

NGUYỄN THANH TUẤN

Deep Learning

Cơ Bản



TÁI BẢN
LẦN THỨ 2

Copyright © 2019 Nguyễn Thanh Tuấn

NTTUAN8.COM

The Legrand Orange Book Template by Mathias Legrand is used

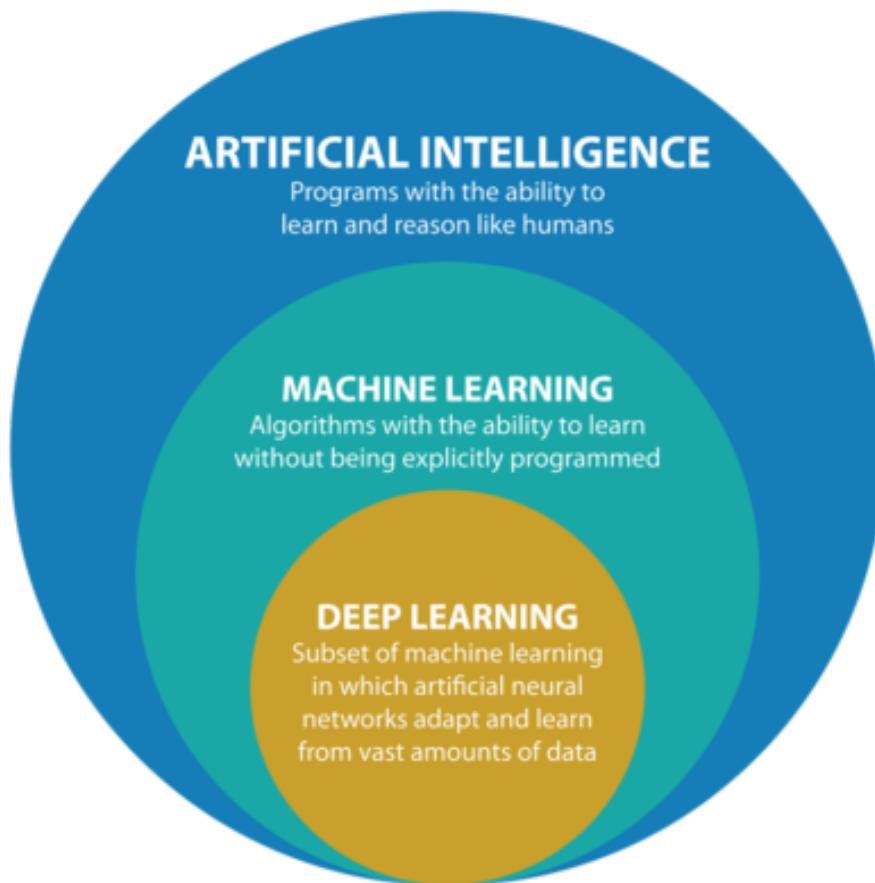
The book cover is designed by Nguyễn Thanh Tú

Version 2, last update, August 2020

Lời mở đầu

Giới thiệu về Deep Learning

Những năm gần đây, AI - Artificial Intelligence (Trí Tuệ Nhân Tạo), và cụ thể hơn là Machine Learning (Máy Học) nổi lên như một minh chứng của cuộc cách mạng công nghiệp lần thứ tư (1 - động cơ hơi nước, 2 - năng lượng điện, 3 - công nghệ thông tin). AI hiện diện trong mọi lĩnh vực của đời sống con người, từ kinh tế, giáo dục, y khoa cho đến những công việc nhà, giải trí hay thậm chí là trong quân sự. Những ứng dụng nổi bật trong việc phát triển AI đến từ nhiều lĩnh vực để giải quyết nhiều vấn đề khác nhau. Nhưng những đột phá phần nhiều đến từ Deep Learning (học sâu) - một mảng nhỏ đang mở rộng dần đến từng loại công việc, từ đơn giản đến phức tạp. Deep Learning đã giúp máy tính thực thi những việc tưởng chừng như không thể vào 15 năm trước: phân loại cả ngàn vật thể khác nhau trong các bức ảnh, tự tạo chú thích cho ảnh, bắt chước giọng nói và chữ viết của con người, giao tiếp với con người, hay thậm chí cả sáng tác văn, phim, ảnh, âm nhạc.



Hình 1: Mối quan hệ AI, ML và DL [3]

Chúng ta có thể thấy Deep learning chỉ là một nhánh nhỏ của Machine Learning. Tuy nhiên trong khoảng 5 năm trở lại đây thì Deep Learning được nhắc đến rất nhiều như một xu hướng mới của cuộc cách mạng AI. Có một số lý do như sau:

- Bùng nổ dữ liệu: Deep learning khai thác được Big Data (dữ liệu lớn) cùng với độ chính xác cao hơn hẳn so với các phương pháp Machine Learning khác trên tập dữ liệu đặc biệt là đối với ảnh. Cụ thể là năm 2012, Alex Krizhevsky, Ilya Sutskever, và người hướng dẫn là Hinton, submit một model làm bất ngờ những người làm việc trong ngành AI, và sau này là cả thế giới khi đạt top-5 error là 16% trong cuộc thi ILSVRC2012. Đây là lần đầu tiên một model Artificial Neural Network (ANN) đạt kết quả state-of-the-art (SOTA).

- Phần cứng phát triển: Sự xuất hiện của GPU GTX 10 series của NVIDIA ra mắt năm 2014 với hiệu năng tính toán cao cũng như giá thành rẻ có thể tiếp cận với hầu hết với mọi người dẫn đến việc nghiên cứu Deep Learning không còn là những bài toán chỉ được nghiên cứu trong các phòng lab đắt tiền của các trường Đại học danh giá và các công ty lớn.

Theo thống kê trên trang [paperswithcode](#) hiện có 16 tasks lớn mà Machine Learning có thể thực hiện trong đó có tới trên 8 tasks Deep learning đạt kết quả SOTA phải kể đến như:

- Computer Vision
- Natural Language Processing
- Medical
- Methodology
- Speech
- Time Series
- Audio
- Music

Ý tưởng và mục đích của cuốn sách

Hồi đầu năm 2019, khi nghiên cứu ứng dụng về Deep Learning trong ngành Y, tôi nhận ra là mặc dù bản thân mình là kỹ sư có khả năng lập trình Deep Learning nhưng lại thiếu kiến thức chuyên môn ngành Y để phát triển ứng dụng chuyên sâu. Ngược lại, các bác sĩ hiểu được các vấn đề chuyên môn thì lại thiếu các kỹ năng lập trình cần thiết.

Thế nên tôi quyết định viết loạt bài viết này để giới thiệu các kiến thức cơ bản về Deep Learning cũng như các ứng dụng của nó để mọi người có kiến thức chuyên môn, có dữ liệu trong các ngành khác như y tế, ngân hàng, nông nghiệp,... có thể tự áp dụng được Deep Learning trong lĩnh vực của họ.

Thêm vào đó tôi muốn cung cấp một nền tảng về toán và Deep Learning cơ bản cho các bạn học sinh, sinh viên có thể làm được ứng dụng và đào sâu nghiên cứu về deep learning trong môi trường học thuật.

Vì hướng tới nhiều độc giả với các background khác nhau nên khi viết tôi giải thích toán chi tiết nhưng đơn giản và dễ hiểu. Bên cạnh đó tôi cũng có các bài ứng dụng Deep Learning trong thực tế xen kẽ giữa các nội dung lý thuyết để bạn đọc dễ tiếp thu hơn.

Cuối cùng, hy vọng qua cuốn sách, bạn đọc có được những kiến thức cơ bản về Deep Learning và thấy được các ứng dụng của nó. Để rồi áp dụng các ý tưởng vào start-up, công ty để có các ứng dụng hay, thiết thực cho xã hội. Bên cạnh đó mong rằng cuốn sách là bệ phóng cho các bạn sinh viên Việt Nam nghiên cứu thêm về Deep Learning để có các nghiên cứu, thuật toán mới.

Yêu cầu

Vì cuốn sách này tôi muốn viết cho tất cả mọi người nên tôi sẽ giải thích tất cả mọi thứ chi tiết nhất có thể. Một số yêu cầu để có thể theo nội dung sách:

- Kiến thức về toán cơ bản cấp ba: hàm số, đạo hàm.
- Kiến thức cơ bản về lập trình Python: biến, vòng lặp (tôi có giới thiệu ở phần dưới)
- Ý thức tự học hỏi kiến thức mới.

Nội dung

Chương I, tôi giới thiệu về cách cài đặt môi trường với Anaconda để chạy code Python cơ bản. Ngoài ra tôi cũng hướng dẫn sử dụng Google Colab, với GPU Tesla K80 được Google cung cấp

miễn phí. Nên bạn đọc có thể train model online thay vì sử dụng máy tính cá nhân.

Chương II, tôi đề cập đến Machine Learning cơ bản với hai thuật toán Linear Regression và Logistic Regression. Đồng thời tôi giới thiệu về thuật toán Gradient descent, rất quan trọng trong Deep Learning. Bên cạnh đó tôi giới thiệu các kiến thức Toán cơ bản như: phép toán với ma trận, biểu diễn bài toán dạng ma trận,...

Chương III, tôi giới thiệu về bài toán Neural Network cũng chính là xương sống của Deep Learning và thuật toán Backpropagation để giải bài toán này. Ngoài ra, để hiểu rõ bản chất của Neural Network nên tôi cũng hướng dẫn mọi người code từ đầu Neural Network và Backpropagation bằng Python trong chương này.

Chương IV, tôi đề cập tới Convolutional Neural Network (CNN) cho bài toán có xử lý ảnh. Sau đó giới thiệu về thư viện Keras và ứng dụng CNN cho bài toán phân loại ảnh với bộ dữ liệu chữ số viết tay (MNIST). Cuối chương tôi giới thiệu về ứng dụng thực tế của CNN cho bài toán ô tô tự lái.

Chương V, tôi giới thiệu một số tips trong Deep Learning như transfer learning, data augmentation, mini-batch gradient descent, dropout, non-linear activation, ... để tăng độ hiệu quả của mô hình.

Chương VI, tiếp nối ý tưởng từ chương IV , tôi đề cập đến hai bài toán lớn của Deep Learning trong Computer Vision. Đó là bài toán về Object Detection và Image Segmentation. Bên cạnh đó tôi có hướng dẫn các bước làm bài toán detect biển số xe máy.

Chương VII, tôi giới thiệu về thuật toán Recurrent Neural Network (RNN) cho bài toán dữ liệu dạng chuỗi và mô hình cải tiến của nó là Long Short Term Memory (LSTM). Sau đó tôi hướng dẫn mọi người áp dụng mô hình LSTM cho bài toán thêm mô tả cho ảnh. Cuối cùng tôi giới thiệu mạng sequence to sequence (seq2seq) cho bài toán dịch cùng cơ chế attention.

Chương cuối tôi giới thiệu về mạng generative adversarial networks (GAN) với một số mô hình GAN như deep convolutional GAN (DCGAN), conditional GAN (cGAN).

Ngoài ra trong cuối mỗi chương tôi đều đưa ra bài tập về thực hành code với Python và đặt ra những câu hỏi để mọi người hiểu rõ thêm về lý thuyết mà tôi đã giới thiệu.

Thông tin liên lạc

Website của [tôi](#).

Facebook cá nhân của [tôi](#).

Group để hỏi đáp tại [đây](#).

Page cập nhật tin tức mới về AI tại [đây](#).

Tất cả code trên sách ở trên [github](#).

Vì đây là bản đầu tiên của cuốn sách nên mọi người có nhận xét, góp ý, phản ánh xin gửi về mail nttuan8.com@gmail.com

Xin cảm ơn mọi người rất nhiều!

Lời cảm ơn

Trước hết tôi xin cảm ơn bạn bè trên Facebook đã nhiệt tình ủng hộ và đóng góp cho các bài viết trong series Deep Learning cơ bản từ những ngày đầu tiên. Các bạn là động lực lớn nhất cho tôi để hoàn thành series và xuất bản sách này.

Xin cảm ơn bạn Lê Vũ Hoàng (Nghiên cứu sinh ngành thống kê, ĐH Trinity College Dublin) đã giúp tôi đọc và chỉnh sửa các bài viết trên blog trước khi đến tay bạn đọc cũng như giúp tôi chỉnh sửa nội dung khi soạn sách.

Xin cảm ơn bạn Nguyễn Thê Hùng (Toán tin - K59 - ĐHBKHN), Nguyễn Thị Xuân Huyền (Điện tử Viễn thông - K60 - ĐHBKHN) đã giúp tôi đọc và chỉnh sửa nội dung sách.

Cuối cùng và quan trọng nhất, tôi xin cảm ơn gia đình, người thân những người luôn động viên và ủng hộ tôi trong dự án này.

Sách tái bản lần thứ hai

Trong quá trình đi dạy và đi làm tôi thấy rất nhiều câu hỏi hay về machine learning, deep learning như: L1 và L2 loss khác nhau thế nào, tại sao L1 tốt cho outliers?,... Nên tôi cập nhật sách để cập nhật các kiến thức cho bạn đọc. Thêm vào đó, qua quá trình dạy tôi sẽ sửa sách theo hướng giải thích các kiến thức đơn giản, dễ hiểu hơn cho bạn đọc. Bên cạnh đó, sách vẫn còn những lỗi nhỏ nên tôi muốn sửa và cập nhật. Các thay đổi trong lần tái bản thứ hai:

- Thêm chương GAN.
- Viết bài sequence to sequence và attention.
- Viết bài hướng dẫn dùng yolo để detect biển số xe.
- Thêm phần visualize CNN.
- So sánh các loss function cho linear regression.
- Giải linear regression bằng đại số tuyến tính.
- Viết lại phần python cơ bản.
- Thêm phần word embedding.
- Thêm phần batch normalization.
- Hướng dẫn lưu model trong machine learning dùng numpy.
- Đọc soát lại và sửa nội dung các phần trong sách.

