban đầu đổi thành số nguyên tương ứng, sau đó – Offset rồi \* Scale
* Để trả về dạng số nguyên tương ứng

<Tensor Lượng Tử>.int\_repr()

* Trả về Scale tương ứng

<Tensor Lượng Tử>.q\_scale()

* Trả về Offset tương ứng

<Tensor Lượng Tử>.q\_zero\_point()

* Ví dụ

foo, far = torch.quantize\_per\_tensor((boo, bar), scar, oppa, torch.qint8)

ifoo = foo.int\_repr()

ifar = far.int\_repr()

* Ta có

|  |  |
| --- | --- |
| boo | [–2.5, 4.31, 2.6, 5.45, 6] |
| bar | [7.1, 3.25, 2.4, 5.6, 45, 50] |
| scar | [0.1, 0.2] |
| oppa | [–12, 21] |
| ifoo | [–37, 31, 14, 42, 48] |
| ifar | [57, 37, 33, 49, 127, 127] |
| foo | [–2.5, 4.3, 2.6, 5.4, 6] |
| far | [7.2, 3.2, 2.4, 5.6, 21.2, 21.2] |

1. Trả Về Tensor Gốc Từ Nhiều Tensor Lượng Tử?

<Các Tensor Gốc> = torch.dequantize(<Các Tensor Lượng Tử>)

* Cơ chế đảo chiều đã nói ở phần lượng tử hóa Tensor
* <Các Tensor Gốc> có kiểu dữ liệu là Floating Point 32 Bit
* Ví dụ

foo, bar = torch.dequantize((boo, far))

1. Lượng Tử Hóa 1 Tensor 1 Cách Tự Động?

<Tensor Lượng Tử> = torch.quantize\_per\_tensor\_dynamic(

<Tensor>, <Kiểu Lượng Tử>, <Có Giảm 1 Bit Không>

)

* <Kiểu Lượng Tử> phải là số nguyên có hoặc không dấu 8 Bit
* <Scale> và <Offset> sẽ được tính làm sao cho <Scale> nhỏ nhất có thể mà <Offset> vẫn nằm trong miền giá trị của <Kiểu Lượng Tử> trong trường hợp

<Có Giảm 1 Bit Không> = False, nếu = True, thì miền giá trị thu hẹp 1 nửa, ví dụ từ –128 đến 127 còn –64 đến 63, từ 0 đến 255 còn từ 0 đến 127

* Ví dụ

foo = torch.quantize\_per\_tensor\_dynamic(bar, torch.quint8, True)

Processing Unit – Bộ Xử Lí:

1. CPU, GPU, CUDA, TPU?

* Mỗi Core tính toán lần lượt, từng phép tính một, ghép nhiều Core vào nhau để tính toán song song
* Mỗi Processing Unit sẽ có nhiều Core, có nhiệm vụ điều khiển chúng
* CPU chỉ có vài Core, nhưng nó có thể thực hiện được nhiều nhiệm vụ khác nhau, mặc định khi tạo Tensor thì nó ở CPU
* GPU có hàng nghìn Core, dùng để thực hiện tính toán song song, CUDA là 1 mô hình giúp bạn lập trình để thực hiện tính toán trên GPU bằng cách cung cấp các thư viện giúp bạn giao tiếp với GPU
* TPU có hàng nghìn Tensor Core, được thiết kê chỉ cho 1 mục đích duy nhất là tính toán với Tensor, nhanh gấp nhiều lần GPU
* 2 Tensor chỉ có thể tính toán được với nhau nếu chúng cùng ở trên 1 CPU hoặc cùng ở trên 1 GPU hoặc cùng ở trên 1 Tensor Core của TPU

1. Cách Import Thư Viện CUDA Trong Pytorch?

from torch import cuda

1. Trả Về Số GPU Hiện Tại Đang Có Trong Máy?

cuda.device\_count()

1. Trả Về Tên Của GPU Nào Đó Trên Máy?

cuda.get\_device\_name(<Index>)

* Các GPU trên máy sẽ được đánh <Index> bắt đầu từ 0
* Ví dụ

foo = cuda.get\_device\_name(0)

* Ta có

|  |  |
| --- | --- |
| Foo | "Tesla T4" |

1. Tạo Tensor Trên GPU?

<Tensor> = torch.tensor(<Dữ Liệu>, device = "cuda")

* <Tensor> sẽ tự động tạo trên GPU có <Index> = 0

1. Copy Tensor Từ CPU Sang GPU?

<GPU Tensor> = <CPU Tensor>.cuda()

* <GPU Tensor> sẽ ở trên GPU có <Index> = 0
* Bạn cũng có thể Copy Module từ CPU sang GPU với cú pháp tương tự

1. Cài đặt thư viện tương tác với TPU?

* Chuyển Runtime sang TPU, sau đó chạy lệnh sau để cài đặt thư viện tương tác với TPU

!pip install cloud-tpu-client==0.10 <https://storage.googleapis.com/tpupytorch/wheels/colab/torch_xla-2.0-cp310-cp310-linux_x86_64.whl>

* Sau đó Restart lại Runtime
* Bạn chỉ được sử dụng 1 TPU trong Google Colab mỗi phiên làm việc
* Mỗi Tensor Core của TPU này sẽ được xem là 1 Device riêng, tổng cộng có 8 Core được đánh <Index> từ 1 đến 8

1. Cách Import Thư Viện Tương Tác Với TPU?

import torch\_xla as tx

1. Trả Về Core Nào Đó?

<Device> = torch\_xla.core.xla\_model.xla\_device(<Index>)

* Để tạo 1 Tensor trên Core này

<Tensor> = torch.tensor(<Dữ Liệu>, device = <Device>)

* Copy Tensor từ nơi nào đó vào Core này

<Tensor Mới> = <Tensor Cũ>.to(<Device>)

* Bạn cũng có thể Copy Module từ nơi nào đó sang Core này với cú pháp tương tự

Computation Graph – Sơ Đồ Tính Toán:

1. Cách Torch Tính Gradient?

* Lưu ý khi bạn gán 1 Tensor vào 1 biến, thực chất biến đó là 1 Pointer chĩa vào Tensor đang ở đâu đó trong bộ nhớ
* Nó tính Gradient = Chain Rule, đúng vậy, bắt đầu từ Node chỉ định, nó sẽ xác định xem Node này được Node nào nhắm tới, giả sử Node q = Node w \* Node z, w có giá trị hiện tại = 10, z có giá trị hiện tại = 9, thì Node w và Node z chính là những Node đang nhắm tới Node y, ở đây, dựa vào toán tử “\*”, nó xác định được q’ = w’z + wz’ = 9w’ + 10z’, dựa vào đây, nó truyền 9 cho Node w và 10 cho Node z, giả sử Node z = x2, x có giá trị hiện tại = 3, nó sẽ tính 10z’ = 10(2xx’) = 60x’, nó tiếp tục truyền 60 cho Node x, x là Node cuối cùng, gọi là Leaf, nghĩa là không còn Node nào nhắm tới nó nữa, nên Gradient hiện tại của x là 0 được + 60, nãy giờ nói về việc lan truyền của z quên mẹ w, giả sử Node w = 3x + y, y có giá trị hiện tại = 1, nó sẽ tính 9w’ = 9(3x’ + y’) = 27x’ + 9y’, nó truyền 27 cho Node x, nhưng như đã nói thì nó sẽ dừng lại tại đây và 27 được + vào Gradient của Node x, làm cho Gradient của nó = 60 + 27 = 87, quên nữa là 9 được truyền cho Node y nhưng đây cũng là Leaf nên Gradient của y là 0 được + 9, vậy sau khi tính toán ta có Gradient của x = 87 và Gradient của y = 9, nghiệm lại thấy biểu thức q = wz = (3x + y)x2 có đạo hàm theo x và theo y tại x = 3, và y = 1 đúng thật là = 87 và 9
* Nếu điều kiện có xuất hiện trong quá trình tính toán thì cũng đéo quan trọng, giả dụ có điều kiện nếu x < 5 thì z = x2, còn không thì z = 4x, rõ ràng trong đồ thị tính toán khi x = 3 sẽ chỉ có Node z = x2, nên nó sẽ tính Gradient theo hàm này, còn cái z = 4x đéo quan tâm
* Lưu ý q phải là Tensor với duy nhất 1 giá trị, bao nhiêu chiều cũng được

1. Đưa Tensor Vào Sơ Đồ Tính Toán?

<Tensor>.require\_grad\_()

* Sơ đồ tính toán chỉ chứa địa chỉ của Tensor
* <Tensor> khi này gọi là Leaf, chứa Gradient khi lan truyền ngược
* Tensor nào đó được tính từ <Tensor> thì địa chỉ của nó cũng sẽ được đưa vào sơ đồ tính toán, nhưng nếu bạn tính Tensor đó trong

with torch.no\_grad():

* Thì nó không được đưa vào sơ đồ tính toán
* Các Pointer không được phép thực hiện tính toán In Place khi địa chỉ của Tensor nó chĩa vào đang ở trong sơ đồ tính toán

1. Thực Hiện Lan Truyền Ngược?

<Tensor>.backward(retain\_graph = <Có Giữ Sơ Đồ Không>)

* Mặc định

|  |  |
| --- | --- |
| <Có Giữ Sơ Đồ Không> | False |

* Khi này, tất cả các Leaf của <Tensor> sẽ nhận được Gradient, đồng thời trong phần sơ đồ tính toán với <Tensor> làm gốc, tất cả các địa chỉ sẽ bị xóa nếu <Có Giữ Sơ Đồ Không> = False, do đó bạn không thể lan truyền ngược lần nữa, nếu = True, thì toàn bộ địa chỉ được giữ lại

1. Truy Cập Gradient Của 1 Tensor?

<Pointer Trỏ Đến Gradient> = <Tensor>.grad

1. Cách Tạo Ảnh Biểu Đồ Tính Toán?

* Giả sử bạn có Node “foo”, và bạn muốn Show tất cả các Node có thể đi đến Node “foo”, dưới dạng biểu đồ, nhập

from torchviz import make\_dot

<Biểu Đồ> = make\_dot(foo, <Danh Sách Các Leaf>)

<Biểu Đồ>.view(<Tên File>)

* <Danh Sách Các Leaf> là 1 Dictionary với Key là tên sẽ được hiển thị lên biểu đồ của Leaf và giá trị là Tensor ứng với Leaf đó
* <Tên File> là đường dẫn đến File ảnh, không có phần mở rộng
* Sau khi chạy đoạn Code trên, sẽ xuất hiện 2 File, 1 không có phần mở rộng, 2 là có phần mở rộng “.pdf” chứa ảnh

1. Cách Tính Thủ Công Gradient Thay Vì Dùng Backward?

from torch.autograd import grad

<Các Gradient> = grad(

<Output>,

<Danh Sách Các Biến Cần Tính Gradient>,

create\_graph = True

)

* <Các Gradient> là 1 Tuple chứa các phần tử là Gradient tương ứng với

<Danh sách các biến cần tính Gradient>

* Đặt “create\_graph = True” để tạo ra 1 sơ đồ tính toán mới, không giống như “backward()”, sơ đồ tính toán cũ dùng để tính toán Gradient không bị xóa, sơ đồ tính toán mới có gốc là <Các Gradient>, và các Node liên quan
* Ví dụ
* x là Leaf, y = x + 1, thì khi tính z = dy / dx, và “create\_graph = True”, thì rõ ràng

y = 1 không quan tâm x, nên trong sơ đồ tính toán mới không có x, nếu y = x2, thì rõ ràng y = 2x, trong sơ đồ tính toán mới có Node x gắn Node z, nhờ vậy, ta có thể tính tính dz / dx hay là đạo hàm bậc 2 của y so với x một lần nữa

* Đặt “create\_graph = False” nếu không muốn tạo sơ đồ tính toán mới

1. Cách Tạo Ra Tensor Mới Không Nằm Trong Sơ Đồ Tính Toán Cùng Chia Sẻ Dữ Liệu Với Tensor Cũ Nằm Trong Sơ Đồ Tính Toán?

<Tensor Mới> = <Tensor Cũ>.detach()

* Câu lệnh trên cũng có thể chuyển Parameter thành Tensor thường

Tensor:

1. Cách Import?

from torch import tensor

1. Các Kiểu Dữ Liệu Của Tensor?

* Số nguyên

torch.<Có Dấu Không>int<Số Bit>

* Mặc định có dấu, nếu <Có Dấu Không> = u thì không dấu, chỉ có tác dụng khi <Số Bit> = 8
* <Số Bit> = 8, 16, 32, 64
* Ví dụ

torch.int16

* Số Floating Point

torch.<Exponent Hay Mantissa>float<Số Bit>

* <Số Bit> = 16, 32, 64
* Nếu <Số Bit> = 16, thì dùng 5 Bit làm Exponent, 10 Bit làm Mantissa
* Nếu <Exponent Hay Mantissa> = b thì 8 Bit làm Exponent, 7 Bit làm Mantissa khi

<Số Bit> = 16

* Ví dụ

torch.bfloat16

* Logic

torch.bool

1. Khi Gán Giá Trị Có Kiểu Dữ Liệu Khác Kiểu Dữ Liệu Của Tensor Vào Tensor Thì Chuyện Gì Sẽ Xảy Ra?

* Giá trị được gán vào sẽ được ép thành kiểu dữ liệu của Tensor

1. Chuyển Kiểu Dữ Liệu Của Tensor?

<Tensor Có Kiểu Dữ Liệu Mới> = <Tensor>.<Kiểu Dữ Liệu>()

* Ví dụ

foo = bar.int()

* Ta có

|  |  |
| --- | --- |
| bar | [True, False, True] |
| foo | [1, 0, 1] |

1. Cách Hình Dung Tensor 3 Chiều?

* Tensor 3 chiều có Shape là (a, b, c), tưởng tượng nó như 1 xấp ảnh đặt trước mặt bạn, và đặt nó khít vào góc tường bên tay trái của bạn, gọi đây là vị trí nằm mặc định, chiều cao xấp ảnh = a, chiều ngang = c, chiều hướng tới mặt bạn = b, đọc giá trị Tensor cũng giống như xem ảnh, từ trái qua phải, từ xa tới gần, từ trên xuống dưới
* Mọi phép biến đổi Shape Tensor sau này cứ liên tưởng tới góc tường và xấp ảnh, dù nó có biến đổi kiểu gì thì giá trị đầu tiên được đọc luôn giống nhau

1. Chuyển Tensor Sang Numpy Array?

<Numpy Array> = <Tensor>.numpy(<Có Copy Không>)

* Mặc định

|  |  |
| --- | --- |
| <Có Copy Không> | False |

* Nếu <Có Copy Không> = False, <Numpy Array> và <Tensor> dùng chung dữ liệu, do đó <Tensor> phải nằm trên CPU, nếu = True, thì không dùng chung, dữ liệu sẽ được Copy tới đâu đó trong CPU

1. Cách Cắt Tensor Nếu Không Biết Số Chiều?

* Dùng dấu “…”
* Ví dụ

foo = bar[:5, …, 2 : 4]

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| bar | (8, 4, 6, 3, 7) |
| foo | (5, 4, 6, 3, 2) |

1. Trả Về Số Chiều Của 1 Tensor?

<Tensor>.dim()

* Ví dụ

foo = bar.dim()

* Ta có

|  |  |
| --- | --- |
| bar | [[[1, 2]]] |
| foo | 3 |

1. Chuyển Vị 1 Batch Tensor?

<Batch Chuyển Vị> = <Batch Tensor>.mT

* <Batch Chuyển Vị> và <Batch Tensor> dùng chung dữ liệu
* Ví dụ

foo = bar.mT

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| bar | (2, 3, 4, 5, 6, 7) |
| foo | (2, 3, 4, 5, 7, 6) |

1. Giới Hạn Giá Trị Của Tensor?

<Tensor Bị Giới Hạn> = <Tensor>.clamp(<Min>, <Max>)

* Ví dụ

foo = bar.clamp(-1, 4)

* Ta có

|  |  |
| --- | --- |
| bar | [[4, 9, 3, 2, –5],  [–2, –1, 1, 2, 3]] |
| foo | [[4, 4, 3, 2, –1],  [–1, –1, 1, 2, 3]] |

1. Quay 1 Tensor?

<Tensor Sau Khi Quay> = <Tensor>.rot90(<Số Lần Quay>, <Mặt Phẳng Quay>)

* Mặc định

|  |  |
| --- | --- |
| <Số Lần Quay> | 1 |
| <Mặt Phẳng Quay> | [0, 1] |

* <Mặt Phẳng Quay> là 1 Iterable gồm 2 phần tử theo thứ tự là A và B
* Tưởng tượng Tensor là xấp ảnh, quay Tensor thì cũng giống như quay xấp ảnh, A và B giống như tia Ox, Oy, Oz, …, giúp xác định 1 mặt phẳng, và xấp ảnh sẽ được quay quanh pháp tuyến của mặt phẳng này, góc quay = 90 \* <Số Lần Quay>, chiều quay từ A quét sang B
* Ví dụ

foo = bar.rot90(1, [1, 2])

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| bar | (2, 3, 4) |
| foo | (2, 4, 3) |

1. 2 Tensor Cộng Trừ Nhân Chia Hadamard Được Khi Nào?

* Trước tiên Tensor có số chiều ít hơn sẽ được ép thêm chiều để 2 Tensor có số chiều = nhau
* Ví dụ
* Tấm ảnh 2D được ép thành tấm ảnh 3D chiều cao = 1
* Tiếp theo nếu 2 Tensor có Shape không = nhau thì xấp ảnh nào nhỏ hơn thì đặt bên trong xấp ảnh lớn hơn, cả 2 xấp ảnh phải sát góc tường, sau đó xấp ảnh nhỏ phân thân lấp đầy xấp ảnh lớn, lưu ý xấp ảnh nhỏ khi đặt trong xấp ảnh lớn, chiều nào không khít thì kích thước chiều đấy phải = 1
* Sau khi ô nào cũng có 2 giá trị, thì tiến hành cộng trừ nhân chia Hadamard
* Việc phân thân Tensor nhỏ lấp kín Tensor lớn gọi là Broadcast
* Tuy nhiên, bản chất thực sự của Broadcast là như sau, ví dụ Tensor A có Shape là

(10, 1, 9, 1, 5), Tensor B có Shape là (1, 7, 9, 3, 1), thì khi 2 Tensor này tính toán với nhau, Shape cuối cùng sẽ là (10, 7, 9, 3, 5), nghĩa là bản chất Singleton Dimension của cả A và B sẽ phân thân để khít với nhau

1. Trả Về Tích Kronecker Giữa 2 Tensor?

<Tensor A>.kron(<Tensor B>)

* Ví dụ

foo = bar.kron(far)

* Ta có

|  |  |  |
| --- | --- | --- |
| bar | far | foo |
| [[2, 4],  [1, 3]] | [5, 6] | [[10, 12, 20, 24],  [5, 6, 15, 18] |

1. Đổi Giá Trị Của Tensor Tại 1 Số Điểm Nào Đó Thành 1 Giá Trị Nào Đó?

<Tensor Sau Khi Đổi Giá Trị> = <Tensor>.masked\_fill(<Mask>, <Giá Trị>)

* <Mask> là Tensor chỉ có giá trị 0 và 1, phải Broadcast được với <Tensor>
* <Tensor Sau Khi Đổi Giá Trị> ban đầu có giá trị của <Tensor>, sau đó những chỗ nào trong <Mask> có giá trị là 1 thì chỗ tương ứng trong <Tensor Sau Khi Đổi Giá Trị> sẽ có giá trị là <Giá Trị>
* Ví dụ

foo = bar.masked\_fill(boo, 9)

* Ta có

|  |  |
| --- | --- |
| bar | [1, 7, 5, 4, 3, 2, 8, 4] |
| boo | [1, 0, 0, 1, 0, 0, 0, 1] |
| foo | [9, 7, 5, 9, 3, 2, 8, 9] |

1. Thay Tất Cả Giá Trị Nào Đó Của 1 Tensor Thành Giá Trị Nào Đó?

<Tensor Muốn Thay>[<Tensor Muốn Thay> == <Giá Trị Cũ>] = <Giá Trị Mới>

* Ví dụ

foo[foo == 1] = 9

* Ta có

|  |  |
| --- | --- |
| foo cũ | [1, 2, 3, 1, 8, 4, 1, 0, 5, 1, 7, 6, 9] |
| foo mới | [9, 2, 3, 9, 8, 4, 9, 0, 5, 9, 7, 6, 9] |

1. Trả Về Tensor N + 1 Chiều Từ Tensor N Chiều?

* <Tensor N + 1 Chiều> = <Tensor N Chiều>.unsqueeze(<Vị Trí Chèn Chiều>)
* <Vị Trí Chèn Chiều> có giá trị từ – n + 1 đến n, giá trị dương thì tính từ trái qua phải, bắt đầu = 0, giá trị âm thì tính từ phải qua trái, bắt đầu = 1
* Khi đưa tensor lên 1 chiều như vậy, thì cấu trúc của nó không thay đổi, ví dụ xấp ảnh 2D đưa lên 3D thì nó vẫn vậy, chỉ có điều nó sẽ nằm ngang, hay úp mặt vào tường trái, hay dựa lưng vào tường phải
* Ví dụ
* x có Shape là (2, 3, 4)

y = x.unsqueeze(2)

* y có Shape là (2, 3, 1, 4)

1. Cách Tạo Ra Tensor Được Reshape Chia Sẻ Cùng Bộ Dữ Liệu Với Tensor Cũ?

* Tensor sau khi Reshape nếu đọc dữ liệu theo thứ tự sẽ y chang khi đọc như vậy với Tensor cũ

<Tensor Sau Reshape> = <Tensor Cũ>.reshape(<Shape>)

* Ví dụ
* “x” có Shape là (3, 4, 6)

“y” = x.reshape(4, 3, 2, 3)

* “y” có Shape (4, 3, 2, 3)

1. Cách Nhanh Nhất Để Cho Tất Cả Các Giá Trị Trong Tensor Về 0?

<Tensor Muốn Đưa Về Mo>.zero\_()

1. Cách Nhanh Nhất Để Chỉ Copy Dữ Liệu Của 1 Tensor?

<Tensor Chỉ Chứa Dữ Liệu Copy> = tensor(<Tensor Bị Copy Dữ Liệu>)

* <Tensor Chỉ Chứa Dữ Liệu Copy> không có trong sơ đồ tính toán

1. Cách Flip Tensor?

<Flip Tensor> = <Tensor>.flip(<Chiều Flip>)

* <Flip Tensor> = bản Copy của <Tensor> nhưng bị Flip
* <Chiều Flip> có thể là 1 số hoặc 1 Iterable nếu muốn Flip theo nhiều chiều
* Ví dụ

foo = bar.flip([0, 1])

* Ta có

|  |  |
| --- | --- |
| bar | [[0, 1, 2, 3],  [4, 5, 6, 7]] |
| foo | [[7, 6, 5, 4],  [3, 2, 1, 0]] |

1. Cách Hoán Vị Shape Của Tensor?

<Tensor Có Shape Hoán Vị> = <Tensor Gốc>.permute(<Hoán Vị>)

* <Tensor Gốc> và <Tensor Có Shape Hoán Vị> đều dùng chung 1 Tensor trong bộ nhớ
* <Tensor Gốc> không bị thay đổi Shape
* Tương đương việc cầm chặt xấp ảnh lên rồi đặt lại vào góc tường theo kiểu nằm khác, lưu ý có thể bị Flip
* Ví dụ
* x có Shape là (3, 4, 5)

y = x.permute(1, 2, 0)

* y có Shape là (4, 5, 3)

1. Cách Gộp Tổng Giá Trị Vào 1 Trục?

* Giả sử bạn muốn gộp xấp ảnh theo kiểu mỗi tấm ảnh sẽ gộp tổng lại tất cả giá trị của nó thành 1 giá trị duy nhất, nói cách khác, bạn muốn độ lớn trục hoành = trục tung = 1, …
* Nhập đoạn Code sau

<Tensor Ép> = <Tensor Gốc>.sum(<Chiều Không Gian>, keepdim = True)

* <Chiều Không Gian> xác định bạn muốn chiều nào bị ép thành 1
* keepdim = True để <Tensor Ép> không bị rút gọn những chiều không gian có độ lớn = 1
* Ví dụ
* x có Shape là (2, 3, 4)

y = x.sum((0, 2), keepdim = True)

* y có Shape là (1, 3, 1), nghĩa là x gộp vào trục tung

1. Trả Về Index Của Max Trong Tensor?

<Các Index> = <Tensor>.argmax(<Chiều Tính Max>, <Có Giữ Chiều Không>)

* Mặc định

|  |  |
| --- | --- |
| <Có Giữ Chiều Không> | False |

* <Các Index> ban đầu = <Tensor> nhưng mỗi dãy theo <Chiều Tính Max> sẽ được thay thế = <Index> theo chiều đó của phần tử lớn nhất trong dãy, nếu có nhiều phần tử cùng lớn nhất thì lấy phần tử có <Index> nhỏ hơn
* Sau đó, nếu <Có Giữ Chiều Không> = False thì

<Các Index> = <Các Index>.squeeze(<Chiều Tính Max>)

* Ví dụ

foo = bar.argmax(1, True)

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 2, 3, 4, 2, 4],  [4, 2, 9, 8, 7, 6],  [1, 2, 3, 4, 5, 7]] |
| foo | [[3],  [2],  [5]] |

1. Trả Về Max Trong Tensor?

<Các Max>, <Các Index> = <Tensor>.max(<Chiều Tính Max>, <Có Giữ Chiều Không>)

* Mặc định

|  |  |
| --- | --- |
| <Có Giữ Chiều Không> | False |

* Đéo khác gì Arg Max, chỉ có điều là trả về thêm <Các Max>, nó đéo khác gì <Các Index> nhưng mỗi phần tử thay vì là <Index> của Max thì là Max luôn
* Ví dụ

foo, far = bar.max(1, True)

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 2, 3, 4, 2, 4],  [4, 2, 9, 8, 7, 6],  [1, 2, 3, 4, 5, 7]] |
| far | [[3],  [2],  [5]] |
| foo | [[4],  [9],  [7]] |

1. Trả Về Tọa Độ Của Tất Cả Phần Tử Khác 0 Trong 1 Tensor?

<Các Tọa độ> = <Tensor>.nonzero()

* <Các Tọa Độ> là 2D Tensor mà mỗi hàng của nó là tọa độ của 1 phần tử khác 0, đồng thời chúng được sắp xếp theo thứ tự từ nhỏ đến lớn
* Ví dụ

foo = bar.nonzero()

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 0, 2, 0, 0, 3, 4],  [4, 0, 7, 8, 9, 0, 0],  [1, 5, 6, 0, 0, 0, 0] |
| foo | [[0, 0],  [0, 2],  [0, 5],  [0, 6],  [1, 0],  [1, 2],  [1, 3],  [1, 4],  [2, 0],  [2, 1],  [2, 2]] |

1. Singleton Tensor Nghĩa Là Gì?

* Là Tensor bao gồm duy nhất 1 số, không phải mảng
* Nếu bạn lặp qua Tensor thì nó sẽ trả về Singleton Tensor thay vì chỉ trả về số
* Ví dụ
* Đây là Singleton Tensor

tensor(1.2)

* Còn đây không phải Singleton Tensor

tensor([1.2])

1. Singleton Dimension?

* Là chiều có kích thước = 1
* Ví dụ
* “foo” có Shape là (4, 1, 5, 1, 9, 4, 7) thì chiều thứ 1 và thứ 3 là các Singleton Dimension
* Lặp Lại Tensor Nhưng Không Tốn Thêm Không Gian Lưu Trữ?

<Tensor Lặp> = <Tensor>.expand(<Shape Của Tensor Lặp>)

* <Tensor Lặp> sẽ dùng chung dữ liệu với <Tensor>
* <Shape Của Tensor Lặp> phải giống y chang Shape của <Tensor> ở những chỗ không phải Singleton Dimesion
* Ví dụ

bar = foo.expand(8, 5, 4, 2, 3)

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| foo | (1, 5, 4, 1, 3) |
| bar | (8, 5, 4, 2, 3) |

1. Trả Về Giá Trị Từ Tensor Chỉ Chứa 1 Giá Trị, Bao Nhiều Chiều Cũng Được?

<Tensor>.item()

1. Contiguous Tensor Là Gì?

* Là tensor mà dữ liệu của nó được lưu tại 1 vùng liên tục trong bộ nhớ, ngược lại Non Contiguous Tensor sẽ lưu tại những vùng không liên tục
* Non Contiguous được tạo ra khi bạn ghép nhiều Tensor lại với nhau, hoặc thay đổi thứ tự Tensor, Pytorch sẽ không tạo ra Tensor mới mà tham chiếu đến các Tensor mảnh ghép

1. Trả Về Tích Hadamard Của 2 Tensor?

<Tensor 1> \* <Tensor 2>

Quick Codes – Mã Nhanh:

1. Trả Về 1 Cột Trong 2D Tensor Dưới Dạng 1D Tensor?

<2D Tensor>[:, <Số Thứ Tự Cột>]

* Ví dụ

foo = bar[:, 1]

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| bar | (3, 4) |
| foo | (3) |

1. Lưu Parameter Của Module Và Load Nó?

* Đặt 2 hàm sau vào trong Module

def load(self, name = None):

if name:

self.path = f”./{name}.pth”

try:

self.load\_state\_dict(torch.load(self.path))

except:

pass

def save(self):

torch.save(self.state\_dict(), self.path)

* Mỗi lần khởi tạo Module mới mà muốn tải lại Parameter đã lưu, thêm dòng Code

<Module>.load(<Tên File Lưu>)

* Nếu chưa có File lưu thì nó sẽ không tải gì cả
* <Tên File Lưu> không cần phần mở rộng
* Mỗi lần muốn lưu Parameter của Module, thêm dòng Code

<Module>.save()

* Nếu chưa có File lưu sẽ tạo ra File mới, nằm trong thư mục làm việc hiện tại

1. Hàm Trả Về Dilated Tensor Của 1D Tensor?

def dilation\_1d(tensor, dilation):

b = torch.zeros(dilation)

b[0] = 1

dilated\_tensor = tensor.kron(b)

return dilated\_tensor[

…,

:dilated\_tensor.shape[-1] - dilation + 1

]

* Ví dụ

foo = dilation\_1d(bar, 3)

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 2, 3],  [4, 5, 6]] |
| foo | [[1, 0, 0, 2, 0, 0, 3],  [4, 0, 0, 5, 0, 0, 6]] |

1. Hàm Trả Về Dilated Tensor Của 2D Tensor?

def dilation\_2d(tensor, dilation\_shape):

dy, dx = dilation\_shape

b = torch.zeros(dy, dx)

b[0, 0] = 1

dilated\_tensor = tensor.kron(b)

return dilated\_tensor[

…,

:dilated\_tensor.shape[-1] - dy + 1,

:dilated\_tensor.shape[-2] - dx + 1

]

* Ví dụ

foo = dilation\_2d(bar, (2, 3))

* Ta có

|  |  |
| --- | --- |
| Bar | [[1, 2, 3],  [4, 5, 6]] |
| Foo | [[1, 0, 0, 2, 0, 0, 3],  [0, 0, 0, 0, 0, 0, 0],  [4, 0, 0, 5, 0, 0, 6]] |

1. Hàm Mở Rộng 1D Tensor Với Giá Trị 0?

def expand\_1d(tensor, expand):

x = tensor.shape[0]

container = torch.zeros(expand + x)

container[:x] = tensor

return container

* Ví dụ

foo = expand\_1d(bar, 3)