Tôi xin cảm ơn bạn đọc đã nhiệt tình ủng hộ và đóng góp cho sách Deep Learning cơ bản từ những ngày đầu tiên xuất bản. Các bạn là động lực lớn nhất cho tôi để tái bản cuốn sách này.

Xin cảm ơn bạn Bùi Thị Liên (Công nghệ thông tin - K61 - ĐHBKHN), Trần Bảo Hiếu (Công nghệ thông tin Việt Nhật Hedspi - K61 - ĐHBKHN), Hoàng Đức Việt (Khoa Học Máy Tính - K63 - ĐHBKHN), Trần Hữu Huy (Khoa Học Máy Tính - K63 - ĐHBKHN), Phạm Văn Hạnh (Khoa Học Máy Tính - K63 - ĐHBKHN), Nguyễn Duy Đạt (Nghiên cứu sinh ngành khoa học máy tính, Đại học Taiwan Tech), Nguyễn Hoàng Minh (K34 Chuyên Lý - THPT KHTN) đã giúp tôi viết các phần mới cũng như đọc và chỉnh sửa sách trong lần tái bản này.

Cuối cùng và quan trọng nhất, tôi xin cảm ơn gia đình, người thân, bạn bè những người luôn bên cạnh động viên và ủng hộ tôi.



Hình 2: Ảnh chụp ở Stonehenge

"*Người vá trời lắp bể*
Kẻ đắp luỹ xây thành
Ta chỉ là chiếc lá
Việc của mình là xanh"

Nguồn: Trái tim người lính (thơ), Nguyễn Sĩ Đại, NXB Thanh niên.

Từ những ngày đầu tiên viết blog tôi luôn quan niệm "chia sẻ là để học hỏi" thế nên "kiến thức là để cho đi". Cuốn sách này được chia sẻ miễn phí tới bạn đọc với thông điệp:

"Vì một cộng đồng AI Việt Nam phát triển bền vững"

Mục lục

	Giới thiệu
1 Cài đặt môi trường	19
1.1 Giới thiệu	19
1.2 Google Colab	19
1.2.1 Tạo file trên google colab	19
1.2.2 Chọn GPU	20
1.2.3 Các thành phần nhỏ	21
1.2.4 Link với google drive	22
1.2.5 Cài thêm thư viện	24
1.3 Hướng dẫn cài đặt anaconda	24
1.3.1 Giới thiệu	24
1.3.2 Yêu cầu phần cứng và phần mềm	25
1.3.3 Cài đặt	25
1.3.4 Hướng dẫn sử dụng Jupyter notebook	31
2 Python cơ bản	33
2.1 Kiểu dữ liệu cơ bản	33
2.1.1 Số	33
2.1.2 Phép tính logic	34
2.1.3 Chuỗi	34
2.2 Containers	35
2.2.1 List	35
2.2.2 Dictionaries	35

2.3	Function	36
2.4	Thư viện trong python	37
2.5	Thư viện Numpy	37
2.5.1	Array indexing	38
2.5.2	Các phép tính trên array	38
2.5.3	Broadcasting	40
2.6	Matplotlib	40

II

Machine learning cơ bản

3	Linear regression	47
3.1	Bài toán	47
3.2	Thiết lập công thức	48
3.2.1	Model	48
3.2.2	Loss function	48
3.3	Thuật toán gradient descent	50
3.3.1	Đạo hàm là gì	50
3.3.2	Gradient descent	51
3.3.3	Áp dụng vào bài toán	54
3.4	Ma trận	55
3.4.1	Ma trận là gì	55
3.4.2	Phép nhân ma trận	56
3.4.3	Element-wise multiplication matrix	56
3.4.4	Biểu diễn bài toán	56
3.5	So sánh các loss function	57
3.5.1	Mean Absolute Error, L1 Loss	57
3.5.2	Mean Square Error, L2 Loss	57
3.5.3	So sánh MAE và MSE (L1 Loss và L2 Loss)	58
3.6	Giải bằng đại số tuyến tính	59
3.7	Python code	61
3.8	Bài tập	63
4	Logistic regression	65
4.1	Bài toán	65
4.2	Xác suất	66
4.3	Hàm sigmoid	67
4.4	Thiết lập bài toán	68
4.4.1	Model	68
4.4.2	Loss function	69
4.5	Chain rule	70
4.5.1	Áp dụng gradient descent	73
4.5.2	Biểu diễn bài toán dưới ma trận	75
4.5.3	Hàm sigmoid được chọn để đạo hàm đẹp	75

4.6	Quan hệ giữa phần trăm và đường thẳng	76
4.7	Ứng dụng	78
4.8	Python code	78
4.9	Bài tập	80

III

Neural Network

5	Neural network	83
5.1	Neural network là gì	83
5.1.1	Hoạt động của các nơron	83
5.2	Mô hình neural network	84
5.2.1	Logistic regression	84
5.2.2	Mô hình tổng quát	85
5.2.3	Kí hiệu	85
5.3	Feedforward	87
5.3.1	Biểu diễn dưới dạng ma trận	88
5.4	Logistic regression với toán tử XOR	89
5.4.1	NOT	90
5.4.2	AND	90
5.4.3	OR	92
5.4.4	XOR	93
5.5	Bài tập	94
6	Backpropagation	97
6.1	Bài toán XOR với neural network	97
6.1.1	Model	97
6.1.2	Loss function	98
6.1.3	Gradient descent	99
6.2	Mô hình tổng quát	103
6.3	Python code	104
6.4	Bài tập	106

IV

Convolutional Neural Network

7	Giới thiệu về xử lý ảnh	109
7.1	Ảnh trong máy tính	109
7.1.1	Hệ màu RGB	109
7.1.2	Ảnh màu	110
7.1.3	Tensor là gì	112
7.1.4	Ảnh xám	114
7.1.5	Chuyển hệ màu của ảnh	115

7.2	Phép tính convolution	115
7.2.1	Convolution	115
7.2.2	Padding	117
7.2.3	Stride	118
7.2.4	Ý nghĩa của phép tính convolution	119
7.3	Bài tập	120
8	Convolutional neural network	123
8.1	Thiết lập bài toán	123
8.2	Convolutional neural network	124
8.2.1	Convolutional layer	124
8.2.2	Pooling layer	129
8.2.3	Fully connected layer	130
8.3	Mạng VGG 16	131
8.4	Visualizing Convolutional Neural Network	132
8.4.1	Visualizing Feature Maps	132
8.4.2	Visualizing Convolutional Filters	134
8.4.3	Visualizing Class Outputs	135
8.4.4	Visualizing Attention Map	135
8.5	Bài tập	137
9	Giới thiệu keras và bài toán phân loại ảnh	139
9.1	Giới thiệu về keras	139
9.2	MNIST Dataset	140
9.2.1	Xây dựng bài toán	140
9.2.2	Chuẩn bị dữ liệu	140
9.2.3	Xây dựng model	142
9.2.4	Loss function	143
9.3	Python code	145
9.4	Ứng dụng của việc phân loại ảnh	147
9.5	Bài tập	147
10	Ứng dụng CNN cho ô tô tự lái	149
10.1	Giới thiệu mô phỏng ô tô tự lái	149
10.2	Bài toán ô tô tự lái	152
10.2.1	Xây dựng bài toán	152
10.2.2	Chuẩn bị dữ liệu	152
10.2.3	Tiền xử lý dữ liệu (Preprocessing)	152
10.2.4	Xây dựng model	154
10.2.5	Loss function	155
10.3	Python code	155
10.4	Áp dụng model cho ô tô tự lái	157
10.5	Bài tập	157

11 Transfer learning và data augmentation	161
11.1 Transfer learning	161
11.1.1 Feature extractor	162
11.1.2 Fine tuning	165
11.1.3 Khi nào nên dùng transfer learning	170
11.2 Data augmentation	170
11.3 Bài tập	174
12 Các kỹ thuật cơ bản trong deep learning	177
12.1 Vectorization	177
12.2 Mini-batch gradient descent	178
12.2.1 Mini-batch gradient descent là gì	178
12.2.2 Các thông số trong mini-batch gradient descent	180
12.3 Bias và variance	181
12.3.1 Bias, variance là gì	181
12.3.2 Bias, variance tradeoff	182
12.3.3 Đánh giá bias and variance	182
12.4 Dropout	183
12.4.1 Dropout là gì	183
12.4.2 Dropout hạn chế việc overfitting	183
12.4.3 Lời khuyên khi dùng dropout	183
12.5 Activation function	184
12.5.1 Non-linear activation function	184
12.5.2 Vanishing và exploding gradient	184
12.5.3 Một số activation thông dụng	186
12.6 Batch Normalize	188
12.6.1 Phân tích nguyên nhân	188
12.6.2 Batch Normalization ra đời	189
12.6.3 Hiệu quả của batch normalization	190
12.7 Bài tập	190

13 Object detection với Faster R-CNN	193
13.1 Bài toán object detection	193
13.2 Faster R-CNN	194
13.2.1 R-CNN (Region with CNN feature)	194
13.2.2 Fast R-CNN	196
13.2.3 Faster R-CNN	199
13.3 Ứng dụng object detection	203
13.4 Bài tập	203

14	Bài toán phát hiện biển số xe máy Việt Nam	205
14.1	Lời mở đầu	205
14.2	Chuẩn bị dữ liệu	205
14.2.1	Đánh giá bộ dữ liệu	205
14.2.2	Các phương pháp tăng sự đa dạng của bộ dữ liệu	206
14.2.3	Gán nhãn dữ liệu	208
14.3	Huấn luyện mô hình	209
14.3.1	Giới thiệu về YOLO-Tinyv4 và darknet	209
14.3.2	Cấu hình darknet	211
14.3.3	Huấn luyện model trên colab	213
14.4	Dự đoán	215
15	Image segmentation với U-Net	217
15.1	Bài toán image segmentation	217
15.1.1	Phân loại bài toán image segmentation	218
15.1.2	Ứng dụng bài toán segmentation	219
15.2	Mạng U-Net với bài toán semantic segmentation	220
15.2.1	Kiến trúc mạng U-Net	220
15.2.2	Code	221
15.3	Bài tập	224

VII

Recurrent Neural Network

16	Recurrent neural network	227
16.1	Recurrent Neural Network là gì?	227
16.1.1	Dữ liệu dạng sequence	228
16.1.2	Phân loại bài toán RNN	228
16.1.3	Ứng dụng bài toán RNN	229
16.2	Mô hình bài toán RNN	229
16.2.1	Mô hình RNN	229
16.2.2	Loss function	230
16.2.3	Backpropagation Through Time (BPTT)	231
16.3	Bài tập	232
17	Long short term memory (LSTM)	233
17.1	Giới thiệu về LSTM	233
17.2	Mô hình LSTM	234
17.3	LSTM chống vanishing gradient	235
17.4	Bài tập	236
18	Ứng dụng thêm mô tả cho ảnh	237
18.1	Ứng dụng	237

18.2	Dataset	238
18.3	Phân tích bài toán	238
18.4	Các bước chi tiết	239
18.4.1	Image embedding với Inception	239
18.4.2	Text preprocessing	240
18.4.3	Word embedding	240
18.4.4	Output	246
18.4.5	Model	246
18.5	Python code	247
19	Seq2seq và attention	257
19.1	Giới thiệu	257
19.2	Mô hình seq2seq	258
19.3	Cơ chế attention	260
19.3.1	Motivation	260
19.3.2	Cách hoạt động	260

VIII

Generative Adversarial Networks (GAN)

20	Giới thiệu về GAN	265
20.1	Ứng dụng của GAN	265
20.1.1	Generate Photographs of Human Faces	265
20.1.2	Image editing	266
20.1.3	Generate Anime characters	267
20.1.4	Generate Realistic Photographs	267
20.1.5	Image-to-Image Translation	268
20.1.6	Unsupervised Image-to-image translation	268
20.1.7	Super-resolution	269
20.1.8	Text to image	269
20.1.9	Generate new human pose	270
20.1.10	Music generation	270
20.2	GAN là gì?	270
20.3	Cấu trúc mạng GAN	271
20.3.1	Generator	272
20.3.2	Discriminator	274
20.3.3	Loss function	275
20.4	Code	276
20.5	Bài tập	279
21	Deep Convolutional GAN (DCGAN)	281
21.1	Cấu trúc mạng	281
21.1.1	Generator	281
21.1.2	Discriminator	285

21.2	Tips	285
21.3	Code	286
21.4	Bài tập	290
22	Conditional GAN (cGAN)	291
22.1	Fashion-MNIST	291
22.2	Cấu trúc mạng	292
22.2.1	Generator	292
22.2.2	Discriminator	293
22.2.3	Loss function	294
22.3	Code	294
22.4	Bài tập	298
	Bibliography	299
	Articles	299
	online	299
	Books	301

Giới thiệu

1	Cài đặt môi trường	19
1.1	Giới thiệu	
1.2	Google Colab	
1.3	Hướng dẫn cài đặt anaconda	
2	Python cơ bản	33
2.1	Kiểu dữ liệu cơ bản	
2.2	Containers	
2.3	Function	
2.4	Thư viện trong python	
2.5	Thư viện Numpy	
2.6	Matplotlib	

1. Cài đặt môi trường

1.1 Giới thiệu

Python là ngôn ngữ được sử dụng phổ biến nhất trong Deep Learning, vậy nên tất cả code trong sách sẽ được viết bằng python và thư viện Deep Learning được chọn để sử dụng là [Keras](#). Trong phần này tôi sẽ hướng dẫn cài đặt môi trường. Có 2 dạng là chạy online dùng google colab và cài trên local dùng anaconda và IDE là spyder, VS Code hoặc jupyter notebook. Hiểu đơn giản thì nếu dùng google colab bạn sẽ viết code python và chạy online, không cần cài gì trên máy cả nên sẽ đơn giản hơn và máy cấu hình yếu vẫn chạy được.

1.2 Google Colab

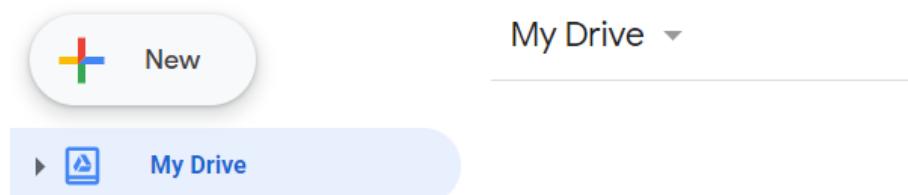
Huấn luyện (hay còn gọi là train) một mô hình Deep Learning, cần xử lý lượng phép tính lớn hơn nhiều so với các mô hình Machine Learning khác. Để cải thiện tốc độ tính toán, người ta dùng GPU (Graphics Processing Unit) thay cho CPU (Central Processing Unit) vì với 1 GPU cho phép xử lý nhiều phép tính song song với rất nhiều core sẽ nhanh hơn nhiều so với CPU. Tuy nhiên giá của GPU thì khá đắt đỏ để mua hoặc thuê server có GPU. Thế nên Google đã cung cấp Google Colab miễn phí có GPU để chạy code python (deep learning) cho mục đích nghiên cứu.

Ở trên môi trường Colab có cài sẵn các thư viện Deep Learning phổ biến như PyTorch, TensorFlow, Keras,... Ngoài ra bạn cũng có thể cài thêm thư viện để chạy nếu cần. Thêm vào đó thì bạn cũng có thể liên kết Colab với google drive và đọc, lưu dữ liệu lên google drive nên rất tiện để sử dụng.

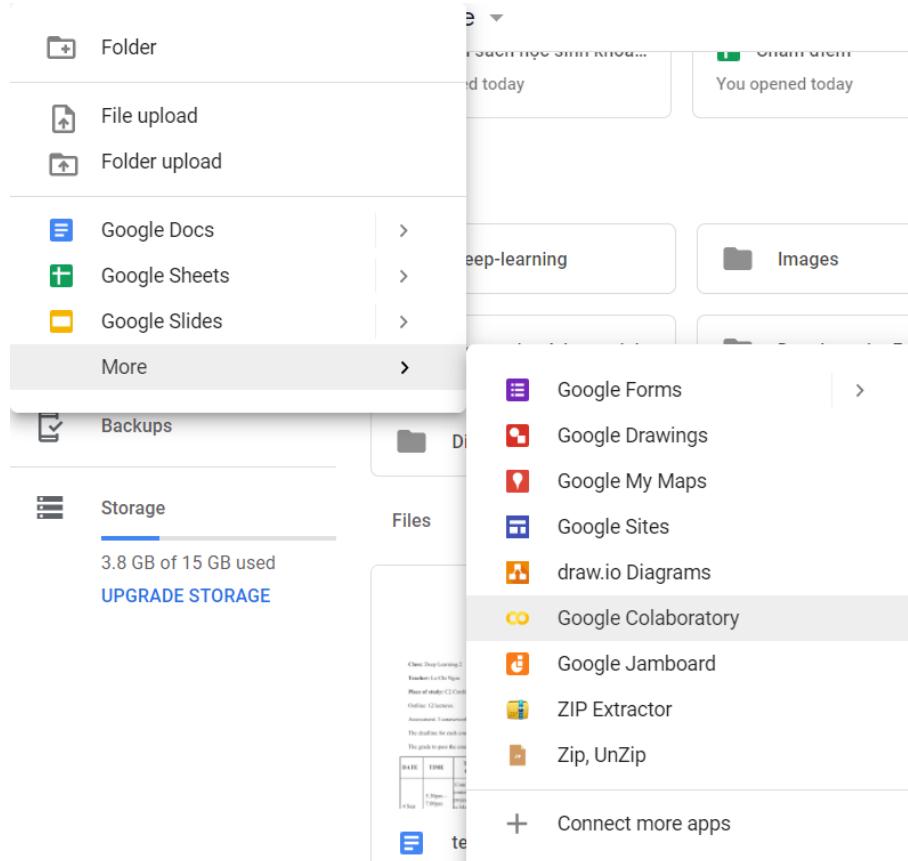
Mặc dù Ở trên Colab chỉ hỗ trợ 2 version Python là 2.7 và 3.6, chưa hỗ trợ ngôn ngữ R và Scala, thì Google Colab vẫn là môi trường tuyệt vời để học và thực hành với deep learning.

1.2.1 Tạo file trên google colab

Đầu tiên bạn vào google drive, tạo folder mà bạn muốn lưu các file colab, rồi chọn nút New

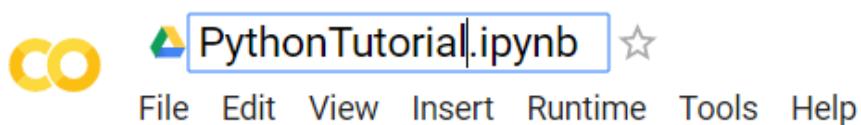


Sau đó kéo xuống chọn Google Colaboratory hoặc bạn có thể truy cập trực tiếp vào [đây](#).



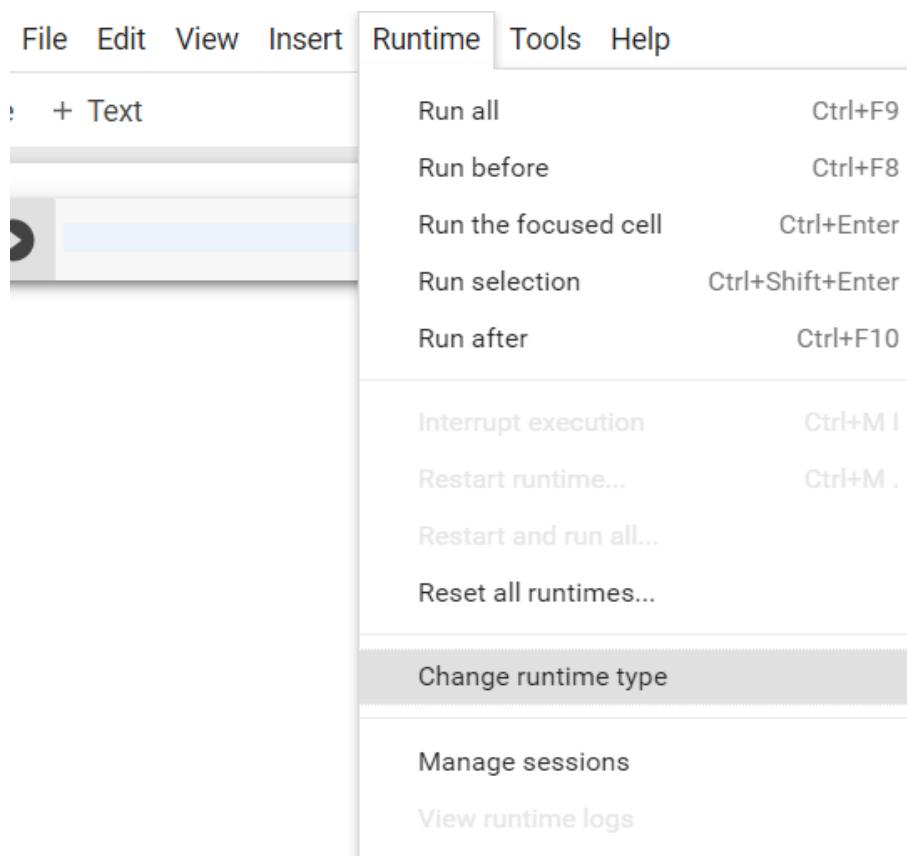
Trong trường hợp bạn không tìm thấy Google Colaboratory, hãy chọn "Connect more apps" ở phía bên dưới, sau đó search Google Colaboratory trên thanh tìm kiếm và cài đặt.

Tiếp đó, bạn click vào phần tên trên cùng của file để đổi tên file cho phù hợp

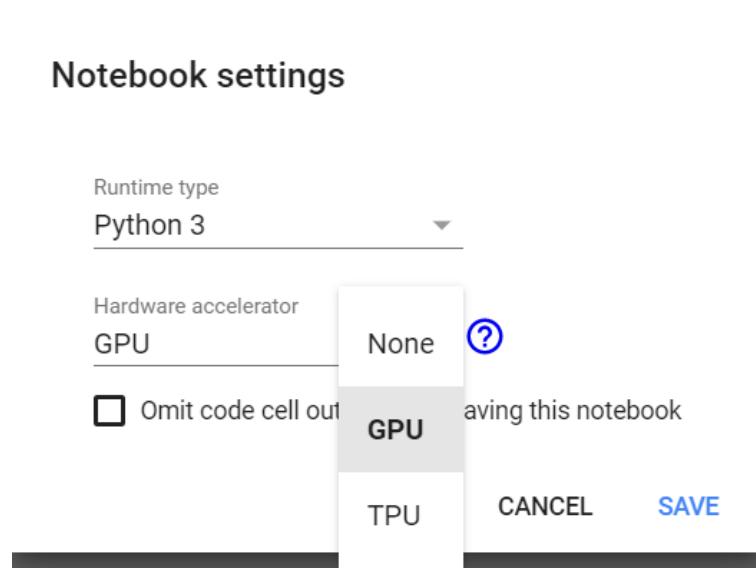


1.2.2 Chọn GPU

Bước này để chọn GPU chạy, bạn chọn Runtime -> Change runtime type



Rồi click vào dấu mũi tên xuống phần Hardware accelerator chọn GPU



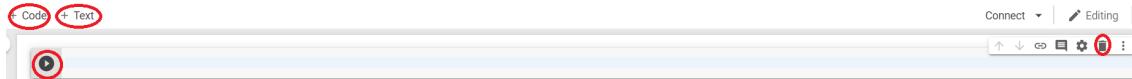
1.2.3 Các thành phần nhỏ

Vì code Python được chia thành từng khối (block) để chạy, bạn nhìn hình bên dưới có:

- nút +Code để thêm 1 block code python
- nút +Text để thêm 1 khối text (giống như comment nhưng có thể format được màu style hơn)
- biểu tượng hình thùng rác để xóa khối code/text đi
- nút mũi tên xoay ngang để chạy khối code/text đây. Bên cạnh đó các bạn cũng có thể chạy

khởi code bằng các phím tắt cho nhanh hơn:

- Cmd/Ctrl + Enter (Cmd cho Mac, Ctrl cho Win): chạy khối code/text.
- Shift + Enter: Chạy và chuyển đổi tương ứng xuống khối code bên dưới.
- Alt + Enter: Chạy và thêm một khối code bên dưới.



1.2.4 Link với google drive

Đoạn code để link tới các file trên Google drive

```
from google.colab import drive
drive.mount('/content/gdrive')
```



Sau khi ấn chạy đoạn code đấy, bạn click vào link trên, chọn tài khoản google bạn đang dùng, rồi chọn accept bạn sẽ có mã code như ở dưới.

Please copy this code, switch to your application and paste it there:

4/rQG5p8lsYJYU08see-
G93vTrxSjU4DxGcPU9Imn54hR0eNSFUHRYwa0

Rồi bạn copy mã code đấy vào ô trống ở trong phần chạy của block hiện tại rồi ấn enter



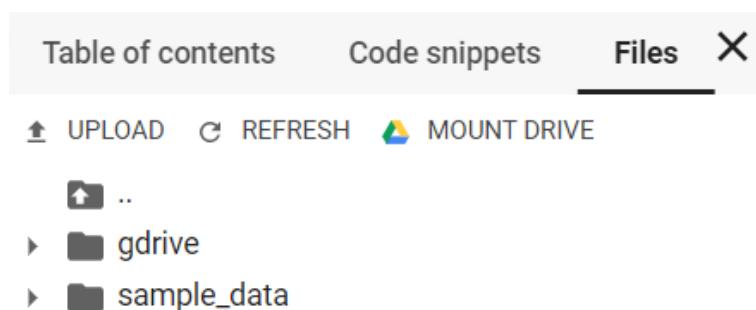
Sau khi link (mount) thành công bạn sẽ thấy dòng chữ **Mounted at /content/gdrive**



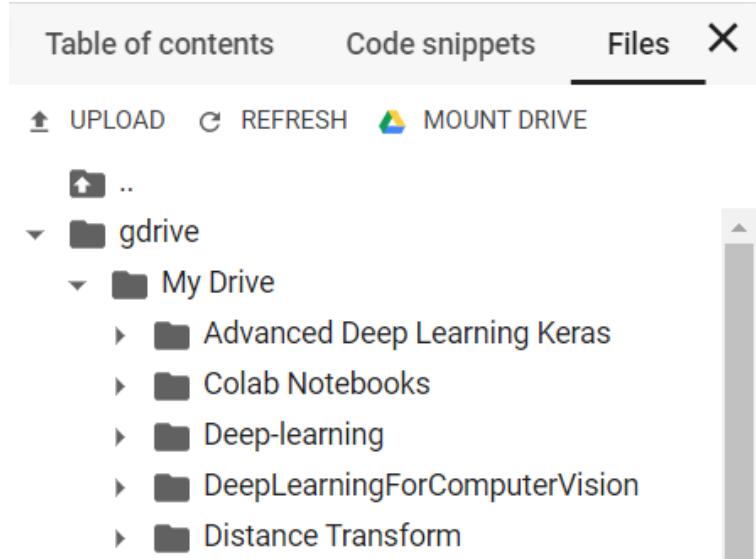
Sau đó bạn click vào nút mũi tên



Rồi chọn tab Files để nhìn thấy các Files, cách tổ chức file và thư mục dạng cây giống như trong window.