* Ta có

|  |  |
| --- | --- |
| bar | [1, 2] |
| foo | [1, 2, 0, 0, 0] |

1. Hàm Mở Rộng 2D Tensor Với Giá Trị 0?

def expand\_2d(tensor, expand\_shape):

x, y = tensor.shape

container = torch.zeros(

expand\_shape[0] + x,

expand\_shape[1] + y

)

container[:x, :y] = tensor

return container

* Ví dụ

foo = expand\_2d(bar, (2, 3))

* Ta có

|  |  |
| --- | --- |
| bar | [[1, 2],  [4, 5]] |
| foo | [[1, 2, 0, 0, 0],  [4, 5, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0]] |

Neural Network – Mạng Thần Kinh:

1. Cách Import?

from torch import nn

1. Cách Sử Dụng Hàm Loss Nào Đó?

criterion = nn.<Tên Hàm Loss>()

loss = criterion(<Tensor Dự Đoán>, <Tensor Nhãn>)

1. Parameter Khác Gì Tensor?

* Trong Pytorch, Parameter thực chất là Container của Tensor, để có thêm 1 đặc tính là nếu bạn đặt nó vào trong 1 Module thì nó sẽ được tự động thêm vào danh sách các Parameter của Module

1. Cách Tạo 1 Parameter?

<Parameter> = nn.parameter.Parameter(<Tensor>)

* <Parameter> sẽ mặc định có “requires\_grad = True”
* <Tensor> phải có kiểu dữ liệu Float

1. Cách Show Parameter Của 1 Module?

<Parameters> = <Module>.parameters()

* <Parameters> là 1 Generator chứa các phần tử là Parameter của Module
* Nếu muốn có thêm tên của Parameter

<Parameters Kèm Tên> = <Module>.named\_parameters()

* <Parameters Kèm Tên> là Generator chứa các phần tử là Tuple gồm 2 giá trị, 1 là tên Parameter, 2 là Parameter
* Ta cũng có thể truy cập Parameter bằng tên của nó

<Module>.<Tên Parameter>

1. Cách Thay Đổi Dữ Liệu Của 1 Parameter?

<Parameter>.data = <Dữ Liệu mới>

* <Dữ Liệu Mới> là Tensor có kiểu Float, không cần “requires\_grad = True”

1. Cách Tạo 1 Module?

* Class kế thừa “nn.Module” sẽ trở thành 1 Module mới, giả sử tên Class là A
* Hàm “forward” trong A sẽ được mặc định gọi khi viết A(<Tham Số>)
* Khi bạn tạo 1 thuộc tính trong A, mà thuộc tính đó lại là Module, thì mọi Parameter của Module đó sẽ trở thành Parameter của A

1. Cách Để Hiện Cấu Trúc Của 1 Module?

print(<Module>)

1. Module List?

* Tạo 1 Module List

<Module List> = nn.ModuleList(<List Có Các Phần Tử Là Module>)

* <Module List> y chang List thông thường, chỉ khác là nếu nó là giá trị của 1 thuộc tính trong 1 Module A, thì mọi Parameter của các Module trong

<List Có Các Phần Tử Là Module> sẽ trở thành Parameter của A, nếu là List, thì điều này không xảy ra

* Ví dụ

foo = nn.ModuleList([nn.Linear(i, 4) for i in range(2, 5)])

* Ta có

|  |  |
| --- | --- |
| foo[0] | nn.Linear(2, 4) |
| foo[1] | nn.Linear(3, 4) |
| foo[2] | nn.Linear(4, 4) |

1. Sequential?

* Tạo 1 lớp Sequential

<Lớp Sequential> = nn.Sequential(<Các Module>)

* Forward

<Output> = <Lớp Sequential>(<Input>)

* Bản chất trong <Lớp Sequential>, Output của lớp trước làm Input của lớp sau
* Ví dụ

foo = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 5))

bar = foo(bob)

* Ta có

|  |  |
| --- | --- |
|  | Shape |
| bob | (10, 3) |
| bar | (10, 5) |

1. Lớp Tuyến Tính?

* Tạo 1 lớp tuyến tính

<Lớp Tuyến Tính> = nn.Linear(<Kích Thước Input>, <Kích Thước Output>)

* <Lớp Tuyến Tính> là 1 Module cơ bản, chỉ gồm 2 Layer

|  |  |
| --- | --- |
|  | Shape |
| <Weight> | (<Kích Thước Output>, <Kích Thước Input>) |
| <Bias> | (<Kích Thước Output>) |

* Forward

<Output> = <Lớp Tuyến Tính>(<Input>)

* Khi đó

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Input>) | (<Kích Thước Output>) |
| (<Kích Thước Batch>,  <Kích Thước Input>) | (<Kích Thước Batch>,  <Kích Thước Output>) |

* Bản chất là thực hiện phép tính

<Output> = <Input> @ <Weight>T + <Bias>

* Thêm tham số “bias = False” nếu không muốn + <Bias>
* Ví dụ

layer\_1 = nn.Linear(3, 5, bias = False)

1. Tại Sao Phải Có “super().\_\_init\_\_()” Khi Khởi Tạo 1 Class Kế Thừa “nn.Module”?

* Để có thể sử dụng “nn.Linear”, … bên trong Class, khi đó mọi Parameter của mấy cái Module cơ bản này cũng được đưa vào danh sách các Parameter của Class

1. Bản Chất Của “nn.Softmax”?

* Giống y chang “nn.Linear” nhưng không có Parameter, hoạt động như hàm Softmax

<Lớp> = nn.Softmax(dim = <Chiều Tính Softmax>)

output = <Lớp>(<Dữ Liệu Feed>)

1. RNN?

* Tạo 1 RNN Cell

<RNN Cell> = nn.RNN(

<Số Neuron Input>, <Số Neuron Hidden>, <Số Stack>,

batch\_first = True, bidirectional = <Có Train 2 Chiều Không>

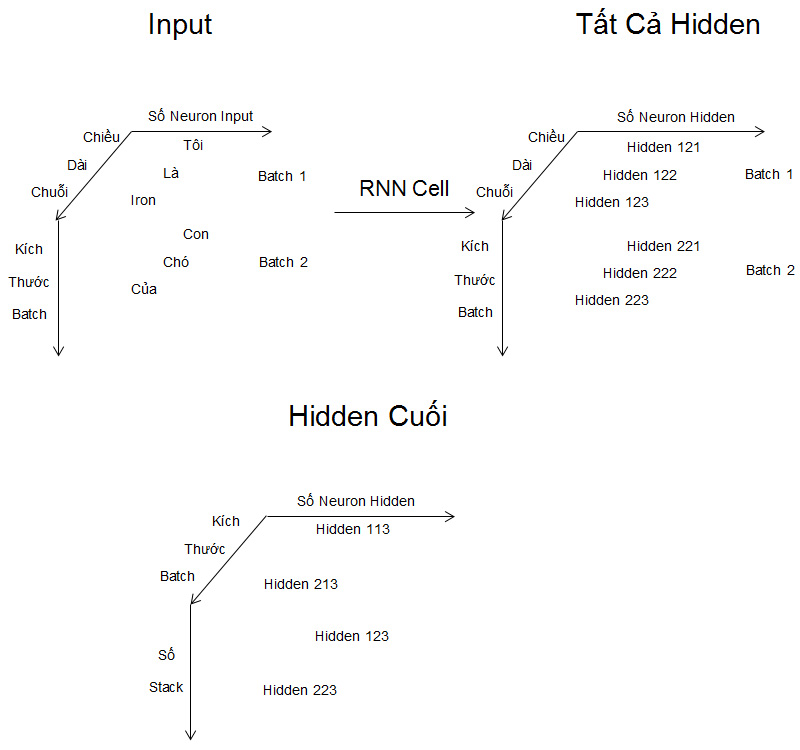
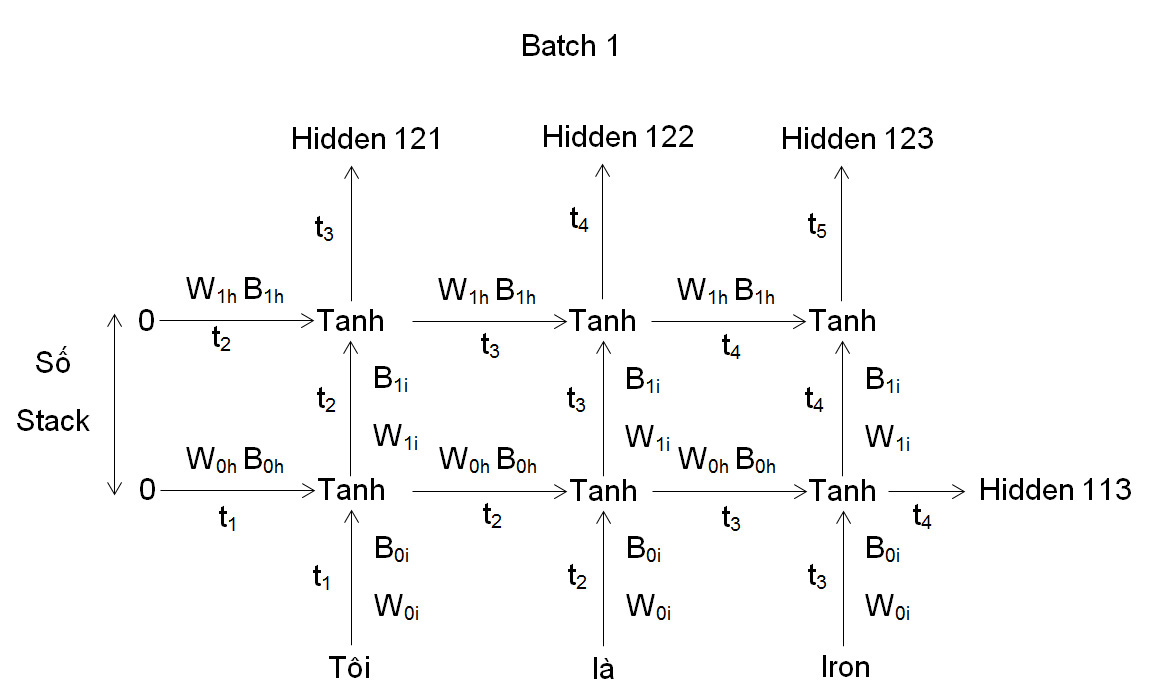
)

* Mặc định

|  |  |
| --- | --- |
| <Số Stack> | 1 |
| <Có Train 2 Chiều Không> | False |

* Nếu <Số Stack> từ 2 trở lên thì gọi là Deep RNN hay Stacked RNN
* Forward

<Tất Cả Hidden>, <Hidden Cuối> = <RNN Cell>(<Input>)

* Minh họa
* 
* 
* W và B là Weight và Bias

|  |  |
| --- | --- |
|  | Shape |
| “Tôi”, “là”, “Iron” | (<Số Neuron Input>) |
| <W0i> | (<Số Neuron Hidden>, <Số Neuron Input>) |
| <W0h>, <W1h>, <W1i> | (<Số Neuron Hidden>, <Số Neuron Hidden>) |
| <Tất Cả Bias> | (<Số Neuron Hidden>) |

* Mỗi Node “Tanh” sẽ cộng 2 mũi tên lại với nhau rồi áp dụng hàm Tanh, dùng hàm Tanh để tránh <Hidden Cuối> sẽ trở lên quá lớn khi <Chiều Dài Chuỗi> lớn
* Thứ tự mũi tên được cho bởi tn
* Mỗi Batch là 1 chuỗi
* Ở đây, <Số Stack> = 2, <Chiều Dài Chuỗi> = 3, <Kích Thước Batch> = 2
* Có được Hidden, ta có thể dùng 1 lớp tuyến tính để chuyển Shape của nó thành (<Số Neuron Input>) rồi ánh xạ nó với từ tương ứng
* Tóm tắt các trường hợp

|  |  |  |
| --- | --- | --- |
| <Input> Shape | <Tất Cả Hidden> Shape | <Hidden Cuối> Shape |
| (<Kích Thước Batch>, <Chiều Dài Chuỗi>, <Số Neuron Input>) | (<Kích Thước Batch>, <Chiều Dài Chuỗi>,  <Số Neuron Hidden>) | (<Số Stack>,  <Kích Thước Batch>,  <Số Neuron Hidden>) |
| (<Chiều Dài Chuỗi>, <Số Neuron Input>) | (<Chiều Dài Chuỗi>,  <Số Neuron Hidden>) | (<Số Stack>,  <Số Neuron Hidden>) |

* Nếu <Có Train 2 Chiều Không> = True, thì sẽ tạo Bidirectional RNN
* Cách hoạt động của 1 Bidirectional RNN như sau
* Ban đầu đặt <Hidden Cuối> = None
* Bản chất bạn tạo ra 2 RNN A và B với Weight và Bias khác hoàn toàn nhau, sau đó bạn Feed <Input> vào Stack đầu tiên của A được <Output 1> và <Hidden Cuối 1>, chèn

<Hidden Cuối 1> xuống dưới đáy <Hidden Cuối>, đồng thời Feed phiên bản Flip theo chiều <Chiều Dài Chuỗi> của <Input> vào Stack đầu tiên của B được <Output 2> và

<Hidden Cuối 2>, chèn <Hidden Cuối 2> xuống dưới đáy <Hidden Cuối>

* Sau đó, phiên bản Flip theo <Chiều Dài Chuỗi> của <Output 2> được Concat vào

<Output 1> được <Input Mới>

* Tiếp theo, <Input Mới> được Feed vào Stack thứ 2 của A, đồng thời phiên bản Flip theo chiều <Chiều Dài Chuỗi> của nó cũng được Feed vào Stack thứ 2 của B, mọi thứ lại diễn ra tương tự
* Tóm tắt các trường hợp của Bidirectional RNN

|  |  |  |
| --- | --- | --- |
| <Input> Shape | <Tất Cả Hidden> Shape | <Hidden Cuối> Shape |
| (<Kích Thước Batch>, <Chiều Dài Chuỗi>, <Số Neuron Input>) | (<Kích Thước Batch>, <Chiều Dài Chuỗi>,  <Số Neuron Hidden> \* 2) | (<Số Stack> \* 2,  <Kích Thước Batch>,  <Số Neuron Hidden>) |
| (<Chiều Dài Chuỗi>, <Số Neuron Input>) | (<Chiều Dài Chuỗi>,  <Số Neuron Hidden> \* 2) | (<Số Stack> \* 2,  <Số Neuron Hidden>) |

1. LSTM?

* Giống y chang RNN, chỉ khác những điểm sau
* Tạo 1 LSTM Cell

<LSTM Cell> = nn.LSTM(

<Số Neuron Input>, <Số Neuron Hidden>, <Số Stack>,

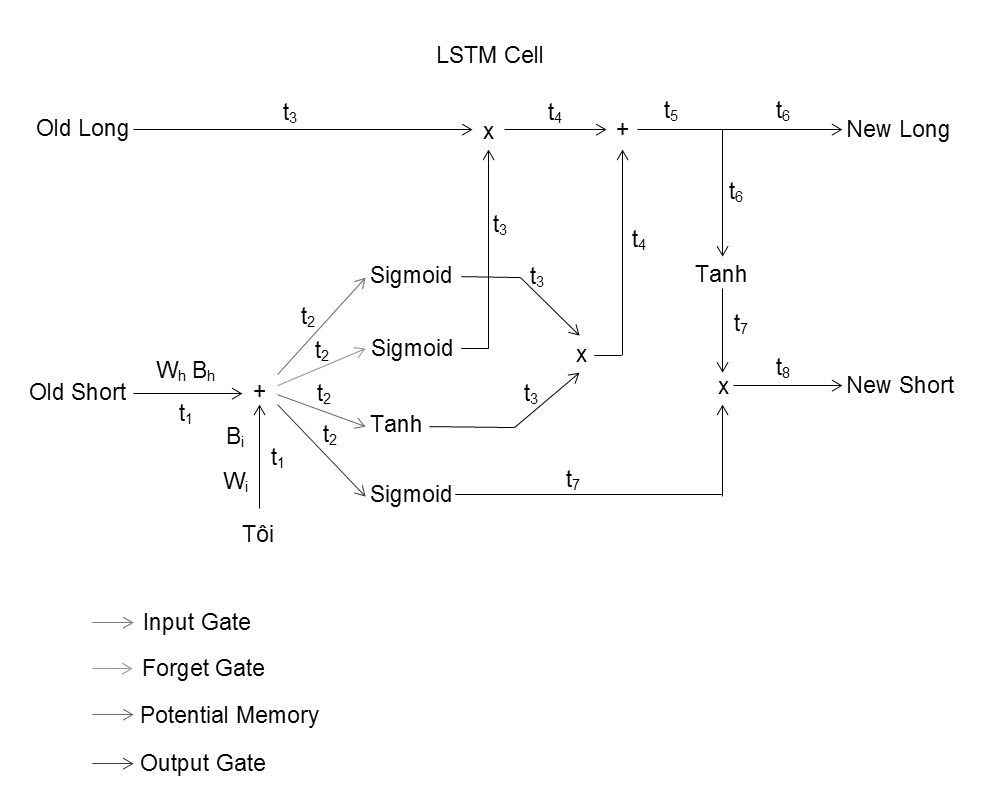
batch\_first = True, bidirectional = <Có Train 2 Chiều Không>