Các file và thư mục trong google drive được lưu ở gdrive/My Drive



Để chuyển thư mục hiện tại đến thư mục khác bạn dùng lệnh

```
\%cd '/content/gdrive/My Drive/Deep-learning'
!ls
```



1.2.5 Cài thêm thư viện

Để cài thêm thư viện bạn dùng cú pháp `!pip install thư viện`, ví dụ lệnh dưới để cài thư viện `scipy`

```
!pip install scipy
```



1.3 Hướng dẫn cài đặt anaconda

1.3.1 Giới thiệu

Anaconda là nền tảng mã nguồn mở về Khoa học dữ liệu trên Python thông dụng nhất hiện nay. Với hơn 11 triệu người dùng, Anaconda là cách nhanh nhất và dễ nhất để học Khoa học dữ liệu với Python hoặc R trên Windows, Linux và Mac OS X. Lợi ích của Anaconda:

- Dễ dàng tải 1500+ packages về Python/R cho data science
- Quản lý thư viện, môi trường và dependency giữa các thư viện dễ dàng
- Dễ dàng phát triển mô hình machine learning và deep learning với scikit-learn, tensorflow, keras

- Xử lý dữ liệu tốc độ cao với numpy, pandas
- Hiển thị kết quả với Matplotlib, Bokeh

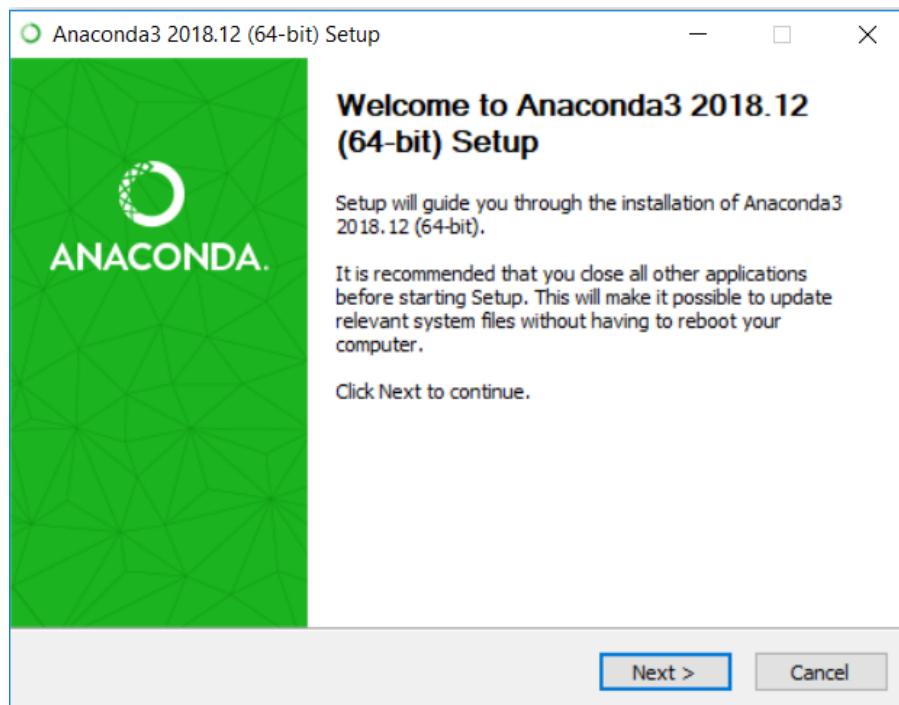
Trong khi đó Spyder là 1 trong những IDE (môi trường tích hợp dùng để phát triển phần mềm) tốt nhất cho data science và quang trọng hơn là nó được cài đặt khi bạn cài đặt Anaconda.

1.3.2 Yêu cầu phần cứng và phần mềm

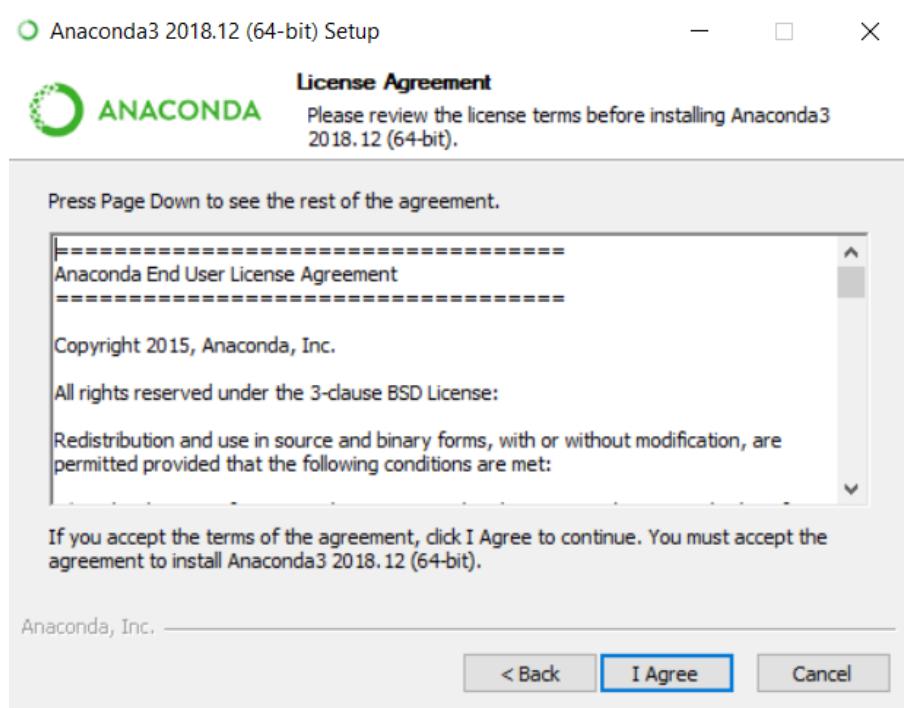
- Hệ điều hành: Win 7, Win 8/8.1, Win 10, Red Hat Enterprise Linux/CentOS 6.7, 7.3, 7.4, and 7.5, and Ubuntu 12.04+.
- Ram tối thiểu 4GB.
- Ổ cứng trống tối thiểu 3GB để tải và cài đặt.

1.3.3 Cài đặt

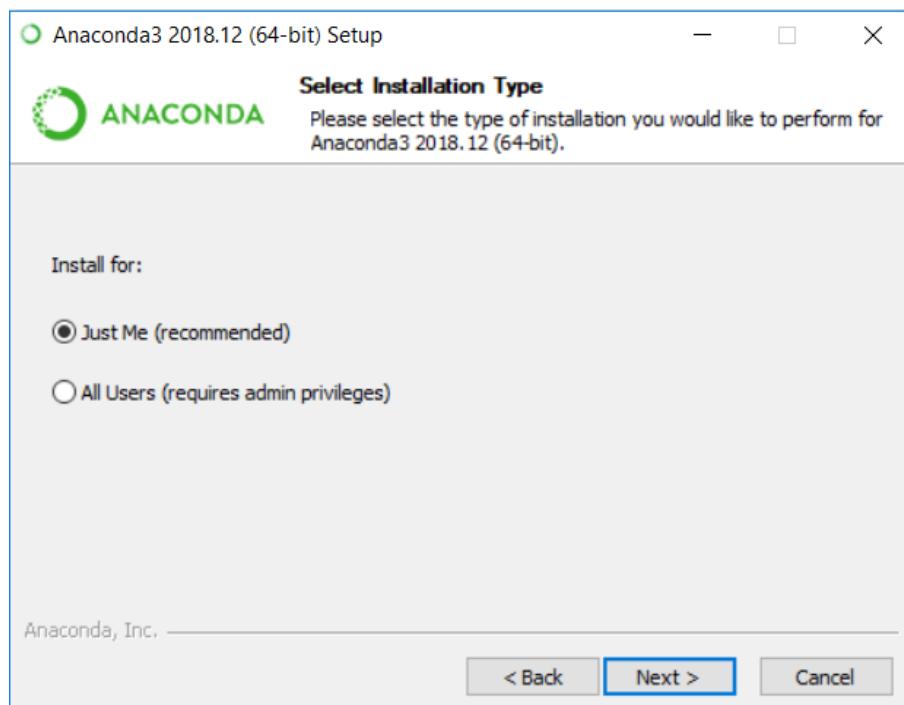
Bạn click vào [đây](#). Sau khi tải xong bạn mở file



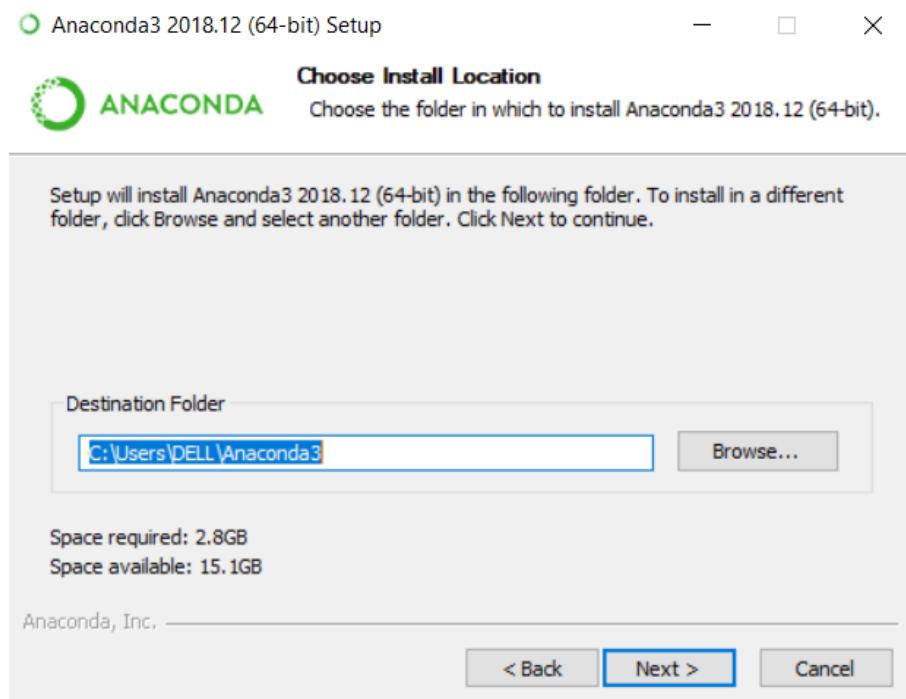
Hình 1.1: Click Next



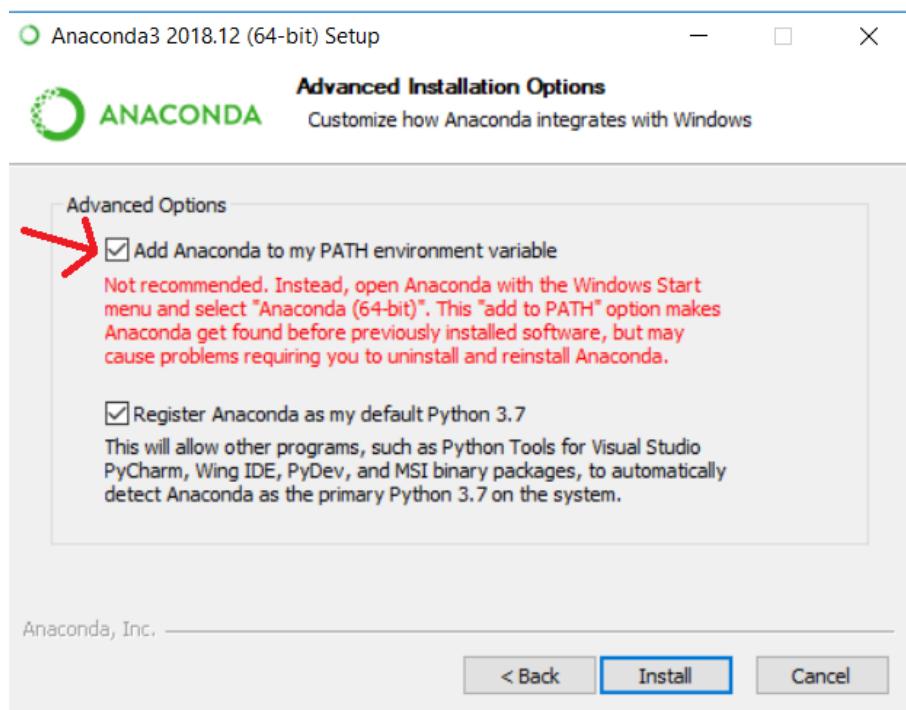
Hình 1.2: Click I agree



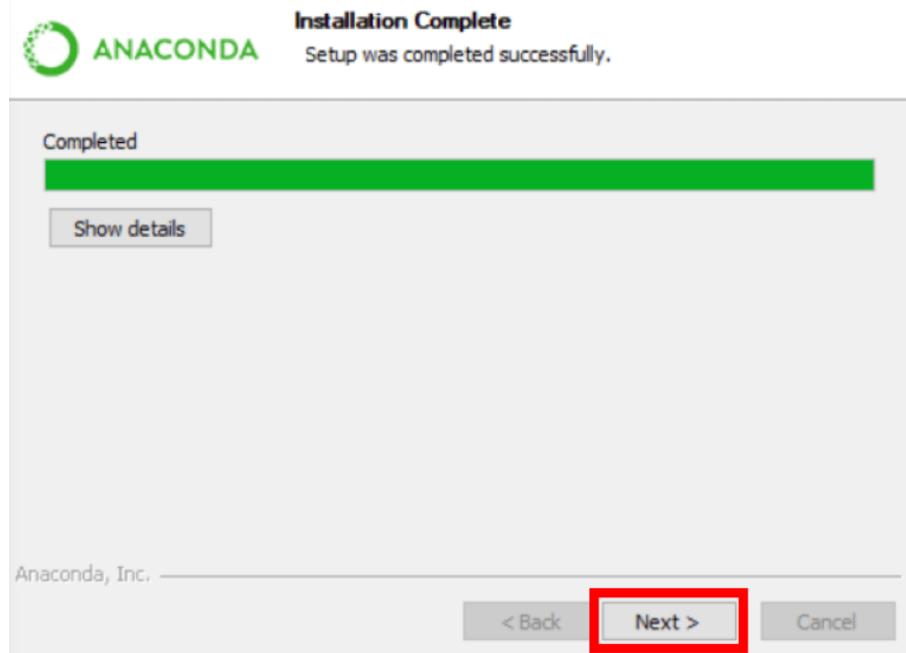
Hình 1.3: Click Next



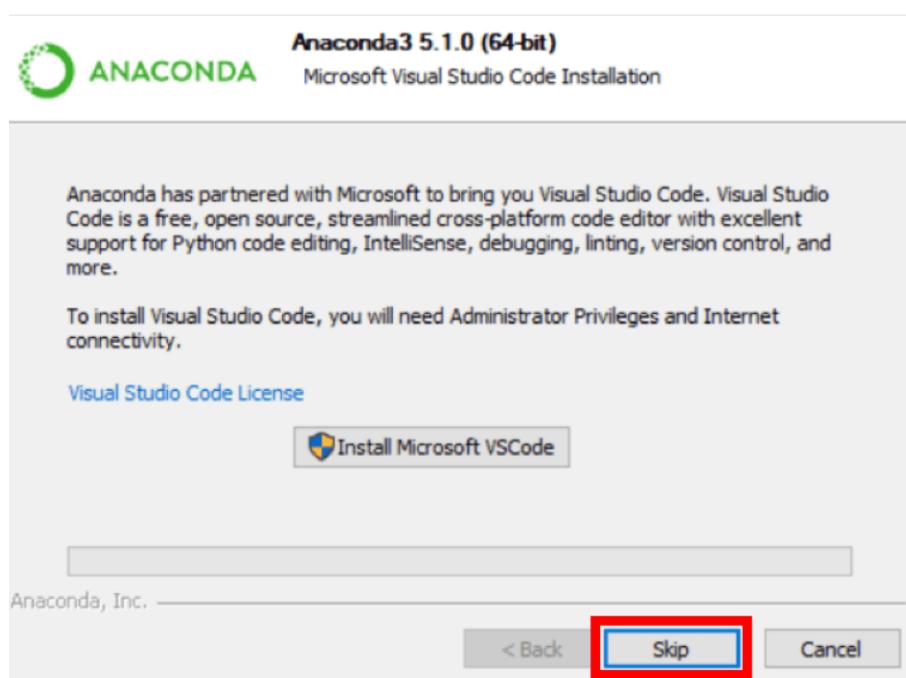
Hình 1.4: Bạn có thể chọn như mục cài đặt khác bằng việc click **Browse...** và chọn, xong thì click **Next**



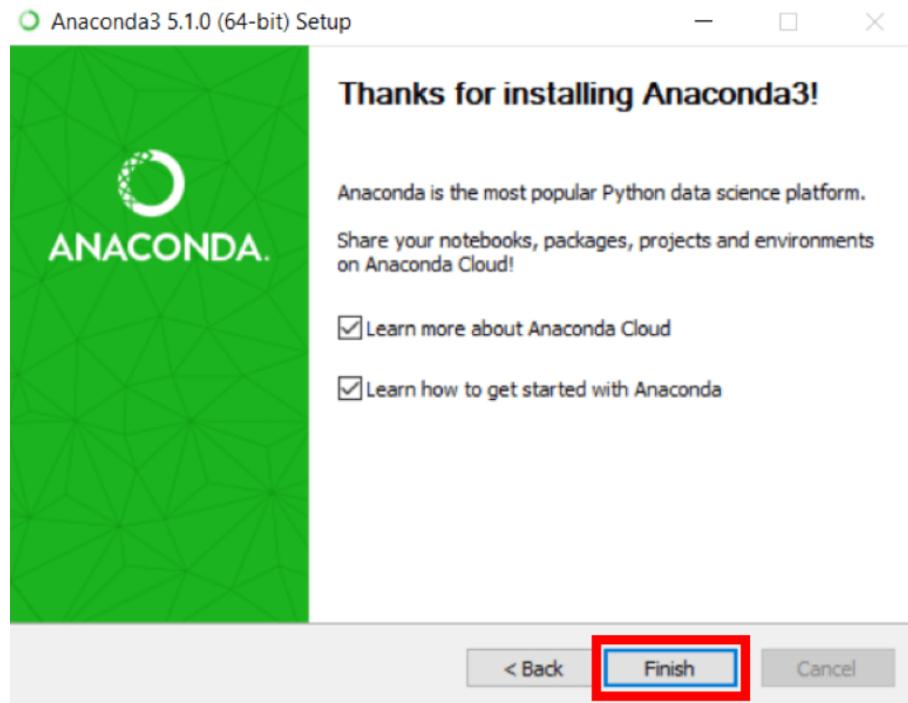
Hình 1.5: Bạn nhớ click vào ô vuông gần dòng **Add Anaconda to my PATH environment variable**



Hình 1.6: Click next



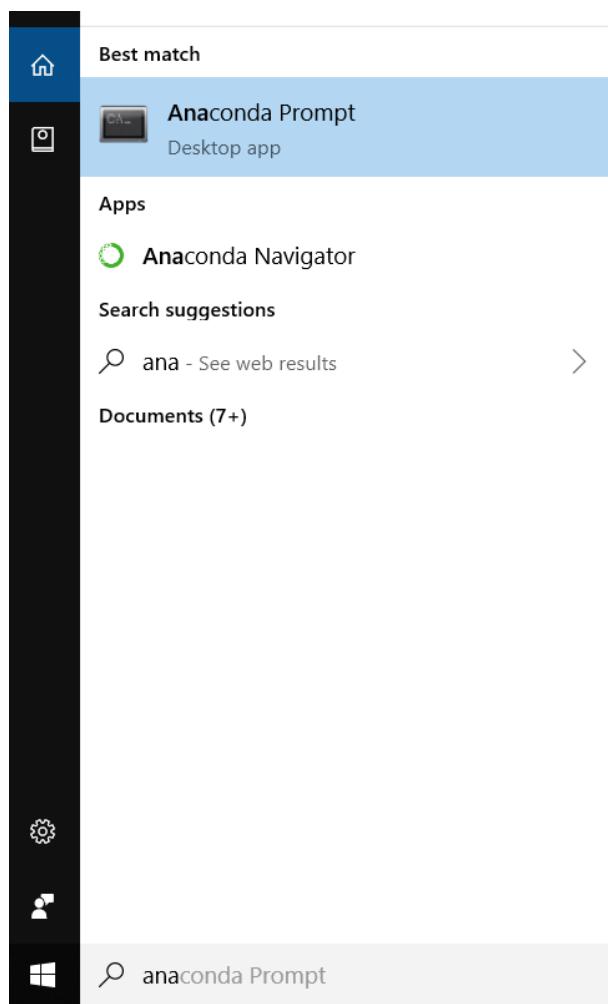
Hình 1.7: Click Skip



Hình 1.8: Click Finish.

Cài đặt một số library cơ bản

Mở anaconda command line và copy lệnh sau để cài library.



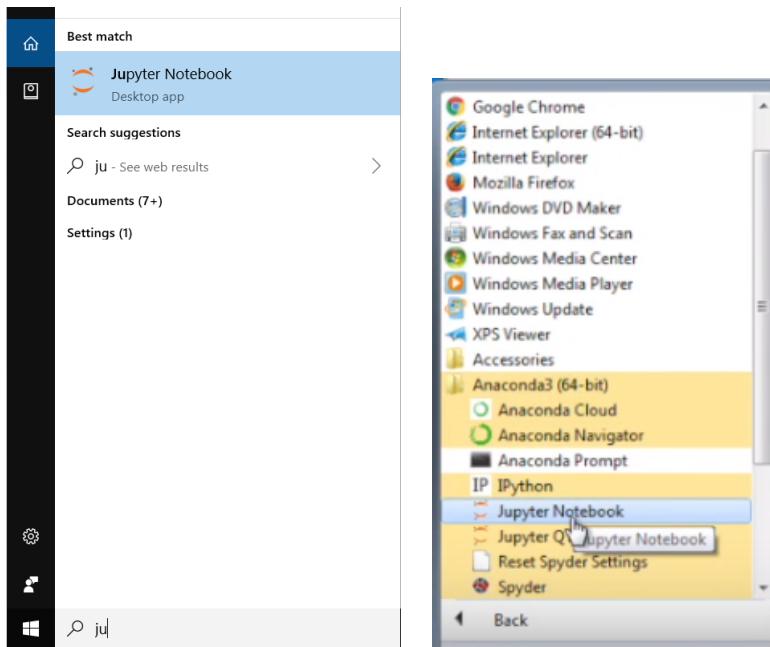
Một số thư viện cơ bản

- **Numpy:** conda install -c anaconda numpy
- **Pandas:** conda install -c anaconda pandas
- **Matplotlib:** conda install -c conda-forge matplotlib
- **Keras:** conda install -c conda-forge keras

Hoặc bạn cũng có thể cài bằng pip trên anaconda prompt

- **Numpy:** pip install numpy
- **Pandas:** pip install pandas
- **Matplotlib:** pip install matplotlib
- **Keras:** pip install Keras

1.3.4 Hướng dẫn sử dụng Jupyter notebook



Hình 1.9: Bên trái: Mở jupyter notebook trên win10, bên phải: Mở trên win7

Giao diện mở ra, click vào new chọn Python3 để tạo 1 notebook mới



Sau đây thì dùng giống như google colab, nhưng điểm khác biệt là jupyter notebook sẽ chạy trên máy của bạn chứ không phải cloud như google colab.

2. Python cơ bản

Phần này sẽ giới thiệu Python cơ bản cho những người mới chưa học lập trình, hoặc từ ngôn ngữ khác chuyển qua dùng Python. Nội dung được tham khảo từ [đây](#)

2.1 Kiểu dữ liệu cơ bản

Giống với các ngôn ngữ lập trình khác, Python cũng có các kiểu dữ liệu cơ bản như integers (số nguyên), floats (số thực), booleans (kiểu dữ liệu True/False) và strings (chuỗi).

Để khai báo biến trong Python không cần chỉ định kiểu dữ liệu trước như những ngôn ngữ khác mà khi ta gán dữ liệu cho biến thì python tự gán kiểu phù hợp cho biến.

Cú pháp để khai báo biến:

```
# variable_name = value  
x = 3
```

Đặt tên biến trong python có một vài quy tắc:

- Tên biến nên có nghĩa. Tên biến do mọi người tự đặt nhưng nên đặt theo ý nghĩa, tác dụng của biến đầy đủ trong chương trình. Thứ nhất là khi mọi người code dài, muốn ở dưới dùng lại thì cũng dễ nhớ. Thứ hai là khi người khác đọc code của bạn sẽ dễ hiểu hơn.
- Tên biến phải bắt đầu với chữ cái hoặc dấu gạch dưới (_).
- Tên biến không được bắt đầu với số.
- Tên biến chỉ được chứa các chữ cái, các số và dấu gạch dưới (A-Z, a-z, 0-9, _)
- Tên biến có phân biệt chữ hoa chữ thường. Ví dụ age, AGE, Age, aGE là 4 tên biến khác nhau

2.1.1 Số

Số nguyên và số thực dùng như các ngôn ngữ khác

```
x = 3  
print(type(x)) # Prints "<class 'int'>"
```

```
print(x)          # Prints "3"
print(x + 1)      # Cộng; prints "4"
print(x - 1)      # Trừ; prints "2"
print(x * 2)      # Nhân; prints "6"
print(x ** 2)     # Lũy thừa; prints "9"
x += 1            # Giống với x = x + 1
print(x)
x *= 2            # Giống x = x * 2
print(x)
y = 2.5
print(type(y))    # Prints <class 'float'>
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

2.1.2 Phép tính logic

Python có kiểu dữ liệu boolean, các biến kiểu này nhận 2 giá trị là True hoặc False. Python có các phép tính logic nhưng dùng các từ tiếng anh (and, or) thay cho kí hiệu (&&, ||):

```
t = True
f = False

print(type(t)) # Prints <class 'bool'>
print(t and f) # AND; prints "False"
print(t or f) # OR; prints "True"
print(not t) # NOT; prints "False"
print(t != f) # XOR; prints "True"
```

2.1.3 Chuỗi

Python có hỗ trợ dạng chuỗi, để lưu giá trị dạng chuỗi có thể để trong dấu " hoặc ' nhưng mở bằng dấu nào phải đóng bằng dấu đó.