)

* Forward

<Tất Cả Short>, (<Short Cuối>, <Long Cuối>) = <LSTM Cell>(<Input>)

* Minh họa



|  |  |
| --- | --- |
|  | Shape |
| “Tôi” | (<Số Neuron Input>) |
| <Wi> | (<Số Neuron Hidden> \* 4, <Số Neuron Input>) |
| <Wh> | (<Số Neuron Hidden> \* 4, <Số Neuron Hidden>) |
| <Tất Cả Bias> | (<Số Neuron Hidden> \* 4) |

* Mỗi Node “+” sẽ cộng 2 mũi tên lại với nhau
* Mỗi Node “x” sẽ tính tích Hadamard
* <Old Long> và <Old Short> ban đầu = 0
* <Short> và <Long> y chang nhau, chỉ khác <Short> đại diện cho Short Term Memory, <Long> đại diện cho Long Term Memory
* Tại Node “+” đầu tiên, có 4 mũi tên chĩa ra nghĩa Tensor tại Node này được cắt thành 4 phần = nhau, phần 1 là Input Gate, phần 2 là Forget Gate, phần 3 là Potential Memory, phần 4 là Output Gate
* Forget Gate sẽ xác định tỉ lệ <Old Long> được giữ lại, tức là xác định nên quên đi phần nào của Long Term Memory
* Input Gate sẽ xác định tỉ lệ Potential Memory đưa vào Long Term Memory, tức là xác định phần nào đáng nhớ
* Output Gate xác định tỉ lệ Long Term Memory mới sẽ trở thành Short Term Memory mới
* Nếu có từ 2 Stack trở lên, thì <New Short> ở lớp dưới sẽ trở thành <Input> của lớp trên
* LSTM khắc phục tình trạng Gradient bị triệt tiêu hoặc bùng nổ của RNN, vì phần Long Term Memory không có Weight

1. GRU?

* Tạo 1 GRU Cell

<GRU Cell> = nn.GRU(

<Số Neuron Input>, <Số Neuron Hidden>, <Số Stack>,

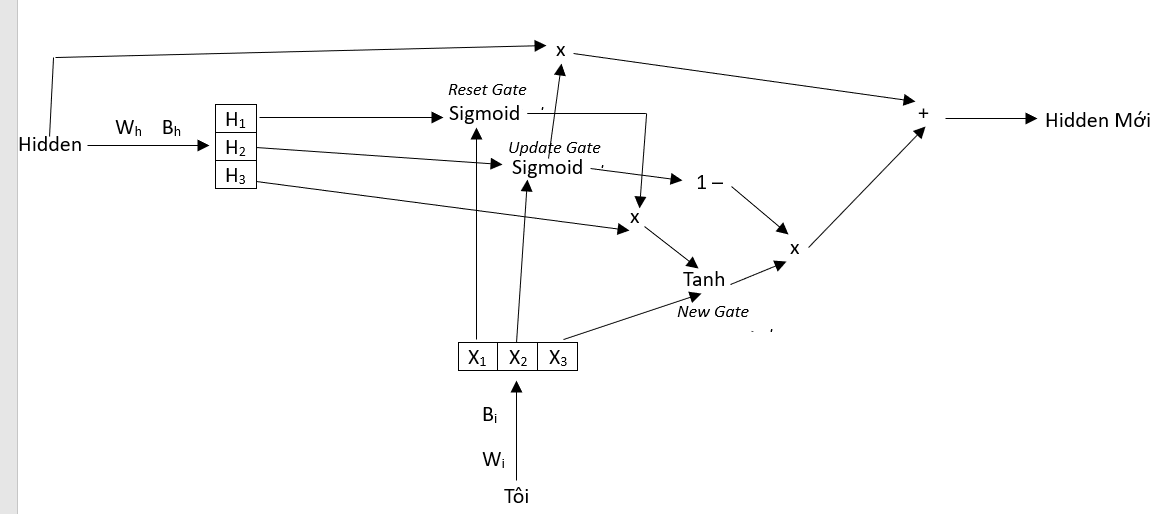
batch\_first = True, bidirectional = <Có Train 2 Chiều Không>

)

* Forward

<Tất Cả Hidden>, <Hidden Cuối> = <GRU Cell>(<Input>)

* Bản chất GRU đéo khác con mẹ gì RNN, nó chỉ khác ở chỗ duy nhất là lúc xử lí Input và Hidden để tạo ra Hidden tiếp theo
* Minh họa



* Gần giống LSTM, 3 khúc X1, X2, X3 được tạo ra bằng cách nhân Input với Weight và Bias khác nhau, nhưng 3 Weight này được Concat lại thành 1 ma trận duy nhất là Wi, 3 Bias cũng Concat thành Bi, tương tự cho H1, H2, H3
* Mỗi Node “Sigmoid” sẽ cộng 2 mũi tên lại với nhau rồi áp dụng hàm Sigmoid
* Node “Tanh” sẽ cộng 2 mũi tên lại với nhau rồi áp dụng hàm Tanh
* Mỗi Node “x” sẽ tính tích Hadamard
* Node “1 –“ sẽ lấy Tensor toàn 1 trừ đi Tensor đầu vào
* Node “+” sẽ tích tổng 2 mũi tên
* Reset Gate sẽ xác định tỉ lệ kí ức cũ ảnh hưởng tới kí ức mới
* Update Gate sẽ xác định tỉ lệ kí ức cũ được giữ lại
* New Gate chứa kí ức mới

|  |  |  |
| --- | --- | --- |
|  | Shape | Tên Parameter |
| “Tôi” | (<Số Neuron Input>) |  |
| Wi | Với Stack dưới cùng  (<Số Neuron Hidden> \* 3,  <Số Neuron Input>)  Với Stack bên trên  (<Số Neuron Hidden> \* 3,  <Số Neuron Hidden>) | weight\_ih\_l<Index Stack>  <Index Stack> = 0, 1, 2, … ứng với Stack dưới cùng, Stack bên trên 1 bậc, Stack bên trên 2 bậc, … |
| Wh | (<Số Neuron Hidden> \* 3,  <Số Neuron Hidden>) | weight\_hh\_l<Index Stack> |
| Bi | (<Số Neuron Hidden> \* 3) | bias\_ih\_l<Index Stack> |
| Bh | (<Số Neuron Hidden> \* 3) | bias\_hh\_l<Index Stack> |

1. 1D CNN?

* Tạo 1 lớp 1D CNN

<Lớp 1D CNN> = nn.Conv1d(

<Số Kênh Input>, <Số Kênh Output>, <Kích Thước Kernel>,

<Stride>, <Padding>, <Dilation>, <Số Nhóm>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Stride> | 1 | <Padding> | 0 |
| <Dilation> | 1 | <Số Nhóm> | 1 |

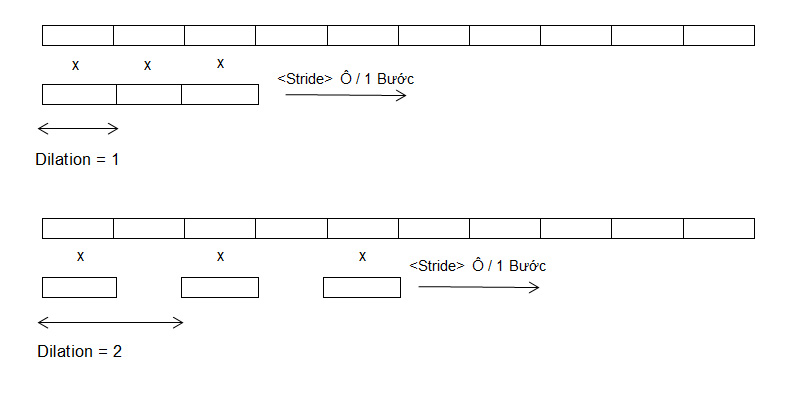
* <Số Kênh Input> và <Số Kênh Output> phải chia hết cho <Số Nhóm>
* Forward

<Output> = <Lớp 1D CNN>(<Input>)

* Minh họa
* <https://drive.google.com/file/d/12hjLNEtjjreO5kMDHzg18tdzl_ih4M3Q/view?usp=sharing>
* Ở minh họa trên

|  |  |  |  |
| --- | --- | --- | --- |
| <Số Kênh Input> | 3 | <Kích Thước Input> | 10 |
| <Số Kênh Output> | 5 | <Kích Thước Output> | 4 |
| <Kích Thước Kernel> | 3 | <Kích Thước Batch> | 10 |
| <Stride> | 2 | <Padding> | 0 |
| <Dilation> | 1 | <Số Nhóm> | 1 |

* 3 ô màu đỏ là ví dụ cho 1 Kernel
* Mỗi kênh Input sẽ có 1 Kernel riêng, sau khi thực hiện Cross Correlation giữa mỗi Kernel với kệnh Input tương ứng, kết quả sẽ được + gộp lại với nhau rồi + Bias tương ứng được 1 kênh Output
* Kernel sẽ dừng trượt nếu lần trượt tiếp theo vượt biên của <Input>
* Đặt <Padding> = k thì Input sẽ được mở rộng sang cả bên trái và bên phải k phần tử, các phần tử này có giá trị = 0
* <Dilation> là độ mở rộng của Kernel
* Minh họa



* Đặt <Số Nhóm> = k nghĩa là bạn chia <Input> thành k phần = nhau theo kênh
* Ví dụ
* <Input> có 15 kênh, <Số Nhóm> = 3, thì mỗi phần sẽ sở hữu 5 kênh <Input>
* Sau đó mỗi phần sẽ hoạt động như 1 lớp 1D CNN độc lập, với <Số Kênh Input Mỗi Phần> = k và <Số Kênh Output Mỗi Phần> = <Số Kênh Output> / k, <Output Mỗi Phần> sau đó sẽ được nối với nhau theo đúng thứ tự để ra <Output>
* Các tham số của <Lớp 1D CNN>

|  |  |
| --- | --- |
|  | Shape |
| <Weight> | (<Số Kênh Output>, <Số Kênh Input> / <Số Nhóm>, <Kích Thước Kernel>) |
| <Bias> | (<Số Kênh Output>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) | (<Kích Thước Batch>,  <Số Kênh Output>,  <Kích Thước Output>) |
| (<Số Kênh Input>,  <Kích Thước Input>) | (<Số Kênh Output>,  <Kích Thước Output>) |

1. 2D CNN?

* Giống y chang 1D CNN, chỉ khác những điểm sau
* Tạo 1 lớp 2D CNN

<Lớp 2D CNN> = nn.Conv2d(

<Số Kênh Input>, <Số Kênh Output>, <Kernel Shape>,

<Stride Shape>, <Padding Shape>, <Dilation Shape>, <Số Nhóm>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Stride Shape> | (1, 1) | <Padding Shape> | (0, 0) |
| <Dilation Shape> | (1, 1) | <Số Nhóm> | 1 |

* Mỗi <Shape> là 1 Tuple gồm 2 phần tử tương ứng theo chiều dọc và chiều ngang
* Nếu các bạn cho <Shape> = giá trị k nào đó thay vì Tuple thì nó tự chuyển về (k, k)
* Forward

<Output> = <Lớp 2D CNN>(<Input>)

* Các tham số của <Lớp 2D CNN>

|  |  |
| --- | --- |
|  | Shape |
| <Weight> | (<Số Kênh Output>,  <Số Kênh Input> / <Số Nhóm>,  <Kích Thước Kernel Theo Chiều Dọc>,  <Kích Thước Kernel Theo Chiều Ngang>) |
| <Bias> | (<Số Kênh Output>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Kích Thước Batch>,  <Số Kênh Output>,  <Kích Thước Output Theo Chiều Dọc>,  <Kích Thước Output Theo Chiều Ngang>) |
| (<Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Số Kênh Output>,  <Kích Thước Output Theo Chiều Dọc>,  <Kích Thước Output Theo Chiều Ngang>) |

1. 1D Transpose CNN?

* Tạo 1 lớp 1D Transpose CNN

<Lớp 1D Transpose CNN> = nn.ConvTranspose1d(

<Số Kênh Input>, <Số Kênh Output>, <Kích Thước Kernel>,

<Stride>, <Padding>, <Padding Thừa>, <Số Nhóm>,

dilation = <Dilation>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Stride> | 1 | <Padding> | 0 |
| <Dilation> | 1 | <Số Nhóm> | 1 |
| <Padding Thừa> | 0 |  |  |

* <Số Kênh Input> và <Số Kênh Output> phải chia hết cho <Số Nhóm>
* Forward

<Output> = <Lớp 1D Transpose CNN>(<Input>)

* Minh họa
* <https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md>
* <Lớp 1D Transpose CNN> bản chất là nghịch đảo của <Lớp 1D CNN>, nó tương đương

nn.Conv1d(

<Số Kênh Input>, <Số Kênh Output>, <Kích Thước Kernel>,

1, <Padding Đã Tính Toán>, <Dilation>, <Số Nhóm>

)

* Với <Input> đã được Dilated với Dilation = <Stride> và thêm <Padding Thừa> vào bên phải, đồng thời <Kernel> bị đảo ngược lại, nói cách khác là thay vì dùng toán tử Cross Correlation thì bây giờ dùng toán tử Convolution

<Padding Đã Tính Toán> = <Dilation> \* (<Kích Thước Kernel> – 1) – <Padding>

* Ví dụ
* Cho

|  |  |  |  |
| --- | --- | --- | --- |
| <Dilation> | 3 | <Padding> | 2 |
| <Kích Thước Kernel> | 4 |  |  |

* Ta có

<Padding Đã Tính Toán> = 3 \* (4 – 1) – 2 = 7

* Ngoài ra, nếu <Padding Thừa> = 0, thì khi đó nếu ta đưa <Output> vào 1 lớp 1D CNN với tham số tương tự, trừ <Số Kênh Input> sẽ hoán đổi với <Số Kênh Output>, ta sẽ được kết quả có Shape = <Input> Shape, nói cách khác, đây chính là lí do vì sao

<Lớp 1D Transpose CNN> và <Lớp 1D CNN> lại là nghịch đảo của nhau

* Các tham số của <Lớp 1D Transpose CNN>

|  |  |
| --- | --- |
|  | Shape |
| <Weight> | (<Số Kênh Input>, <Số Kênh Output> / <Số Nhóm>, <Kích Thước Kernel>) |
| <Bias> | (<Số Kênh Output>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) | (<Kích Thước Batch>,  <Số Kênh Output>,  <Kích Thước Output>) |
| (<Số Kênh Input>,  <Kích Thước Input>) | (<Số Kênh Output>,  <Kích Thước Output>) |

1. 2D Transpose CNN?