```
# greet = "hello"          # câu lệnh này lõi
hello = 'hello'            # gán giá trị chuỗi cho biến, chuỗi đặt trong 2 dấu '
world = "world"           # chuỗi cũng có thể đặt trong dấu ".
print(hello)               # Prints "hello"
print(len(hello))          # Độ dài chuỗi; prints "5"
hw = hello + ' ' + world  # Nối chuỗi bằng dấu +
print(hw)                  # prints "hello world"
hw12 = '{} {} {}'.format(hello, world, 12) # Cách format chuỗi
print(hw12)                # prints "hello world 12"
```

Kiểu string có rất nhiều method để xử lý chuỗi.

2.2 Containers

Các kiểu dữ liệu cơ bản chỉ chứa một giá trị mỗi biến (số, chuỗi), tuy nhiên nhiều lúc mình cần chứa nhiều giá trị, ví dụ chứa tên học sinh trong một lớp có 100 bạn. Mình không thể tạo 100 biến để lưu tên 100 bạn. Vậy nên cần các kiểu dữ liệu có thể chứa nhiều giá trị khác nhau. Đó là container (collection). Python có một số container như: list, tuple, set, dictionary. Dưới tôi sẽ trình bày hai kiểu dữ liệu collection mà mọi người hay gặp nhất trong python là list và dictionary.

2.2.1 List

List trong Python giống như mảng (array) nhưng không cố định kích thước và có thể chứa nhiều kiểu dữ liệu khác nhau trong 1 list.

```
xs = [3, 1, 2]      # Tao 1 list
print(xs, xs[2])   # Prints "[3, 1, 2] 2"
print(xs[-1])       # Chỉ số âm là đếm phần tử từ cuối list lên; prints "2"
xs[2] = 'foo'        # List có thể chứa nhiều kiểu phần tử khác nhau
print(xs)            # Prints "[3, 1, 'foo']"
xs.append('bar')     # Thêm phần tử vào cuối list
print(xs)            # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()         # Bỏ đi phần tử cuối cùng khỏi list và trả về phần tử đầy
print(x, xs)         # Prints "bar [3, 1, 'foo']"
```

Slicing Thay vì lấy từng phần tử một trong list thì python hỗ trợ truy xuất nhiều phần tử 1 lúc gọi là slicing.

```
nums = list(range(5))      # range sinh ra 1 list các phần tử
print(nums)                 # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])            # Lấy phần tử thứ 2->4, python chỉ số mảng từ 0;
print(nums[2:])             # Lấy từ phần tử thứ 2 đến hết; prints "[2, 3, 4]"
print(nums[:2])             # Lấy từ đầu đến phần tử thứ 2; prints "[0, 1]"
print(nums[:])              # Lấy tất cả phần tử trong list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])            # Lấy từ phần tử đầu đến phần tử gần cuối trong list;
                           # prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]          # Gán giá trị mới cho phần tử trong mảng từ vị trí 2->4
print(nums)                 # Prints "[0, 1, 8, 9, 4]"
```

Loops Để duyệt và in ra các phần tử trong list

```
animals = ['cat', 'dog', 'monkey']
# duyệt giá trị không cần chỉ số
for animal in animals:
    print('%s' % (animal))
# duyệt giá trị kèm chỉ số dùng enumerate
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", in mỗi thành phần trong list 1 dòng
```

2.2.2 Dictionaries

Dictionaries lưu thông tin dưới dạng key, value.

```

d = {'cat': 'cute', 'dog': 'furry'} # Tạo dictionary, các phần tử dạng key:value
print(d['cat'])                  # Lấy ra value của key 'cat' trong dictionary prints "cute"
print('cat' in d)                # Kiểm tra key có trong dictionary không; prints "True"
d['fish'] = 'wet'                 # Gán key, value, d[key] = value
print(d['fish'])                 # Prints "wet"
# print(d['monkey'])            # Lỗi vì key 'monkey' không trong dictionary
del d['fish']                   # Xóa phần tử key:value từ dictionary

```

Loop Duyệt qua các phần tử trong dictionary

```

d = {'person': 2, 'cat': 4, 'spider': 8}
# Duyệt key
for animal in d:
    print('A %s has %d legs' % (animal, d[animal]))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"

# Duyệt value
for legs in d.values():
    print('%d legs' % (legs))
# Prints "2 legs", "4 legs", "8 legs"

# Duyệt cả key và value
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"

```

2.3 Function

Function (hàm) là một khối code python được thực hiện một hoặc một số chức năng nhất định. Hàm có input và có output, trước khi viết hàm mọi người nên xác định hàm này để làm gì? input là gì? output làm gì?

Ví dụ hàm kiểm tra số nguyên tố, mục đích để kiểm tra 1 số xem có phải là số nguyên tố hay không, input là 1 số nguyên dương, output dạng boolean (True/False). True tức là số input là số nguyên tố, False nghĩa là không phải số nguyên tố.

Function trong python được định nghĩa với keyword **def**.

```

# Hàm có input là 1 số và output xem số đó âm, dương hay số 0
# Định nghĩa hàm
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    # Gọi hàm

```

```
print(sign(x))
# Prints "negative", "zero", "positive"
```

2.4 Thư viện trong python

Thư viện (library) bao gồm các hàm (function) hay lớp (class) được viết sẵn với các chức năng khác nhau. Ví dụ thư viện math cung cấp các hàm về tính toán như exp, sqrt, floor,...

Muốn nhập thư viện vào chương trình, ta dùng cú pháp: import *tên thư viện*, ví dụ: import numpy.

Đối với những thư viện có tên dài, ta thường rút ngắn lại để sau này dễ sử dụng. Khi đó, ta sử dụng cú pháp: import *tên thư viện* as *tên rút gọn*, ví dụ: import matplotlib.pyplot as plt. Sau này, khi sử dụng đến thư viện, ta chỉ cần gọi đến tên rút gọn thay vì phải gõ lại tên đầy đủ của thư viện, ví dụ thay vì viết matplotlib.pyplot.plot, ta chỉ cần viết plt.plot.

2.5 Thư viện Numpy

Vì Python là scripting language nên không thích hợp cho machine learning, numpy giải quyết vấn đề trên bằng cách xây dựng 1 thư viện viết bằng C nhưng có interface Python. Như vậy Numpy cộng hưởng 2 ưu điểm của 2 ngôn ngữ: nhanh như C và đơn giản như Python. Điều này giúp ích rất nhiều cho cộng đồng Machine Learning trên Python.

Mảng trong numpy gồm các phần tử có dùng kiểu giá trị, chỉ số không âm được bắt đầu từ 0, số chiều được gọi là rank của mảng Numpy, và shape của mảng là một tuple các số nguyên đưa ra kích thước của mảng theo mỗi chiều.

```
import numpy as np

a = np.array([1, 2, 3])      # Tao array 1 chiều
print(type(a))              # Prints "<class 'numpy.ndarray'>"
print(a.shape)               # Prints "(3,)"
print(a[0], a[1], a[2])     # Prints "1 2 3"
a[0] = 5                    # Thay doi phan tu vi tri so 0
print(a)                     # Prints "[5, 2, 3]

b = np.array([[1,2,3],[4,5,6]])    # Tao array 2 chiều
print(b.shape)                # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])  # Prints "1 2 4"
```

Ngoài ra có những cách khác để tạo array với giá trị mặc định

```
import numpy as np

a = np.zeros((2,2))      # Tao array voi tat ca cac phan tu 0
print(a)                  # Prints "[[ 0.  0.]
                           #           [ 0.  0.]]"

b = np.ones((1,2))        # Tao array voi cac phan tu 1
print(b)                  # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)    # Tao array voi cac phan tu 7
```

```

print(c)          # Prints "[[ 7.  7.]
#                  [ 7.  7.]]"

d = np.eye(2)    # Tao identity matrix kích thước 2*2
print(d)          # Prints "[[ 1.  0.]
#                  [ 0.  1.]]"

e = np.random.random((2,2)) # Tao array với các phần tử được tạo ngẫu nhiên
print(e)          # Might print "[[ 0.91940167  0.08143941]
#                           [ 0.68744134  0.87236687]]"

```

2.5.1 Array indexing

Tương tự như list, numpy array cũng có thể slice. Tuy nhiên vì numpy array có nhiều chiều, nên khi dùng slice phải chỉ định rõ chiều nào.

```

import numpy as np

# Tao array 2 chiều với kích thước (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Dùng slide để lấy ra subarray gồm 2 hàng đầu tiên (1 & 2) và 2 cột (2 & 3).
# Output là array kích thước 2*2
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

print(a[0, 1])  # Prints "2"
a[0, 1] = 77    # Chính sửa phần tử trong array
print(a[0, 1])  # Prints "77"

```

Bên cạnh đó cũng có thể dùng các chỉ số với slice index. Tuy nhiên số chiều array sẽ giảm đi.

```

import numpy as np

# Tao array 2 chiều kích thước (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

row_r1 = a[1, :]    # Lấy ra hàng thứ 2 trong a, output array 1 chiều
row_r2 = a[1:2, :]  # Lấy ra hàng thứ 1&2 trong a, output array 2 chiều
print(row_r1, row_r1.shape) # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape) # Prints "[[5 6 7 8]] (1, 4)"

```

2.5.2 Các phép tính trên array

Các phép tính với ma trận được hỗ trợ trên numpy

```

import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Tính tổng
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Phép trừ
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Phép nhân element-wise
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Phép chia element-wise
# [[ 0.2          0.33333333]
#  [ 0.42857143  0.5         ]]
print(x / y)
print(np.divide(x, y))

# Tính căn bậc hai
# [[ 1.          1.41421356]
#  [ 1.73205081  2.          ]]
print(np.sqrt(x))

```

Phép toán * dùng để nhân element-wise chứ không phải nhân ma trận thông thường. Thay vào đó dùng np.dot để nhân 2 ma trận.

```

import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Nhân ma trận, output số 219
print(v.dot(w))
print(np.dot(v, w))

# Nhân ma trận; output ma trận 2*2

```

```
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

Numpy cũng hỗ trợ tính tổng array theo các chiều khác nhau

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x)) # Tính tổng tất cả phần tử trong array; prints "10"
print(np.sum(x, axis=0)) # Tính tổng phần tử mỗi hàng; prints "[4 6]"
print(np.sum(x, axis=1)) # Tính tổng phần tử mỗi cột; prints "[3 7]"
```

2.5.3 Broadcasting

Broadcasting là một kĩ thuật cho phép numpy làm việc với các array có shape khác nhau khi thực hiện các phép toán.

```
import numpy as np

# Cộng vector v với mỗi hàng của ma trận x, kết quả lưu ở ma trận y.
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x) # Tạo 1 array có chiều giống x

# Dùng loop để vector v với mỗi hàng của ma trận
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
# Kết quả của y
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]

# Cộng kiểu broadcasting trong python
print(x + v)
# Kết quả
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]

# 2 kết quả cho ra giống nhau
```

2.6 Matplotlib

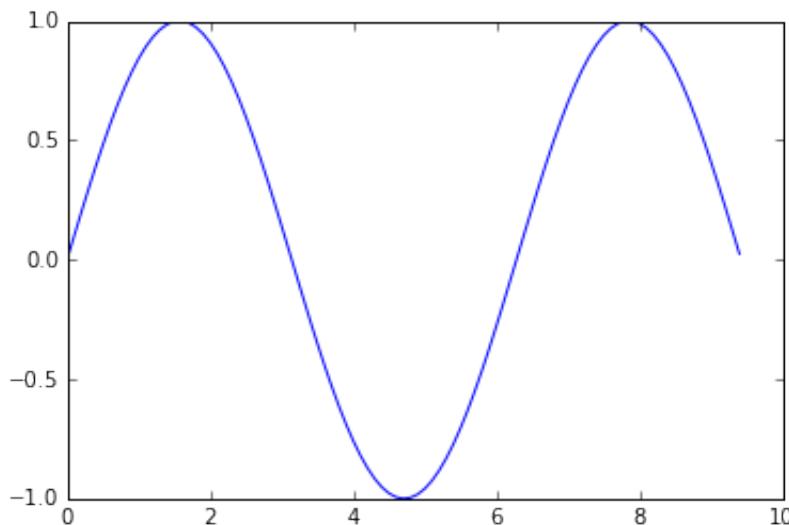
Thư viện matplotlib dùng để vẽ đồ thị. Phần này tôi đi qua các tính năng cơ bản của module matplotlib.pyplot

Hàm plot để vẽ dữ liệu 2d trong python

```
import numpy as np
import matplotlib.pyplot as plt

# Tính tọa độ x, y cho đồ thị hình sin
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
plt.show() # You must call plt.show() to make graphics appear.
```

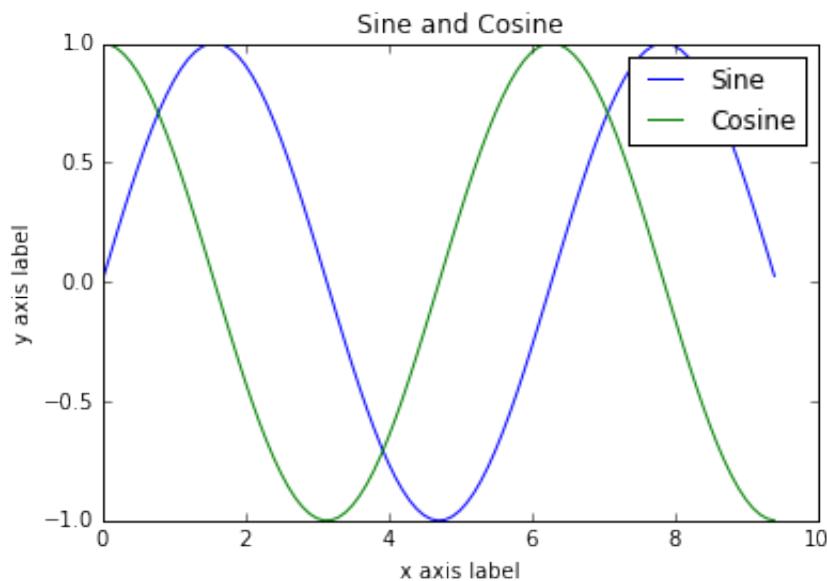


Ta có thể vẽ nhiều đường trên 1 đồ thị cũng như thêm tên đồ thị, tên trục x, y cũng như tên các đường tương ứng lên đồ thị.