* Tạo 1 lớp 2D Transpose CNN

<Lớp 2D Transpose CNN> = nn.ConvTranspose2d(

<Số Kênh Input>, <Số Kênh Output>, <Kernel Shape>,

<Stride Shape>, <Padding Shape>, <Padding Thừa Shape>, <Số Nhóm>,

dilation = <Dilation Shape>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Stride> | (1, 1) | <Padding> | (0, 0) |
| <Dilation> | (1, 1) | <Số Nhóm> | 1 |
| <Padding Thừa> | (0, 0) |  |  |

* <Số Kênh Input> và <Số Kênh Output> phải chia hết cho <Số Nhóm>
* Forward

<Output> = <Lớp 2D Transpose CNN>(<Input>)

* Mối quan hệ của <Lớp 2D Transpose CNN> với <Lớp 2D CNN> đéo khác gì giữa <Lớp 1D Transpose CNN> và <Lớp 2D CNN>
* Các tham số của <Lớp 2D Transpose CNN>

|  |  |
| --- | --- |
|  | Shape |
| <Weight> | (<Số Kênh Input>,  <Số Kênh Output> / <Số Nhóm>,  <Kích Thước Kernel Theo Chiều Dọc>,  <Kích Thước Kernel Theo Chiều Ngang>) |
| <Bias> | (<Số Kênh Output>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Kích Thước Batch>,  <Số Kênh Output>,  <Kích Thước Output Theo Chiều Dọc>,  <Kích Thước Output Theo Chiều Ngang>) |
| (<Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Số Kênh Output>,  <Kích Thước Output Theo Chiều Dọc>,  <Kích Thước Output Theo Chiều Ngang>) |

1. 1D Max Pool?

* Tạo 1 lớp 1D Max Pool

<Lớp 1D Max Pool> = nn. MaxPool1d(

<Kích Thước Kernel>, <Stride>, <Padding>, <Dilation>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Stride> | <Kích Thước Kernel> | <Padding> | 0 |
| <Dilation> | 1 |  |  |

* Forward

<Output> = <Lớp 1D Max Pool>(<Input>)

* <Lớp 1D Max Pool> hoạt động đéo khác gì 1 lớp 1D CNN, chỉ có điều thay vì dùng toán tử Cross Correlation giữa Kernal với Input, thì nó dùng toán tử Max với phần Input nằm trong Kernel
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Output>) |
| (<Số Kênh Input>,  <Kích Thước Input>) | (<Số Kênh Input>,  <Kích Thước Output>) |

1. 2D Max Pool?

* Tạo 1 lớp 2D Max Pool

<Lớp 2D Max Pool> = nn.MaxPool2d(

<Kernel Shape>, <Stride Shape>, <Padding Shape>, <Dilation Shape>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Stride Shape> | <Kernel Shape> | <Padding Shape> | (0, 0) |
| <Dilation Shape> | (1, 1) |  |  |

* Forward

<Output> = <Lớp 2D Max Pool>(<Input>)

* <Lớp 2D Max Pool> hoạt động đéo khác gì 1 lớp 2D CNN, chỉ có điều thay vì dùng toán tử Cross Correlation giữa Kernal với Input, thì nó dùng toán tử Max với phần Input nằm trong Kernel
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Output Theo Chiều Dọc>,  <Kích Thước Output Theo Chiều Ngang>) |
| (<Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Số Kênh Input>,  <Kích Thước Output Theo Chiều Dọc>,  <Kích Thước Output Theo Chiều Ngang>) |

1. 1D Batch Norm?

* Tạo 1 lớp 1D Batch Norm

<Lớp 1D Batch Norm> = nn.BatchNorm1d(

<Số Kênh Input>, <Epsilon>, <Momentum>

)

* Mặc định

|  |  |
| --- | --- |
| <Epsilon> | 1e–5 |
| <Momentum> | 0.1 |

* Forward

<Output> = <Lớp 1D Batch Norm>(<Input>)

* Minh họa
* <https://drive.google.com/file/d/11WXXHIZq2OXuRdzZudolHHrY-zwZEIUs/view?usp=sharing>
* Ở minh họa trên

|  |  |  |  |
| --- | --- | --- | --- |
| <Kích Thước Batch> khi Train | 8 | <Kích Thước Input> | 5 |
| <Kích Thước Batch> khi Test | 4 | <Số Kênh Input> | 3 |

* 5 ô màu đỏ là ví dụ cho 1 kênh của 1 <Input>
* <Input Mean> có các phần tử là Mean của lát cắt tương ứng trong <Input>, cắt theo chiều <Kích Thước Batch> và <Kích Thước Input>
* <Input Variance> cũng tương tự, lưu ý là tính = ước lượng chệch, hay Variance không có hiệu chỉnh, mục đích là để đơn giản hóa quá trình tính toán
* <Weight> có giá trị ban đầu = 1 ở tất cả phần tử
* <Bias> có giá trị ban đầu = 0 ở tất cả phần tử
* Nghĩa là giai đoạn đầu khi Train, phân phối của Input sẽ được chuyển về phân phối chuẩn, trong quá trình Train, thông qua lan truyền ngược, <Weight> và <Bias> sẽ từ từ thay đổi tiến tới trạng thái tối ưu nhất
* <Running Mean> chính là EWMA của <Input Mean>, khi Train, cứ mỗi lần Forward thì <Running Mean> sẽ được cập nhật, ban đầu nó có giá trị = 0 ở tất cả các phần tử và được cập nhật theo công thức sau

<Running Mean Mới> =

(1 – <Momentum>) \* <Running Mean Cũ> + <Momentum> \* <Input Mean Mới>

* Tương tự với <Running Variance>
* Khi Test, thì <Lớp 1D Batch Norm> sẽ tự động chuyển sang sử dụng

<Tham Số Running>

* Dễ thấy, khi Train và Test, <Kích Thước Batch> và <Kích Thước Input> có thể thay đổi tùy ý
* Để chuyển sang chế độ Train

<Lớp 1D Batch Norm>.train()

* Để chuyển sang chế độ Test

<Lớp 1D Batch Norm>.eval()

* Chế độ mặc định là Train
* Để truy cập vào <Tham Số Running>

<Running Mean> = <Lớp 1D Batch Norm>.running\_mean

<Running Variance> = <Lớp 1D Batch Norm>.running\_var

* Các tham số của <Lớp 1D Batch Norm>

|  |  |
| --- | --- |
|  | Shape |
| <Weight>, <Bias>, <Running Mean>, <Running Variance> | (<Số Kênh Input>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) |
| (<Kích Thước Batch>,  <Số Kênh Input>) | (<Kích Thước Batch>,  <Số Kênh Input>) |

1. 2D Batch Norm?

* Giống y chang 1D Batch Norm, Input thay vì 1D thì bây giờ gấp lại thành 2D
* Tạo 1 lớp 2D Batch Norm

<Lớp 1D Batch Norm> = nn.BatchNorm2d(

<Số Kênh Input>, <Epsilon>, <Momentum>

)

* Forward

<Output> = <Lớp 2D Batch Norm>(<Input>)

* Các tham số của <Lớp 2D Batch Norm>

|  |  |
| --- | --- |
|  | Shape |
| <Weight>, <Bias>, <Running Mean>, <Running Variance> | (<Số Kênh Input>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) |

1. Layer Norm?

* Tạo 1 lớp Layer Norm

<Lớp Layer Norm> = nn.LayerNorm(

<Shape Của D Chiều Cuối Cùng>, <Epsilon>

)

* <Epsilon> mặc định = 1e–5
* Forward

<Output> = <Lớp Layer Norm>(<Input>)

* <Input Mean> và <Input Variance> được tính theo <D Chiều Cuối Cùng>
* <Input Variance> sẽ được tính = ước lượng chệch, nghĩa là không hiệu chỉnh, để đơn giản hóa quá trình tính toán
* Công thức chuẩn hóa giống của Batch Norm, có nhân <Weight> và + <Bias>, lưu ý nhân <Weight> là thực hiện tích Hadamard
* <Weight> và <Bias> sẽ được điều chỉnh thông qua lan truyền ngược
* Các tham số của <Lớp Layer Norm>

|  |  |
| --- | --- |
|  | Shape |
| <Weight>, <Bias> | (<Shape Của D Chiều Cuối Cùng>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (…, <Shape Của D Chiều Cuối Cùng>) | (…, <Shape Của D Chiều Cuối Cùng>) |

* Ví dụ

|  |  |
| --- | --- |
| <Shape Của D Chiều Cuối Cùng> | (4, 5, 6) |
| <Input> Shape | (7, 8, 4, 5, 6) |
| <Output> Shape | (7, 8, 4, 5, 6) |
| <Weight> | (4, 5, 6) |
| <Bias> | (4, 5, 6) |

1. 1D Instance Norm?

* Chả khác đéo gì 1D Batch Norm
* Tạo 1 lớp 1D Instance Norm

<Lớp 1D Instance Norm> = nn.InstanceNorm1d(

<Số Kênh Input>, <Epsilon>, <Momentum>,

affine = True, track\_running\_stats = True

)

* Forward

<Output> = <Lớp 1D Instance Norm>(<Input>)

* Sự khác biệt có lẽ là duy nhất đó là, trong khi Train, thay vì chuẩn hóa nguyên 1 Batch của 1 kênh <Input> theo Mean và Variance của Batch đó như trong 1D Batch Norm, thì 1D Instance Norm sẽ chuẩn hóa từng kênh <Input> trong Batch đó theo Mean và Variance của riêng chúng
* Lưu ý <Kích Thước Input> phải > 1, bởi vì nếu = 1 thì làm đéo gì tính được Variance
* Các tham số của <Lớp 1D Instance Norm>

|  |  |
| --- | --- |
|  | Shape |
| <Weight>, <Bias>, <Running Mean>, <Running Variance> | (<Số Kênh Input>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input>) |
| (<Số Kênh Input>,  <Kích Thước Input>) | (<Số Kênh Input>,  <Kích Thước Input>) |

1. 2D Instance Norm?

* Giống y chang 1D Instance Norm, Input thay vì 1D thì bây giờ gấp lại thành 2D
* Tạo 1 lớp 2D Instance Norm

<Lớp 2D Instance Norm> = nn.InstanceNorm2d(

<Số Kênh Input>, <Epsilon>, <Momentum>,

affine = True, track\_running\_stats = True

)

* Forward

<Output> = <Lớp 2D Instance Norm>(<Input>)

* Các tham số của <Lớp 2D Instance Norm>

|  |  |
| --- | --- |
|  | Shape |
| <Weight>, <Bias>, <Running Mean>, <Running Variance> | (<Số Kênh Input>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) |
| (<Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) | (<Số Kênh Input>,  <Kích Thước Input Theo Chiều Dọc>,  <Kích Thước Input Theo Chiều Ngang>) |

1. Group Norm?

* Cách hoạt động gần giống Layer Norm
* Tạo 1 lớp Group Norm

<Lớp Group Norm> = nn.GroupNorm(

<Số Nhóm>, <Số Kênh Input>, <Epsilon>

)

* <Epsilon> mặc định = 1e–5
* <Số Kênh Input> phải chia hết cho <Số Nhóm>
* Forward

<Output> = <Lớp Group Norm>(<Input>)

* Giả sử có 1 Batch, mỗi phần tử trong Batch có nhiều kênh, giả sử ta có phần tử A, chia A thành các nhóm có Shape = nhau, ví dụ, A có Shape là (6, 4, 5), 6 ở đây là <Số Kênh Input>, thì khi chia A thành 3 phần, 3 ở đây là <Số Nhóm>, 3 phần lần lượt là A[0 : 2, : , : ], A[2 : 4, : , : ], A[4 : 6, : , : ], chuẩn hóa mỗi phần dựa theo Mean và Variance của riêng chúng, lưu ý Variance ở đây là ước lượng chệch, nghĩa là không có hiệu chỉnh, mục đích đơn giản hóa việc tính toán
* Kết quả sau đó sẽ đem nhân <Weight> và + <Bias> giống như Batch Norm
* <Weight> và <Bias> sẽ được điều chỉnh thông qua lan truyền ngược
* Các tham số của <Lớp Group Norm>

|  |  |
| --- | --- |
|  | Shape |
| <Weight>, <Bias> | (<Số Kênh Input>) |

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Output> Shape |
| (<Kích Thước Batch>,  <Số Kênh Input>,  …) | (<Kích Thước Batch>,  <Số Kênh Input>,  …) |

1. Drop Out?

* Tạo 1 lớp Drop Out

<Lớp Drop Out> = nn.Dropout(<Tỉ Lệ Drop Out>)

* <Tỉ Lệ Drop Out> mặc định = 0.5
* Forward

<Output> = <Lớp Drop Out>(<Input>)

* Khi Train, <Output> sẽ = k lần tích Hadamard của <Input> với 1 ma trận, ma trận này có Shape của <Input>, chỉ chứa giá trị 0 và 1 một cách ngẫu nhiên, và thay đổi qua mỗi lần Forward, xác suất của 0 = <Tỉ Lệ Drop Out>
* Công thức của k

k = 1 / (1 – <Tỉ Lệ Drop Out>)

* Mục đích phải Scale lên k lần là để tổng giá trị của <Input> không bị thay đổi
* <Lớp Drop Out> có tác dụng tắt ngẫu nhiên các Neuron, do khi nhân với 0 thì sẽ không có Gradient, và do đó, Neuron không học, tránh quá khớp

1. Multi Head Attention?

* Tạo 1 lớp Multi Head Attention

<Lớp Multi Head Attention> = nn.MultiheadAttention(

<Kích Thước Q>, <Số Đầu>,

kdim = <Kích Thước K>, vdim = <Kích Thước V>,

batch\_first = True

)

* <Kích Thước K> và <Kích Thước V> mặc định = <Kích Thước Q>
* <Kích Thước Q> phải chia hết cho <Số Đầu>
* Forward

<Output>, <Attention Weight Trung Bình> = <Lớp Multi Head Attention>(<Q>, <K>, <V>)

* <Chiều Dài Chuỗi K> phải = <Chiều Dài Chuỗi V>
* Minh họa
* <https://drive.google.com/file/d/1BMVJZPEosn5biyoN2jNkfvq08epig9Rs/view?usp=sharing>
* Ở minh họa trên

|  |  |  |  |
| --- | --- | --- | --- |
| <Kích Thước Q> | 4 | <Chiều Dài Chuỗi Q> | 2 |
| <Kích Thước K> | 6 | <Chiều Dài Chuỗi K>, <Chiều Dài Chuỗi V> | 5 |
| <Kích Thước V> | 5 | <Kích Thước Batch> | 3 |

* 4 ô màu đỏ ở <Q> là ví dụ cho Query của 1 Token nào đó, 6 ô màu đỏ ở <K> và 5 ô màu đỏ ở <V> là ví dụ cho Key và Value của 1 Token khác
* Tưởng tượng <Q> là “Tèo” và “Tí”, <K> là “An”, “Bình”, “Đức”, “V” là “Yếu”, “Mạnh”, “Quái Vật”, ta cầm từng phần tử trong Q đi hỏi từng phần tử trong K xem mày có ảnh hưởng nhiều tới tao không, sau đó dùng mức độ ảnh hưởng đó đối chiếu với “V”, ví dụ “Tèo” bị “An” ảnh hưởng nhiều, nhưng bị ảnh hưởng ít bởi “Bình” và “Đức”, suy ra “Tèo” “Yếu”, “Tí” bị ảnh hưởng nhiều bởi “Đức” và “Bình”, suy ra “Tí” “Mạnh Vãi Lồn”
* Mức độ ảnh hưởng đo được = tích vô hướng giữa Query với từng Key, sau khi quy đổi ra trọng số thì giá trị Query nhận được = tổng Value \* trọng số tương ứng
* Các tham số của <Lớp Multi Head Attention>
* Nếu <Kích Thước K> hoặc <Kích Thước V> ≠ <Kích Thước Q>

|  |  |
| --- | --- |
|  | Shape |
| <Q Weight> | (<Kích Thước Q>, <Kích Thước Q>) |
| <K Weight> | (<Kích Thước Q>, <Kích Thước K>) |
| <V Weight> | (<Kích Thước Q>, <Kích Thước V>) |
| <In Bias> | (<Kích Thước Q> \* 3) |
| <Out Weight> | (<Kích Thước Q>, <Kích Thước Q>) |
| <Out Bias> | (<Kích Thước Q>) |