```
import numpy as np
import matplotlib.pyplot as plt

# Tính tọa độ x, y cho đồ thị sin và cos
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Vẽ đồ thị sin, cos dùng matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



Bên cạnh đấy ta có thể vẽ nhiều subplot và mỗi đồ thị trên 1 subplot riêng.

```

import numpy as np
import matplotlib.pyplot as plt

# Tính tọa độ x, y cho đồ thị sin, cos
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Trên grid kích thước 2*1, vẽ trên grid 1
plt.subplot(2, 1, 1)

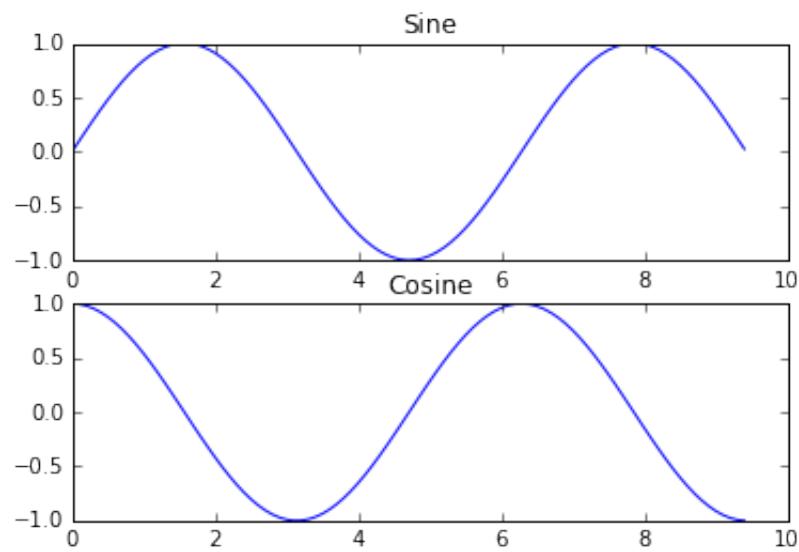
plt.plot(x, y_sin)
plt.title('Sine')

# Trên grid kích thước 2*1, vẽ trên grid 2
plt.subplot(2, 1, 2)

plt.plot(x, y_cos)
plt.title('Cosine')

plt.show()

```



Machine learning cơ bản

3 Linear regression 47

- 3.1 Bài toán
- 3.2 Thiết lập công thức
- 3.3 Thuật toán gradient descent
- 3.4 Ma trận
- 3.5 So sánh các loss function
- 3.6 Giải bằng đại số tuyến tính
- 3.7 Python code
- 3.8 Bài tập

4 Logistic regression 65

- 4.1 Bài toán
- 4.2 Xác suất
- 4.3 Hàm sigmoid
- 4.4 Thiết lập bài toán
- 4.5 Chain rule
- 4.6 Quan hệ giữa phần trăm và đường thẳng
- 4.7 Ứng dụng
- 4.8 Python code
- 4.9 Bài tập

3. Linear regression

Thuật toán linear regression giải quyết các bài toán có đầu ra là giá trị thực, ví dụ: dự đoán giá nhà, dự đoán giá cổ phiếu, dự đoán tuổi,...

3.1 Bài toán

Bạn làm ở công ty bất động sản, bạn có dữ liệu về diện tích và giá nhà, giờ có một ngôi nhà mới bạn muốn ước tính xem giá ngôi nhà đó khoảng bao nhiêu. Trên thực tế thì giá nhà phụ thuộc rất nhiều yếu tố: diện tích, số phòng, gần trung tâm thương mại,... nhưng để cho bài toán đơn giản giả sử giá nhà chỉ phụ thuộc vào diện tích căn nhà. Bạn có dữ liệu về diện tích và giá bán của 30 căn nhà như sau:

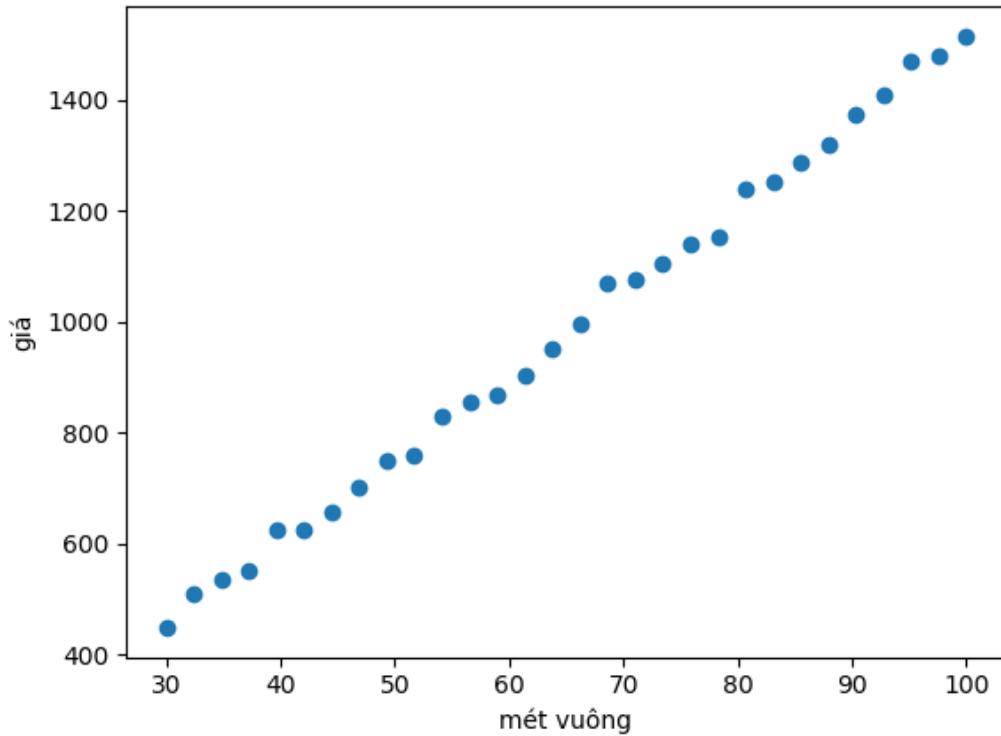
Diện tích(m ²)	Giá bán (triệu VNĐ)
30	448.524
32.4138	509.248
34.8276	535.104
37.2414	551.432
39.6552	623.418
...	...

Khi có dữ liệu mình sẽ visualize dữ liệu lên hình 3.1

Nếu giờ yêu cầu bạn ước lượng nhà 50 mét vuông khoảng bao nhiêu tiền thì bạn sẽ làm thế nào? Vẽ một đường thẳng gần với các điểm trên nhất và tính giá nhà ở điểm 50 như ở hình 3.2

Về mặt lập trình cũng cần làm 2 việc như vậy:

1. Training: Tìm đường thẳng (model) gần các điểm trên nhất. Mọi người có thể vẽ ngay được đường thẳng mô tả dữ liệu từ hình 1, nhưng máy tính thì không, nó phải đi tìm bằng thuật toán Gradient descent ở phía dưới. (Từ model và đường thẳng được dùng thay thế lẫn nhau trong phần còn lại của bài này).
2. Prediction: Dự đoán xem giá của ngôi nhà 50 m² có giá bao nhiêu dựa trên đường tìm được ở phần trên.



Hình 3.1: Đồ thị quan hệ giữa diện tích và giá nhà.

3.2 Thiết lập công thức

3.2.1 Model

Phương trình đường thẳng có dạng $y = ax + b$ ví dụ hình 3.3. Thay vì dùng kí hiệu a, b cho phương trình đường thẳng; để tiện cho biểu diễn ma trận phần sau ta sẽ thay $w_1 = a, w_0 = b$

Nên phương trình được viết lại thành: $y = w_1 * x + w_0 \Rightarrow$ Việc tìm đường thẳng giờ thành việc tìm w_0, w_1 .

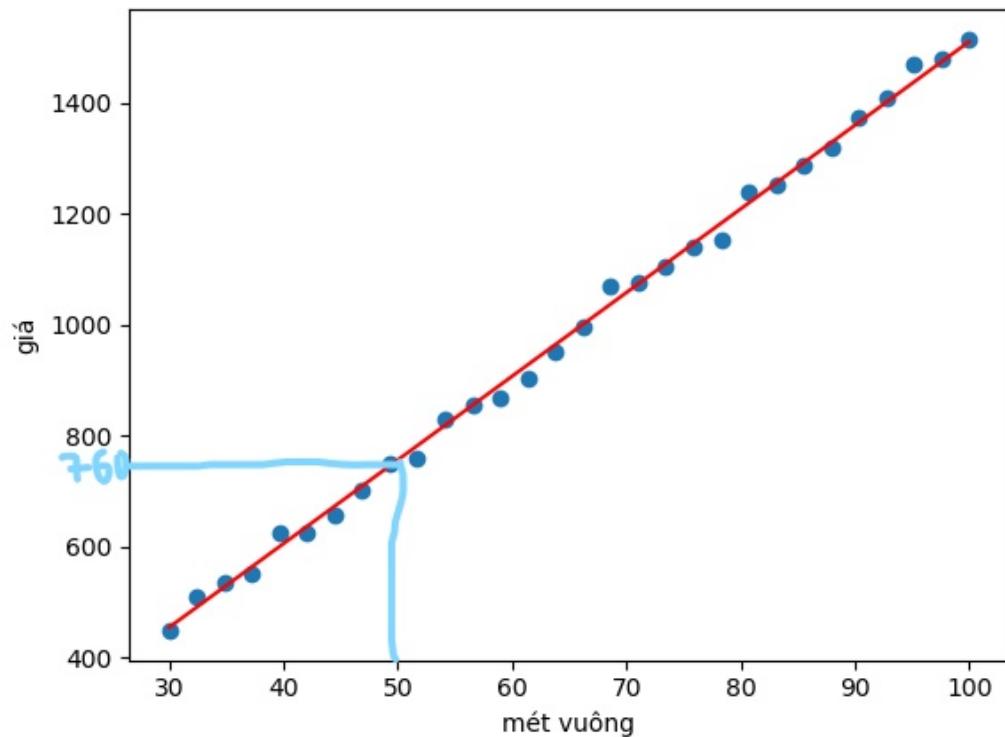
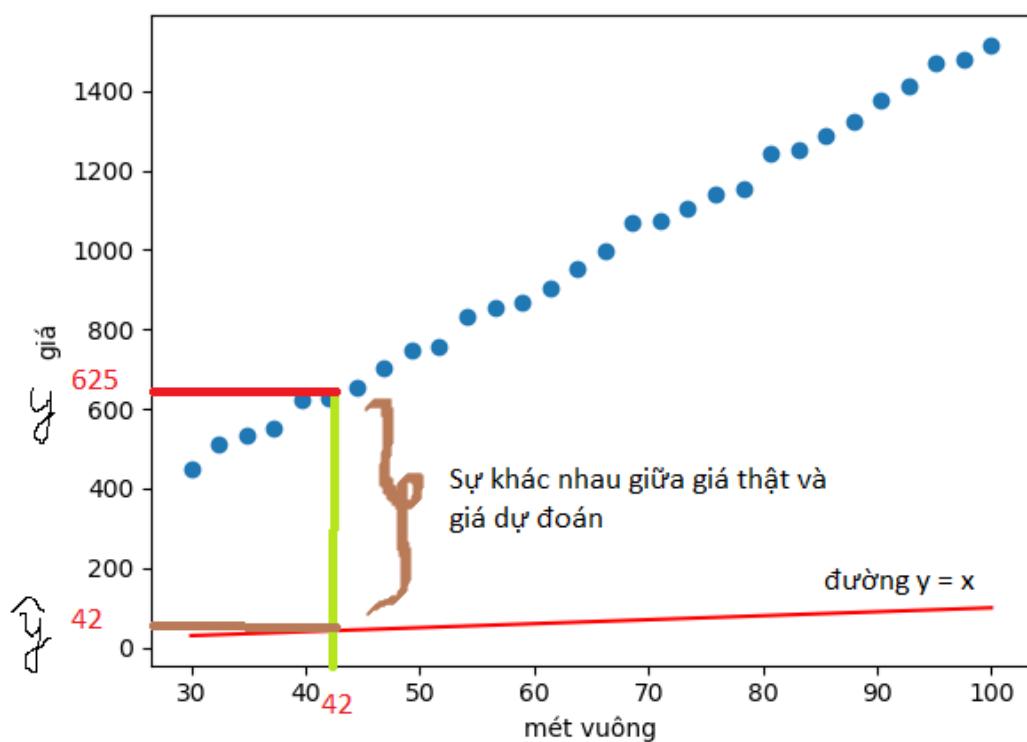
Để tiện cho việc thiết lập công thức, ta sẽ đặt ký hiệu cho dữ liệu ở bảng dữ liệu: $(x_1, y_1) = (30, 448.524), (x_2, y_2) = (32.4138, 509.248), \dots$

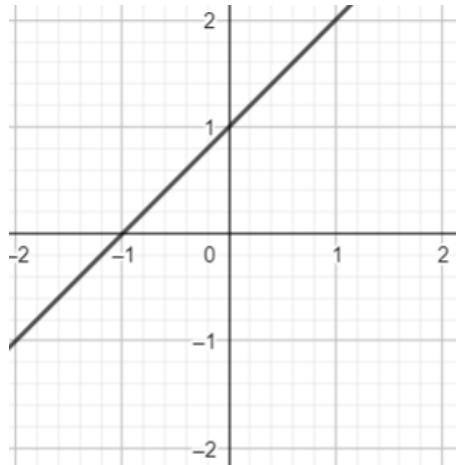
Tức là nhà diện tích x_i thực sự có giá y_i . Còn giá trị mà model hiện tại đang dự đoán kí hiệu là $\hat{y}_i = w_1 * x_i + w_0$

3.2.2 Loss function

Việc tìm w_0, w_1 có thể đơn giản nếu làm bằng mắt nhưng máy tính không biết điều đấy, nên ban đầu giá trị được chọn ngẫu nhiên ví dụ $w_0 = 0, w_1 = 1$ sau đấy được chỉnh dần.