* Nếu ngược lại

|  |  |
| --- | --- |
|  | Shape |
| <In Weight> | (<Kích Thước Q> \* 3, <Kích Thước Q>) |
| <In Bias> | (<Kích Thước Q> \* 3) |
| <Out Weight> | (<Kích Thước Q>, < Kích Thước Q>) |
| <Out Bias> | (<Kích Thước Q>) |

* <In Weight> là 3 thằng <Q Weight>, <K Weight>, <V Weight> nối với nhau theo đúng thứ tự
* Tóm tắt các trường hợp

|  |  |  |  |
| --- | --- | --- | --- |
| <Q> Shape,  <Output> Shape | <K> Shape | <V> Shape | <Attention Weight Trung Bình> Shape |
| (<Kích Thước Batch>,  <Chiều Dài Chuỗi Q>, <Kích Thước Q>) | (<Kích Thước Batch>,  <Chiều Dài Chuỗi K>, <Kích Thước K>) | (<Kích Thước Batch>,  <Chiều Dài Chuỗi K>, <Kích Thước V>) | (<Kích Thước Batch>, <Chiều Dài Chuỗi Q>, <Chiều Dài Chuỗi K>) |
| (<Chiều Dài Chuỗi Q>, <Kích Thước Q>) | (<Chiều Dài Chuỗi K>, <Kích Thước K>) | (<Chiều Dài Chuỗi K>, <Kích Thước V>) | (<Chiều Dài Chuỗi Q>, <Chiều Dài Chuỗi K>) |

* Nếu muốn Masking, thì khi Forward

<Output>, <Attention Weight Trung Bình> = <Lớp Multi Head Attention>(

<Q>, <K>, <V>,

<Key Padding Mask>, attn\_mask = <Attention Mask>

)

* Ngay trước khi hàm Softmax thực thi, thì <Key Padding Mask> và <Attention Mask> sẽ được + vào các đầu
* Nếu <Mask> có kiểu Boolean thì True sẽ = –∞ và False = 0
* Cách + <Attention Mask>

|  |  |
| --- | --- |
| <Attention Mask> Shape | Cách + |
| (<Chiều Dài Chuỗi Q>,  <Chiều Dài Chuỗi K>) | <Mask> sẽ được Broadcast rồi + vào tất cả các đầu của tất cả phần tử trong Batch |
| (<Kích Thước Batch> \* <Số Đầu>, <Chiều Dài Chuỗi Q>,  <Chiều Dài Chuỗi K>) | <Mask> sẽ được chia ra các khúc liên tiếp có Shape = nhau, ma trận thứ k của khúc thứ n sẽ được + vào đầu thứ k của phần tử thứ n trong Batch |

* Cách + <Key Padding Mask>

|  |  |
| --- | --- |
| <Key Padding Mask> Shape | Cách + |
| (<Chiều Dài Chuỗi K>) | <Key Padding Mask> sẽ được Broadcast rồi + vào tất cả các đầu  Dùng khi không có Batch |
| (<Kích Thước Batch>,  <Chiều Dài Chuỗi K>) | <Key Padding Mask> sẽ được Reshape thành  (<Kích Thước Batch>, 1, <Chiều Dài Chuỗi K>),  Broadcast rồi + vào tất cả các đầu của tất cả phần tử trong Batch  Dùng khi có Batch |

1. Embedding?

* Đéo khác gì lớp tuyến tính chỉ có điều không có Bias
* Tạo 1 lớp Embedding

<Lớp Embedding> = nn.Embedding(<Kích Thước Input>, <Kích Thước Output>)

* Forward

<Output> = <Lớp Embedding>(<Index Tensor>)

* Mỗi phần tử trong <Index Tensor> sẽ được đổi về One Hot Vector với kích thước là

<Kích Thước Input>, vị trí Hot là giá trị của phần tử

* Điểm khác biệt nữa là Weight> của <Lớp Embedding> bị chuyển vị so với lớp tuyến tính thông thường
* Các tham số của <Lớp Embedding>

|  |  |
| --- | --- |
|  | Shape |
| <Weight> | (<Kích Thước Input>, <Kích Thước Output>) |

* Tóm tắt các trường hợp

|  |
| --- |
| <Output> Shape |
| (<Index Tensor Shape>, <Kích Thước Output>) |

1. Flatten?

* Tạo 1 lớp Flatten

<Lớp Flatten> = nn.Flatten(<Chiều Bắt Đầu>, <Chiều Kết Thúc>)

* Mặc định

|  |  |
| --- | --- |
| <Chiều Bắt Đầu> | 1 |
| <Chiều Kết Thúc> | –1 |

* Forward

<Output> = <Lớp Flatten>(<Input>)

* Ví dụ

|  |  |
| --- | --- |
| <Input> Shape | (2, 3, 4, 5, 6, 7) |
| <Chiều Bắt Đầu> | 2 |
| <Chiều Kết Thúc> | 4 |
| <Output> Shape | (2, 3, 120, 7) |

* Lưu ý bảo toàn thứ tự đọc giá trị

1. Transformer?

* Tạo 1 lớp Transformer

<Lớp Transformer> = nn.Transformer(

<Kích Thước Embedding>, <Số Đầu>, <Số Stack Encoder>, <Số Stack Decoder>,

<Kích Thước Feed Forward>, <Tỉ Lệ Drop Out>, batch\_first = True

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Kích Thước Embedding> | 512 | <Số Stack Decoder> | 6 |
| <Số Đầu> | 8 | <Kích Thước Feed Forward> | 2048 |
| <Số Stack Encoder> | 6 | <Tỉ Lệ Drop Out> | 0.1 |

* Forward

<Output> = <Lớp Transformer>(

<Input>, <Target>,

<Attention Mask 1>, <Attention Mask 2>, <Attention Mask 3>,

<Key Padding Mask 1>, <Key Padding Mask 2>, <Key Padding Mask 3>

)

* Mặc định

|  |  |
| --- | --- |
| Tất cả <Mask> | 0 |

* Quá trình xử lí <Input> và <Target> trong <Lớp Transformer>
* Ban đầu ở Encoder, <Input> sẽ được đưa vào lớp Multi Head Attention A1, nó sẽ đảm nhiệm vai trò của cả Query, Key, và Value, do đó lớp này còn gọi là Self Attention

A1 = nn.Multi HeadAttention(<Kích Thước Embedding>, <Số Đầu>, batch\_first = True)

<A1 Output> = A1(

<Input>, <Input>, <Input>,

<Key Padding Mask 1>, attn\_mask = <Attention Mask 1>

)[0]

* Sau đó <A1 Output> bị Drop Out, rồi kết hợp với <Input> bằng 1 kết nối Residual và cho đi qua lớp Layer Norm N1

N1 = nn.LayerNorm(<Kích Thước Embedding>)

<N1 Output> = N1(<Input> + <A1 Output>)

* Tiếp theo <N1 Output> sẽ đi qua lớp tuyến tính F1 với hàm kích hoạt ReLU, bị Drop Out rồi qua lớp tuyến tính F2, bị Drop Out lần nữa, sau đó kết hợp <N1 Output> ban đầu bằng kết nối Residual và cuối cùng cho đi qua lớp Layer Norm N2

F1 = nn.Linear(<Kích Thước Embedding>, <Kích Thước Feed Forward>)

F2 = nn.Linear(<Kích Thước Feed Forward>, Kích Thước Embedding>)

N2 = nn.LayerNorm(<Kích Thước Embedding>)

<N2 Output> = N2(<N1 Output> + F2(ReLU(F1(<N1 Output>))))

* Quá trình trên từ lúc vào A1 tới lúc ra N2 sẽ được lặp đi lặp lại với số lần =

<Số Stack Encoder> nhưng với tham số khác nhau, Output của lần này sẽ là Input của lần sau, kết quả cuối cùng là <Stack Encoder Output> sẽ đi qua lớp Layer Norm N3

N3 = nn.LayerNorm(<Kích Thước Embedding>)

<Encoder Output> = N3(<Stack Encoder Output>)

* Chuyển qua Decoder, ban đầu <Target> sẽ được đưa vào lớp Multi Head Attention A2, nó sẽ đảm nhiệm vai trò của cả Query, Key, và Value, hay gọi là Self Attention

A2 = nn.Multi HeadAttention(<Kích Thước Embedding>, <Số Đầu>, batch\_first = True)

<A2 Output> = A2(

<Target>, <Target>, <Target>,

<Key Padding Mask 2>, attn\_mask = <Attention Mask 2>

)[0]

* Sau đó <A2 Output> bị Drop Out, rồi kết hợp với <Target> bằng kết nối Residual và cho đi qua lớp Layer Norm N4

N4 = nn.LayerNorm(<Kích Thước Embedding>)

<N4 Output> = N4(<Target> + <A2 Output>)

* Tiếp theo <N4 Output> sẽ trở thành Query, còn <Encoder Output> sẽ trở thành Key và Value cho lớp Multi Head Attention A3

A3 = nn.Multi HeadAttention(<Kích Thước Embedding>, <Số Đầu>, batch\_first = True)

<A3 Output> = A3(

<N4 Output>, <Encoder Output>, <Encoder Output>,

<Key Padding Mask 3>, attn\_mask = <Attention Mask 3>

)[0]

* Sau đó <A3 Output> bị Drop Out, rồi kết hợp với <N4 Output> bằng kết nối Residual và cho đi qua lớp Layer Norm N5

N5 = nn.LayerNorm(<Kích Thước Embedding>)

<N5 Output> = N5(<N4 Output> + <A3 Output>)

* Tiếp theo <N5 Output> sẽ đi qua lớp tuyến tính F3 với hàm kích hoạt ReLU, bị Drop Out rồi qua lớp tuyến tính F4, bị Drop Out lần nữa, sau đó kết hợp <N5 Output> ban đầu bằng kết nối Residual và cuối cùng cho đi qua lớp Layer Norm N6

F3 = nn.Linear(<Kích Thước Embedding>, <Kích Thước Feed Forward>)

F4 = nn.Linear(<Kích Thước Feed Forward>, Kích Thước Embedding>)

N6 = nn.LayerNorm(<Kích Thước Embedding>)

<N6 Output> = N6(<N5 Output> + F4(ReLU(F3(<N5 Output>))))

* Quá trình trên từ lúc vào A2 tới lúc ra N6 sẽ được lặp đi lặp lại với số lần =

<Số Stack Decoder> nhưng với tham số khác nhau, Output của lần này sẽ là Input của lần sau, kết quả cuối cùng là <Stack Decoder Output> sẽ đi qua lớp Layer Norm N7

N7 = nn.LayerNorm(<Kích Thước Embedding>)

<Output> = N7(<Stack Decoder Output>)

* Mục tiêu Train là <Output> phải = <Target> dịch sang trái 1 từ
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Input> Shape | <Target> Shape, <Output> Shape |
| (<Kích Thước Batch>,  <Chiều Dài Chuỗi Input>,  <Kích Thước Embedding>) | (<Kích Thước Batch>,  <Chiều Dài Chuỗi Target>,  <Kích Thước Embedding>) |
| (<Chiều Dài Chuỗi Input>,  <Kích Thước Embedding>) | (<Chiều Dài Chuỗi Target>,  <Kích Thước Embedding>) |

* Có 1 số lưu ý sau
* <Attention Mask 2> phải là ma trận sao cho các phần tử phía trên đường chéo chính = –∞, các phần tử còn lại = 0, vì khi tính Softmax, các phần tử –∞ sẽ trở thành 0, như vậy tất cả từ chỉ có thể chú ý đến những từ đằng trước nó, còn đằng sau khi mặc kệ, đó là do khi Test, ta tạo từng từ một chứ không biết hết câu, hệ quả là khi Test, ta cũng phải sử dụng

<Attention Mask 2>

* Cách tạo < Attention Mask 2> nhanh nhất

<Attention Mask 2> = <Lớp Transformer>. generate\_square\_subsequent\_mask(

<Chiều Dài Chuỗi Target>

)

* Vì độ dài mỗi câu dùng để Train là không giống nhau, nên khi tạo Batch, ta phải Padding cho chúng những từ bất kì để các câu có độ dài = nhau, để tránh những nội dung dư thừa ảnh hưởng đến kết quả, dùng <Key Padding Mask> chứa các phần tử có giá trị là 0 và –∞, vị trí các phần tử có giá trị –∞ phải là vị trí của các từ dư thừa

1. Cross Entropy Loss?

* Tạo 1 lớp Cross Entropy Loss

<Lớp Cross Entropy Loss> = nn.CrossEntropyLoss(

weight = <Weight>, ignore\_index = <Index Bỏ Qua>, reduction = <Cách Tính Loss>,

label\_smoothing = <Độ Mượt>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Weight> | None | <Index Bỏ Qua> | –100 |
| <Cách Tính Loss> | "mean" | <Độ Mượt> | 0 |

* Forward

<Output> = <Lớp Cross Entropy Loss>(<Input>, <Target>)

* Ta có
* Trường hợp 1

|  |  |
| --- | --- |
| <Input> Shape | (<Số Kênh Input>) |
| <Target> Shape | (<Số Kênh Input>) |
| <Weight> Shape, nếu có | (<Số Kênh Input>) |

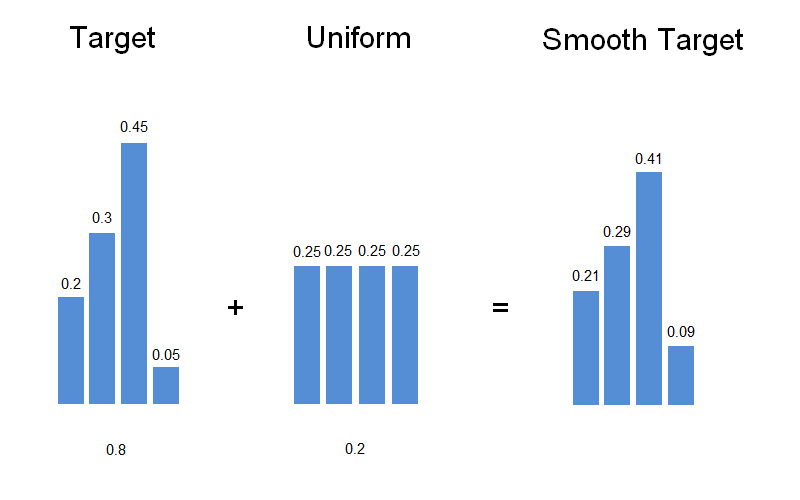
* Ban đầu, <Input> sẽ đi qua hàm Softmax

<Probability> = Softmax(<Input>)

* Tiếp theo, <Target> sẽ được làm mượt, điều này để tránh việc quá khớp, bằng cách Mix nó với 1 phân phối đều liên tục

<Smooth Target> = (1 – <Độ Mượt>) \* <Target> + <Độ Mượt> / <Số Kênh Input>

* Minh họa



* Cuối cùng, tính tổng Cross Entropy có trọng số giữa <Probability> và <Smooth Target>, nếu <Weight> = None, thì trọng số tự động = 1

<Output> = Sum(–<Smooth Target> \* Log(<Probability>) \* <Weight>)

* Ta có

|  |  |
| --- | --- |
| <Output> Shape | () |

* Trường hợp 2

|  |  |
| --- | --- |
| <Input> Shape | (<Kích Thước Batch>, <Số Kênh Input>) |
| <Target> Shape | (<Kích Thước Batch>, <Số Kênh Input>) |
| <Weight> Shape, nếu có | (<Số Kênh Input>) |

* Mỗi phần tử trong Batch sẽ được tính toán y chang như trường hợp 1, thu được

<Các Output>, nếu <Cách Tính Loss> = "none", thì trả về <Các Output> luôn, nếu = "sum" thì trả về Sum(<Các Output>), nếu = "mean" thì trả về Mean(<Các Output>)

* Ta có

|  |  |
| --- | --- |
| <Cách Tính Loss> | <Output> Shape |
| "none" | (<Kích Thước Batch>) |
| "sum" | () |
| "mean" | () |

* Trường hợp 3

|  |  |
| --- | --- |
| <Input> Shape | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước 1>,  <Kích Thước 2>,  …) |
| <Target> Shape | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước 1>,  <Kích Thước 2>,  …) |
| <Weight> Shape, nếu có | (<Số Kênh Input>) |

* Mỗi phần tử có Shape là (1, <Số Kênh Input>, 1, 1, …) trong <Input> và phần tử tương ứng trong <Target> sẽ tính toán với nhau y chang trường hợp 1, thu được <Các Output> và xử lí chúng y chang trường hợp 2
* Ta có

|  |  |
| --- | --- |
| <Cách Tính Loss> | <Output> Shape |
| "none" | (<Kích Thước Batch>,  <Kích Thước 1>,  <Kích Thước 2>,  …) |
| "sum" | () |
| "mean" | () |

* Trường hợp 4

|  |  |
| --- | --- |
| <Input> Shape | (<Số Kênh Input>) |
| <Target> Shape | () |
| <Weight> Shape, nếu có | (<Số Kênh Input>) |

* Ban đầu, <Target> sẽ trở thành One Hot Vector với Hot Index = giá trị của nó, số phần tử của Vector này = <Số Kênh Input>

<One Hot Target> = OneHot(<Target>)

* Mọi thứ tiếp diễn như trường hợp 1, tuy nhiên, nếu <Index Bỏ Qua> = <Target> thì

<Output> = 0

* Cuối cùng nếu <Cách Tính Loss> = "mean" thì

<Output> = <Output> / <Weight>[<Target>]

* Trường hợp 5

|  |  |
| --- | --- |
| <Input> Shape | (<Kích Thước Batch>, <Số Kênh Input>) |
| <Target> Shape | (<Kích Thước Batch>) |
| <Weight> Shape, nếu có | (<Số Kênh Input>) |

* Mỗi phần tử trong Batch sẽ được tính toán y chang như trường hợp 4 và 2, khúc cuối có thay đổi 1 chút, nếu <Cách Tính Loss> = "mean" thì

<Output> = <Output> / Mean(<Weight>[<Target>])

* Lưu ý những phần tử có <Index Bỏ Qua> sẽ không được tính vào Mean
* Trường hợp 6

|  |  |
| --- | --- |
| <Input> Shape | (<Kích Thước Batch>,  <Số Kênh Input>,  <Kích Thước 1>,  <Kích Thước 2>,  …) |
| <Target> Shape | (<Kích Thước Batch>,  <Kích Thước 1>,  <Kích Thước 2>,  …) |
| <Weight> Shape, nếu có | (<Số Kênh Input>) |

* Mỗi phần tử có Shape là (1, <Số Kênh Input>, 1, 1, …) trong <Input> và phần tử tương ứng trong <Target> sẽ tính toán với nhau y chang trường hợp 4, 3 và 5

1. Binary Cross Entropy Loss?

* Tạo 1 lớp Binary Cross Entropy Loss

<Lớp Binary Cross Entropy Loss> = nn.BCELoss(

weight = <Weight>, reduction = <Cách Tính Loss>

)

* Mặc định

|  |  |
| --- | --- |
| <Weight> | None |
| <Cách Tính Loss> | "mean" |

* Forward

<Output> = <Lớp Binary Cross Entropy Loss>(<Input>, <Target>)

* Các phần tử của <Input> và <Target> phải có giá trị từ 0 đến 1
* Ban đầu, tính Binary Cross Entropy giữa <Input> và <Target>

<Binary Cross Entropy> = –(<Target> \* Log(<Input>) + (1 – <Target>) \* Log(1 – <Input>))

* Tiếp theo nhân <Weight> nếu có, <Weight> phải Broadcast được với

<Binary Cross Entropy>

<Output> = <Weight> \* <Binary Cross Entropy>

* Nếu <Cách Tích Loss> = "sum"

<Output> = Sum(<Output>)

* Nếu <Cách Tích Loss> = "mean"

<Output> = Mean(<Output>)

* Ta có

|  |  |
| --- | --- |
| <Cách Tính Loss> | <Output> Shape |
| "none" | (<Input Shape>) |
| "sum" | () |
| "mean" | () |

1. Sigmoid Binary Cross Entropy Loss?

* Tạo 1 lớp Sigmoid Binary Cross Entropy Loss

<Lớp Sigmoid Binary Cross Entropy Loss> = nn.BCEWithLogitsLoss(

weight = <Weight>, reduction = <Cách Tính Loss>, pos\_weight = <Trọng Số Của 1>

)

* Mặc định

|  |  |
| --- | --- |
| <Weight> | None |
| <Trọng Số Của 1> | None |
| <Cách Tính Loss> | "mean" |

* Forward

<Output> = <Lớp Sigmoid Binary Cross Entropy Loss>(<Input>, <Target>)

* Các phần tử của <Input> và <Target> phải có giá trị từ 0 đến 1
* Ban đầu, <Input> đi qua hàm Sigmoid

<Probability> = Sigmoid(<Input>)

* Tiếp theo, tính Binary Cross Entropy giữa <Probability> và <Target>, tính thêm

<Trọng Số Của 1> nếu có, <Trọng Số Của 1> phải Broadcast được với <Input>

<Binary Cross Entropy> =

–(<Trọng Số Của 1> \* <Target> \* Log(<Input>) + (1 – <Target>) \* Log(1 – <Input>))

* Phần còn lại đéo khác gì Binary Cross Entropy

1. Mean Squared Error Loss?

* Tạo 1 lớp Mean Squared Error Loss

<Lớp Mean Squared Error Loss> = nn.MSELoss(reduction = <Cách Tính Loss>)

* Mặc định

|  |  |
| --- | --- |
| <Cách Tính Loss> | "mean" |

* Forward

<Output> = <Lớp Mean Squared Error Loss>(<Input>, <Target>)

* Ban đầu, tính Mean Squared Error giữa <Input> và <Target>

<Output> = (<Input> – <Target>) \*\* 2

* Phần còn lại đéo khác gì Binary Cross Entropy

Architecture Optimization – Tối Ưu Mô Hình:

1. Cách Import?

from torch.ao import quantization

from torch.ao.nn import quantized

1. Cơ Chế + 2 Tensor Lượng Tử?

* 2 Tensor lượng tử chuyển về Floating Point rồi mới +

1. Cơ Chế Nhân 1 Tensor Lượng Tử Với 1 Floating Point?

* Chuyển Tensor lượng tử thành Floating Point

1. Cơ Chế Nhân Ma Trận 2 Tensor Lượng Tử?

* Gọi A, B là 2 Tensor gốc, Aq, Bq là Tensor lượng tử tương ứng với giá trị là các số nguyên, sA, sB là Scale tương ứng, oA, oB là Offset tương ứng, ta muốn tính AB bằng Aq, Bq, sA, sB, oA, oB

* Số hạng đầu tiên tính = cách nhân ma trận số nguyên nên rất nhanh
* Số hạng thứ 2 và 3, do oA, oB đều là số nguyên, nên Broadcast rồi nhân ma trận rất nhanh
* Số hạng cuối thì khỏi bàn
* Tất cả sau đó chuyển thành Floating Point rồi + vào

1. Nhân Ma Trận 2 Tensor Lượng Tử?

<Output> = quantized.functional.linear(

<Input 1>, <Input 2>, <Bias>, <Output Scale>, <Output Offset>

)

* Mặc định

|  |  |
| --- | --- |
| <Bias> | None |
| <Output Scale> | <Input 1 Scale> |
| <Output Offset> | <Input 1 Offset> |

* <Input 1> là Tensor lượng tử có kiểu số nguyên không dấu 8 Bit, phải 2 chiều trở lên
* <Input 2> là Tensor lượng tử có kiểu số nguyên có dấu 8 Bit, phải 2 chiều
* <Bias> là Tensor thường
* <Output> là Tensor lượng tử có kiểu số nguyên không dấu 8 Bit

1. Lượng Tử Hóa 1 Module?

<Module Lượng Tử> = quantization.quantize\_dynamic(

<Module>, <Các Loại Lớp>, torch.qint8

)

* <Các Loại Lớp> là 1 Set có các phần tử là kiểu lớp, như lớp tuyến tính, lớp LSTM, …, tất cả các lớp trong <Module> có kiểu thuộc <Các Loại Lớp> sẽ được lượng tử hóa thành các lớp lượng tử động
* Ví dụ

foo = quantization.quantize\_dynamic(bar, {nn.Linear, nn.LSTM}, torch.qint8)

* <Module Lượng Tử> hoạt động đéo khác gì <Module>, nhưng bạn không có quyền truy cập các Parameter của các lớp bị lượng tử hóa, nói cách khác, không thể Train các Parameter này

1. Lớp Tuyến Tính Lượng Tử?

* Tạo 1 lớp tuyến tính lượng tử

<Lớp Tuyến Tính Lượng Tử> = quantized.Linear(

<Kích Thước Input>, <Kích Thước Output>

)

* <Lớp Tuyến Tính Lượng Tử> đéo khác gì lớp tuyến tính thường, chỉ có điều đéo Train được, <Weight> là 1 Tensor lượng tử có kiểu dữ liệu là số nguyên có dấu 8 Bit, mặc định có

Scale = 1 và Offset = 0, Bias là Tensor thường mặc định = 0

* Trả về Weight

<Lớp Tuyến Tính Lượng Tử>.weight()

* Trả về Bias

<Lớp Tuyến Tính Lượng Tử>.bias()

* Trả về 1 lúc cả Weight và Bias

<Weight>, <Bias> = <Lớp Tuyến Tính Lượng Tử>.\_packed\_params.\_weight\_bias()

* Để thay đổi Weight hoặc Bias

<Lớp Tuyến Tính Lượng Tử>.set\_weight\_bias(<Weight Mới>, <Bias Mới>)

* Forward

<Output> = <Lớp Tuyến Tính>(<Input>)

* <Input> phải là 1 Tensor lượng tử có kiểu dữ liệu là số nguyên không dấu 8 Bit
* <Output> cũng là 1 Tensor lượng tử có kiểu giống <Input>
* Mặc định

|  |  |
| --- | --- |
| <Output Scale> | 1 |
| <Output Offset> | 0 |

* Để thay đổi <Output Scale>

<Lớp Tuyến Tính Lượng Tử>.scale = <Output Scale Mới>

* Để thay đổi <Output Offset>

<Lớp Tuyến Tính Lượng Tử>.zero\_point = <Output Offset Mới>

1. Lớp Tuyến Tính Lượng Tử Động?

* Tạo 1 lớp tuyến tính lượng tử

<Lớp Tuyến Tính Lượng Tử Động> = quantized.dynamic.Linear(

<Kích Thước Input>, <Kích Thước Output>

)

* Đéo khác gì 1 lớp tuyến tính lượng tử thường, chỉ có điều giờ đây <Input> và <Output> sẽ là Tensor thường

Functional – Hàm:

1. Cách Import?

import torch.nn.functional as f

1. Trả Về Tensor Được Padding?

f.pad(<Tensor>, <Kích Thước Padding>, value = <Giá Trị Padding>)

* <Kích thước padding> là 1 Iterable gồm các phần tử là kích thước Padding trái, phải, trên, dưới, …
* Các phần tử ở vùng Padding sẽ có giá trị là <Giá Trị Padding>, mặc định = 0
* Ví dụ

foo = f.pad(bar, (2, 3), value = 7)

* Ta có

|  |  |
| --- | --- |
| bar | [1, 2, 3] |
| foo | [7, 7, 1, 2, 3, 7, 7, 7] |

* Tổng quát

|  |  |
| --- | --- |
| Số chiều <Tensor> | Thứ tự Padding |
| 1 | (trái, phải) |
| 2 | (trái, phải, trên, dưới) |
| 3 | (trái, phải, xa, gần, trên, dưới) |
| … | … |

* Nếu tưởng tượng Tensor là xấp ảnh, dễ thấy Padding theo tia Ox, Oy, Oz, …

1. Trả Về One Hot Vector Từ Index?

<One Hot Vector> = f.one\_hot(<Index Tensor>, <Số Kênh>)

* Mặc định

|  |  |
| --- | --- |
| <Số Kênh> | Max(<Index Tensor>) |

* <One Hot Vector> = <Index Tensor> nhưng mỗi phần tử được thay thế = 1 One Hot Vector tương ứng có Hot Index = giá trị của phần tử đó, số phần tử của Vector này = <Số Kênh>
* Ví dụ

foo = f.one\_hot(bar, 8)

* Ta có

|  |  |
| --- | --- |
| bar | [2, 5, 4] |
| foo | [[0, 0, 1, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 1, 0, 0],  [0, 0, 0, 0, 1, 0, 0, 0]] |

Optimizer – Tối Ưu Hóa:

1. Cách Import?

from torch import optim

1. Cách Sử Dụng Optimizer Nào Đó?

optimizer = optim.<Tên Optimizer>(<Dãy Các Tensor>, lr = <Learning Rate>)

optimizer.step()

* <Dãy Các Tensor> là bất kì đối tượng Iterable nào có phần tử là Tensor
* Cách hoạt động của “optimizer.step()” là nó cập nhật giá trị của tất cả Tensor trong <Dãy Các Tensor> dựa trên Gradient hiện tại của mỗi Tensor,

<Learning Rate>, và thêm 1 số yếu tố khác như Decay, EWMA, … mà nó lưu vào biến “optimizer” qua mỗi bước

* Để Reset Gradient của mọi Tensor trong <Dãy Các Tensor> về mo, viết

optimizer.zero\_grad()

Torch Vision:

1. Cách Import?

from torchvision import transforms

1. Tạo Hàm Có Tác Dụng Biến Đổi PIL Image Thành Tensor?

<Hàm Biến Đổi> = transforms.PILToTensor()

* Để biến đổi

<Tensor Image> = <Hàm Biến Đổi>(<PIL Image>)

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <PIL Image> Shape | <Tensor Image> Shape |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>,  <Số Kênh Ảnh>) | (<Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (1,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) |

* Thứ tự kênh sẽ là R, G, B cho ảnh RGB

1. Tạo Hàm Có Tác Dụng Biến Đổi Numpy Image Hoặc PIL Image Thành Tensor Và Chuẩn Hóa Nó?

<Hàm Biến Đổi> = transforms.ToTensor()

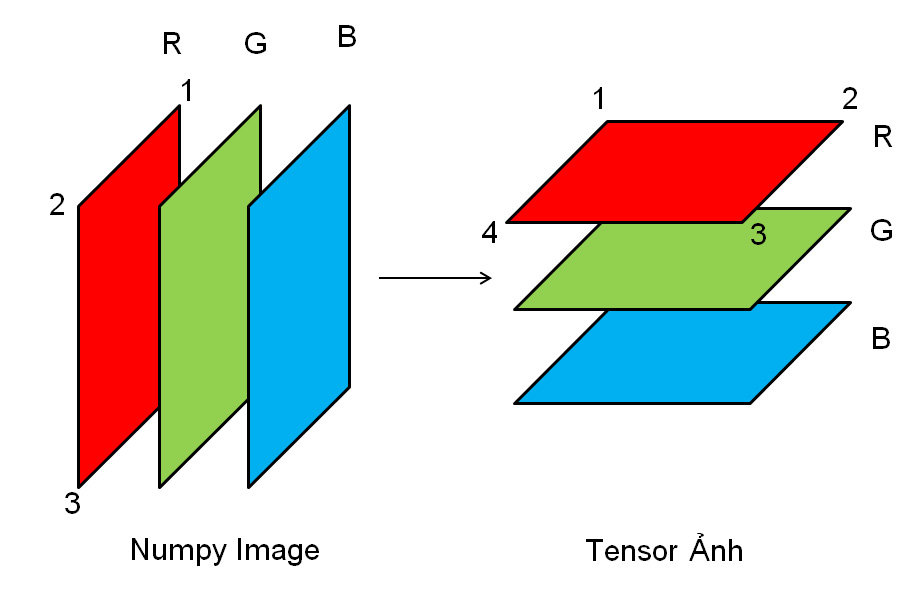
* Để biến đổi

<Tensor Image> = <Hàm Biến Đổi>(<Numpy Image Hoặc PIL Image>)

* Nếu <Numpy Image Hoặc PIL Image> có kiểu dữ liệu Int thì sẽ được chuẩn hóa

<Tensor Image> = (<Numpy Image Hoặc PIL Image> % 256) / 255

* Do đó mỗi phần tử của <Tensor Image> có giá trị từ 0 đến 1
* Nhưng nếu <Numpy Image Hoặc PIL Image> có kiểu dữ liệu Float thì giữ nguyên, nghĩa là không được chuẩn hóa
* Minh họa



* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <PIL Image> Shape | <Tensor Image> Shape |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>,  <Số Kênh Ảnh>) | (<Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (1,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) |

1. Tạo Hàm Có Tác Dụng Thay Đổi Kích Thước Ảnh?

<Hàm Biến Đổi> = transforms.Resize(

<Output Shape>, max\_size = <Kích Thước Tối Đa>, antialias = True

)

* <Output Shape> là 1 Tuple, phần tử thứ 1 là chiều dọc, thứ 2 là chiều ngang
* Mặc định

|  |  |
| --- | --- |
| <Kích Thước Tối Đa> | None |

* Để biến đổi

<Ảnh Biến Đổi> = <Hàm Biến Đổi>(<Ảnh>)

* Trường hợp 1, <Ảnh> là PIL Image
* <Ảnh Biến Đổi> sẽ là PIL Image
* Chiều dọc vào chiều ngang của <Ảnh> sẽ Scale theo hệ số khác nhau sao cho kích thước cuối cùng = <Output Shape>
* Nếu chỉ định <Output shape> là 1 số, thì cả chiều dọc và chiều ngang của <Ảnh> đều Scale theo cùng 1 hệ số sao cho chiều nào ngắn hơn sau khi Scale sẽ = <Output Shape>, đồng thời sau khi Scale, nếu chiều nào dài hơn mà vượt quá <Kích Thước Tối Đa> thì cả chiều dọc và chiều ngang sẽ tiếp tục được Scale lại theo cùng 1 hệ số sao cho chiều nào dài hơn sau khi Scale sẽ = <Kích Thước Tối Đa>
* Khi Scale sẽ sử dụng nội suy màu
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Ảnh> Shape | <Ảnh Biến Đổi> Shape |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>,  <Số Kênh Ảnh>) | (<Kích Thước Chiều Dọc Ảnh Biến Đổi>,  <Kích Thước Chiều Ngang Ảnh Biến Đổi>,  <Số Kênh Ảnh>) |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (<Kích Thước Chiều Dọc Ảnh Biến Đổi>,  <Kích Thước Chiều Ngang Ảnh Biến Đổi>) |

* Trường hợp 2, <Ảnh> là 1 hay Batch Tensor Image
* <Ảnh Biến Đổi> sẽ là 1 hay Batch Tensor Image
* Cách biến đổi giống trường hợp 1
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Ảnh> Shape | <Ảnh Biến Đổi> Shape |
| (<Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (<Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh Biến Đổi>,  <Kích Thước Chiều Ngang Ảnh Biến Đổi>) |
| (<Kích Thước Batch>,  <Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (<Kích Thước Batch>,  <Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh Biến Đổi>,  <Kích Thước Chiều Ngang Ảnh Biến Đổi>) |

1. Tạo Hàm Có Tác Dụng Chuẩn Hóa Tensor Image?

<Hàm Biến Đổi> = transforms.Normalize(<Mean>, <Độ Lệch Chuẩn>)

* Để biến đổi

<Tensor Image Chuẩn Hóa> = <Hàm Biến Đổi>(<Tensor Image>)

* Đầu tiên, nếu <Mean> có Shape là (k) thì sẽ được Reshape lại thành (k, 1, 1), tương tự với <Độ Lệch Chuẩn>, các trường hợp còn lại giữ nguyên
* Tiếp theo, tiến hành chuẩn hóa, <Mean> và <Độ Lệch Chuẩn> phải Broadcast được với <Tensor Image>

<Tensor Image Chuẩn Hóa> = (<Tensor Image> – <Mean>) / <Độ Lệch Chuẩn>

* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Tensor Image> Shape | <Tensor Image Chuẩn Hóa> Shape |
| (…,  <Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (…,  <Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) |

1. Tạo Hàm Có Tác Dụng Quay Ảnh 1 Góc Ngẫu Nhiên?

<Hàm Biến Đổi> = transforms.RandomRotation(

<Giới Hạn Quay>,

expand = <Có Mở Rộng Không>, center = <Tâm Quay>, fill = <Màu Fill>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Có Mở Rộng Không> | False | <Tâm Quay> | None |
| <Màu Fill> | 0 |  |  |

* <Giới Hạn Quay> là 1 Iterable bao gồm 2 phần tử là cận dưới và cận trên
* <Tâm Quay> là 1 Iterable bao gồm 2 phần tử là tọa độ ngang và dọc
* <Màu Fill> là 1 Iterable có số phần tử = <Số Kênh Ảnh>
* Để biến đổi

<Ảnh Đã Quay> = <Hàm Biến Đổi>(<Ảnh>)

* Trường hợp 1, <Ảnh> là PIL Image
* <Ảnh Đã Quay > sẽ là PIL Image
* Ban đầu, tách riêng <Ảnh> và khung của nó ra, xoay <Ảnh> ngược chiều kim đồng hồ 1 góc ngẫu nhiên nằm trong <Giới Hạn Quay>
* Nếu chỉ định <Giới Hạn Quay> là 1 số, thì góc ngẫu nhiên sẽ nằm trong khoảng từ

–<Giới Hạn Quay> đến <Giới Hạn Quay>

* Tâm quay là chính giữa ảnh nếu <Tâm Quay> không được chỉ định, còn không thì tâm quay sẽ là <Tâm Quay>
* Sau khi quay, khung của <Ảnh> sẽ Scale sao cho nó = kích thước Bounding Box của <Ảnh> sau khi quay, tâm Scale = tâm của khung, sau đó, những chỗ nào trong khung không bị <Ảnh> đè lên sẽ được tô màu, màu = <Màu Fill>, dùng thang màu của <Ảnh>
* Nếu chỉ định <Màu Fill> là 1 số, thì nó sẽ là giá trị cho tất cả các kênh màu
* <Ảnh Đã Quay> là tất cả những gì nằm trong khung
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Ảnh> Shape | <Ảnh Đã Quay> Shape |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>,  <Số Kênh Ảnh>) | (<Kích Thước Chiều Dọc Ảnh Đã Quay>,  <Kích Thước Chiều Ngang Ảnh Đã Quay>,  <Số Kênh Ảnh>) |
| (<Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (<Kích Thước Chiều Dọc Ảnh Đã Quay>,  <Kích Thước Chiều Ngang Ảnh Đã Quay>) |

* Trường hợp 2, <Ảnh> là 1 hay Batch Tensor Image
* <Ảnh Đã Quay> sẽ là 1 hay Batch Tensor Image
* Cách biến đổi giống trường hợp 1, nếu là Batch thì tất cả ảnh trong Batch đều quay cùng 1 góc, cùng 1 <Tâm Quay>, cùng 1 <Màu Fill>
* Tóm tắt các trường hợp

|  |  |
| --- | --- |
| <Ảnh> Shape | <Ảnh Đã Quay> Shape |
| (<Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (<Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh Đã Quay>,  <Kích Thước Chiều Ngang Ảnh Đã Quay>) |
| (<Kích Thước Batch>,  <Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh>,  <Kích Thước Chiều Ngang Ảnh>) | (<Kích Thước Batch>,  <Số Kênh Ảnh>,  <Kích Thước Chiều Dọc Ảnh Đã Quay>,  <Kích Thước Chiều Ngang Ảnh Đã Quay>) |

1. Tạo Hàm Có Tác Dụng Flip Ảnh Qua Trục Dọc 1 Cách Ngẫu Nhiên?

<Hàm Biến Đổi> = transforms.RandomRotation(<Tỉ Lệ Flip>)

* Mặc định

|  |  |
| --- | --- |
| <Tỉ Lệ Flip> | 0.5 |

* Để biến đổi

<Ảnh Bị Flip> = <Hàm Biến Đổi>(<Ảnh>)

* <Ảnh> có thể là PIL Image hoặc Tensor, không giới hạn số chiều
* Nếu <Ảnh> là Tensor thì tương đương nguyên khối Tensor sẽ bị Scale –1 lần theo chiều cuối cùng

1. Tạo Hàm Có Tác Dụng Crop Ảnh 1 Cách Ngẫu Nhiên?

Hàm Biến Đổi> = transforms.RandomCrop(

<Crop Shape>, <Padding>, <Có Padding Thêm Không>, <Màu Fill>, <Chế Độ Padding>

)

* <Padding> là Iterable gồm 4 phần tử là Padding trái, trên, phải, dưới
* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Padding> | None | <Có Padding Thêm Không> | False |
| <Màu Fill> | 0 | <Chế Độ Padding> | "constant" |

* Để biến đổi

<Ảnh Bị Crop> = <Hàm Biến Đổi>(<Ảnh>)

* <Ảnh> có thể là PIL Image hoặc 2D trở lên Tensor
* Xét <Ảnh> là Tensor, nếu là PIL Image thì suy ra tương tự
* Ban đầu, nếu chỉ định <Padidng>, thì <Ảnh> sẽ được Padding tất cả các cạnh tương ứng ở 2 chiều cuối cùng
* Nếu <Padding> chỉ có 2 phần tử, thì phần tử 1 là Padding trái phải, phần tử 2 là Padding trên dưới
* Nếu <Padding> chỉ có 1 phần tử, thì Padding tất cả các cạnh như nhau
* Sau đó, nếu trong 2 chiều cuối cùng sau khi Padding, có ít nhất 1 chiều < chiều tương ứng trong <Crop Shape>, thì sẽ báo lỗi, nhưng nếu <Có Padding Thêm Không> = True, thì <Ảnh> sẽ tiếp tục được Padding 1 cách ngẫu nhiên sao cho vừa đủ khít với <Crop Shape>
* Nếu <Chế Độ Padding> = "constant" thì tất cả những chỗ Padding sẽ được tô màu =

<Màu Fill>, cách tô y chang khi quay ảnh, trừ việc <Màu Fill> chỉ có thể là 1 số với trường hợp <Ảnh> là Tensor

* Nếu <Chế Độ Padding> = "symmetric" thì tất cả những chỗ Padding sẽ được đặt những phiên bản Copy của <Ảnh> kề nhau như những ô gách lát nền, 2 ô gạch kề nhau là Flip của nhau, <Màu Fill> trở nên vô nghĩa
* Cuối cùng, tiến hành Crop <Ảnh> đã được Padding, lưu ý không Crop ra ngoài <Ảnh>
* Khi Crop thì Crop nguyên khối Tensor, nghĩa là nếu Tensor 3D, 4D, … thì tất cả tấm ảnh trong đó đều được Padding rồi Crop như nhau

1. Gộp Chung Nhiều Hàm Biến Đổi Thành 1 Hàm Duy Nhất?

<Hàm Biến Đổi Tổng Hợp> = transforms.Compose(<Các Hàm Biến Đổi>)

* <Các Hàm Biến Đổi> là 1 Iterable chứa các phần tử là <Hàm Biến Đổi>
* Ví dụ

foo = transforms.Compose([

Transforms.ToTensor(),

Transforms.Resize((500, 600), antialias = True),

Transforms.Normalize(1, 2)

])

1. Cách Sử Dụng Hàm Biến Đổi Nhanh?

transforms.functional.<Tên Hàm Biến Đổi Viết Thường>(<Ảnh>, <Các Tham Số>)

* Ví dụ

foo = transforms.functional.resize(bar, (500, 600), antialias = True)

Data – Dữ Liệu:

1. Cách Import?

from torchvision import datasets

from torch.utils.data import Dataset, DataLoader as loader

1. Cách Tạo 1 Dataset Tiêu Chuẩn?

class <Tên Dataset>(Dataset):

def \_\_init\_\_(self, <Các Tham Số>):

<Làm Gì Đó Khi Khởi Tạo>

def \_\_len\_\_(self):

return <Kích Thước Dataset>

def \_\_getitem\_\_(self, <Index>):

<Load Dữ Liệu Từ Đường Dẫn Với Index>

return <Dữ Liệu>

1. Cách Sử Dụng Data Loader Để Xáo Trộn Dataset Và Chia Thành Các Batch?

<Data Loader> = loader(

<Dataset>, <Kích Thước Batch>, <Có Trộn Không>,

num\_workers = <Số Core>, drop\_last = <Có Bỏ Batch Cuối Không>

)

* Mặc định

|  |  |  |  |
| --- | --- | --- | --- |
| <Kích Thước Batch> | 1 | <Có Trộn Không> | False |
| <Có Bỏ Batch Cuối Không> | False | <Số Core> | 0 |

* <Data Loader> hoạt động giống như 1 Generator vô hạn, bản chất nó không lưu trữ các Batch của <Dataset> mà lưu trữ các Batch <Index> của nó
* <Data Loader> sẽ gọi hàm “\_\_len\_\_” của <Dataset> để nhận được kích thước là N, khi đó tập hợp các <Index> là (0, 1, 2, …, N – 1), nó sẽ được chia thành các Batch lần lượt từ trái qua phải đều nhau, nếu N không chia hết cho <Kích Thước Batch> thì Batch cuối sẽ là phần thừa, Batch này sẽ bị bỏ qua nếu <Có Bỏ Batch Cuối Không> = True
* Nếu <Có Trộn Không> = True, tập hợp các <Index> sẽ bị xáo trộn lại mỗi lần duyệt hết tập hợp các <Index>
* Sau mỗi lần lấy được Batch <Index> thì <Data Loader> sẽ Pass mỗi <Index> trong cái Batch này vào hàm “\_\_getitem\_\_” của <Dataset> để nó trả về dữ liệu, sẽ có <Số Core> Core + thêm Core chính đang làm việc trong CPU để Load dữ liệu, sau đó nó sẽ trả về Batch dữ liệu

1. Tải Nhanh Dataset Có Sẵn Nào Đó?

<Dataset> = datasets.MNIST(

download = <Có Tải Không>,

root = <Đường Dẫn>,

train = <Tải Dữ Liệu Train À>,

transform = <Hàm Biến Đổi>

)

* <Đường Dẫn> là đường dẫn từ thư mục làm việc hiện tại đến thư mục con là nơi chứa dữ liệu, chưa có thư mục thì tự động tạo
* Ví dụ
* Thư mục làm việc hiện tại là “/content”, nếu <Đường Dẫn> = "foo/bar/foobar" thì thư mục chứa dữ liệu là "/content/foo/bar/foobar"
* Trường hợp <Có Tải Không> = True thì nó tự động tải nếu chưa có dữ liệu trong <Đường Dẫn> hoặc không tải mà dùng lại dữ liệu cũ nếu đã có rồi, nếu không có kết nối Internet thì sẽ thông báo lỗi, trường hợp = False thì nếu dữ liệu đã tải rồi thì khỏi tại lại và khi không có Internet thì nó sẽ không báo lỗi, nếu dữ liệu đã có rồi thì nó sẽ báo lỗi
* Nếu <Tải Dữ Liệu Train À> = True thì dữ liệu tải về là dữ liệu Train, còn không thì Test
* <Hàm Biến Đổi> sẽ được thực thi lên phần <Input> trả về mỗi khi hàm “\_\_getitem\_\_” của

<Dataset> được gọi