Rõ ràng có thể thấy đường $y = x$ như ở hình 3.4 không hề gần các điểm hay không phải là đường mà ta cần tìm. Ví dụ tại điểm $x = 42$ (nhà $42 m^2$) giá thật là 625 triệu nhưng giá mà model dự đoán chỉ là 42 triệu.

Hình 3.2: Ước tính giá căn nhà 50 m²Hình 3.4: Sự khác nhau tại điểm $x = 42$ của model đường thẳng $y = x$ và giá trị thực tế ở bảng 1



Hình 3.3: Ví dụ đường thẳng $y = x + 1$ ($a = 1$ và $b = 1$)

Nên giờ cần 1 hàm để đánh giá là đường thẳng với bộ tham số $(w_0, w_1) = (0, 1)$ có tốt hay không. Với mỗi điểm dữ liệu (x_i, y_i) độ chênh lệch giữa giá thật và giá dự đoán được tính bằng: $\frac{1}{2} * (\hat{y}_i - y_i)^2$. Và độ chênh lệch trên toàn bộ dữ liệu tính bằng tổng chênh lệch của từng điểm:

$$J = \frac{1}{2} * \frac{1}{N} * \left(\sum_{i=1}^N (\hat{y}_i - y_i)^2 \right) \text{ (N là số điểm dữ liệu). Nhận xét:}$$

- J không âm
- J càng nhỏ thì đường thẳng càng gần điểm dữ liệu. Nếu $J = 0$ thì đường thẳng đi qua tất cả các điểm dữ liệu.

J được gọi là loss function, hàm để đánh giá xem bộ tham số hiện tại có tốt với dữ liệu không.

=> Bài toán tìm đường thẳng gần các điểm dữ liệu nhất trở thành tìm w_0, w_1 sao cho hàm J đạt giá trị nhỏ nhất.

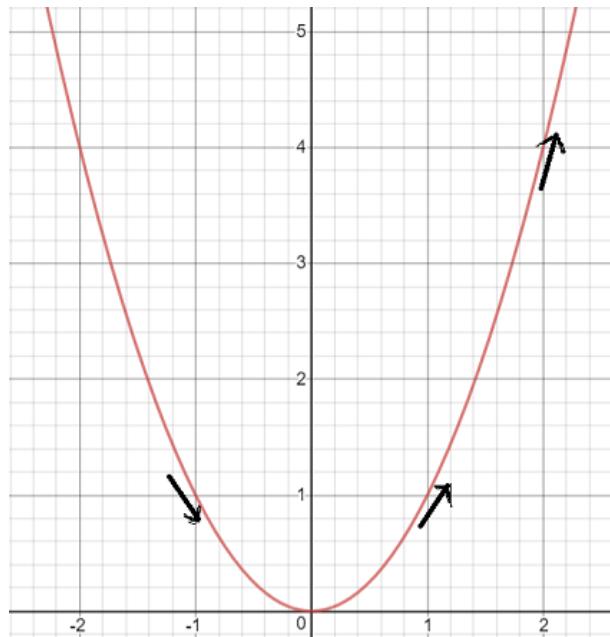
Tóm tắt: Cần tìm đường thẳng (model) fit nhất với dữ liệu, tương ứng với việc tìm tham số w_0, w_1 để cực tiểu hóa hàm J .

Giờ cần một thuật toán để tìm giá trị nhỏ nhất của hàm $J(w_0, w_1)$. Đó chính là thuật toán gradient descent.

3.3 Thuật toán gradient descent

3.3.1 Đạo hàm là gì

Có nhiều người có thể tính được đạo hàm của hàm $f(x) = x^2$ hay $f(x) = \sin(\cos(x))$ nhưng vẫn không biết thực sự đạo hàm là gì. Theo tiếng hán đạo là con đường, hàm là hàm số nên đạo hàm chỉ sự biến đổi của hàm số hay có tên thân thương hơn là độ dốc của đồ thị.

Hình 3.5: Đồ thị $y = x^2$

Như mọi người đã học đạo hàm $f(x) = x^2$ là $f'(x) = \frac{df(x)}{dx} = 2 * x$ (hoàn toàn có thể chứng minh từ định nghĩa nhưng cấp 3 mọi người đã học quá nhiều về công thức nên tôi không đề cập lại). Nhận xét:

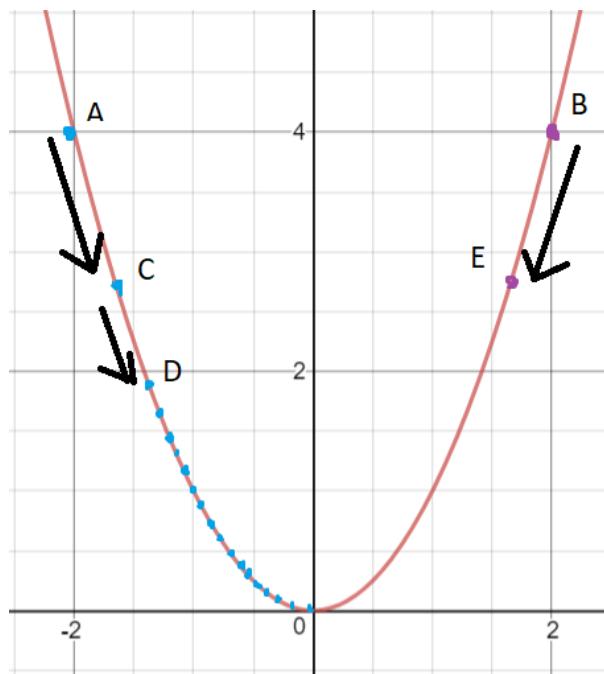
- $f'(1) = 2 * 1 < f'(2) = 2 * 2$ nên mọi người có thể thấy trên hình là đồ thị tại điểm $x = 2$ dốc hơn đồ thị tại điểm $x = 1$, tuy nhiên $f'(-2) = -4 < f'(-1) = -2$ nhưng đồ thị tại $x = -2$ dốc hơn đồ thị tại $x = -1 \Rightarrow$ trị tuyệt đối của đạo hàm tại một điểm càng lớn thì đồ thị tại điểm đấy càng dốc.
- $f'(-1) = 2 * (-1) = -2 < 0 \Rightarrow$ đồ thị đang giảm hay khi tăng x thì y sẽ giảm; ngược lại đạo hàm tại điểm nào đó mà dương thì đồ thị tại điểm đấy đang tăng.

3.3.2 Gradient descent

Gradient descent là thuật toán tìm giá trị nhỏ nhất của hàm số $f(x)$ dựa trên đạo hàm. Thuật toán:

1. Khởi tạo giá trị $x = x_0$ tùy ý
2. Gán $x = x - \text{learning_rate} * f'(x)$ (`learning_rate` là hằng số dương ví dụ `learning_rate = 0.001`)
3. Tính lại $f(x)$: Nếu $f(x)$ đủ nhỏ thì dừng lại, ngược lại tiếp tục bước 2

Thuật toán sẽ lặp lại bước 2 một số lần đủ lớn (100 hoặc 1000 lần tùy vào bài toán và hệ số `learning_rate`) cho đến khi $f(x)$ đạt giá trị đủ nhỏ. Ví dụ cần tìm giá trị nhỏ nhất hàm $y = x^2$, hàm này ai cũng biết là giá trị nhỏ nhất là 0 tại $x = 0$ nhưng để cho mọi người dễ hình dung hơn về thuật toán Gradient descent nên tôi lấy ví dụ đơn giản.



Hình 3.6: Ví dụ về thuật toán gradient descent

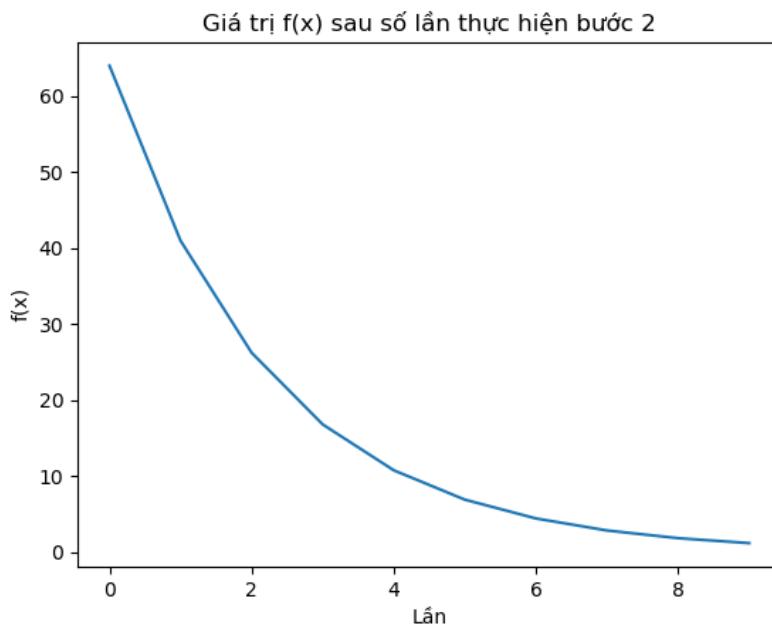
1. Bước 1: Khởi tạo giá trị ngẫu nhiên $x = -2$ (điểm A).
2. Bước 2: Do ở A đồ thị giảm nên $f'(-2) = 2*(-2) = -4 < 0 \Rightarrow$ Khi gán $x = x - learning_rate * f'(x)$ nên x tăng nên đồ thị bước tiếp theo ở điểm C. Tiếp tục thực hiện bước 2, gán $x = x - learning_rate * f'(x)$ thì đồ thị ở điểm D,... \Rightarrow hàm số giảm dần dần tiến tới giá trị nhỏ nhất.

Mọi người có để ý là trị tuyệt đối của đạo hàm tại A lớn hơn tại C và tại C lớn hơn tại D không? Đến khi đến gần điểm đạt giá trị nhỏ nhất $x = 0$, thì đạo hàm xấp xỉ 0 đến khi hàm đạt giá trị nhỏ nhất tại $x = 0$, thì đạo hàm bằng 0, nên tại điểm gần giá trị nhỏ nhất thì bước 2 gán $x = x - learning_rate * f'(x)$ là không đáng kể và gần như là giữ nguyên giá trị của x .

Tương tự nếu giá trị khởi tạo tại $x = 2$ (tại B) thì đạo hàm tại B dương nên do $x = x - learning_rate * f'(x)$ giảm \rightarrow đồ thị ở điểm E \rightarrow rồi tiếp tục gán $x=x - learning_rate * f'(x)$ thì hàm $f(x)$ cũng sẽ giảm dần dần đến giá trị nhỏ nhất.

Ví dụ: chọn $x = 10$, $learning_rate = 0.1$, bước 2 sẽ cập nhật $x = x - learning_rate * f'(x) = x - learning_rate * 2*x$, giá trị $f(x) = x^2$ sẽ thay đổi qua các lần thực hiện bước 2 như sau:

Lần	x	f(x)
1	8.00	64.00
2	6.40	40.96
3	5.12	26.21
4	4.10	16.78
5	3.28	10.74
6	2.62	6.87
7	2.10	4.40
8	1.68	2.81
9	1.34	1.80
10	1.07	1.15



Hình 3.7: Ví dụ về thuật toán gradient descent

```

import numpy as np
import matplotlib.pyplot as plt
x = 10

y = []
for i in range(10):
    x = x - 0.1 * 2 * x
    y.append(x**2)

plt.plot(y)
plt.xlabel('Số lần')
plt.ylabel('f(x)')
plt.title('Giá trị f(x) sau số lần thực hiện bước 2')
plt.show()

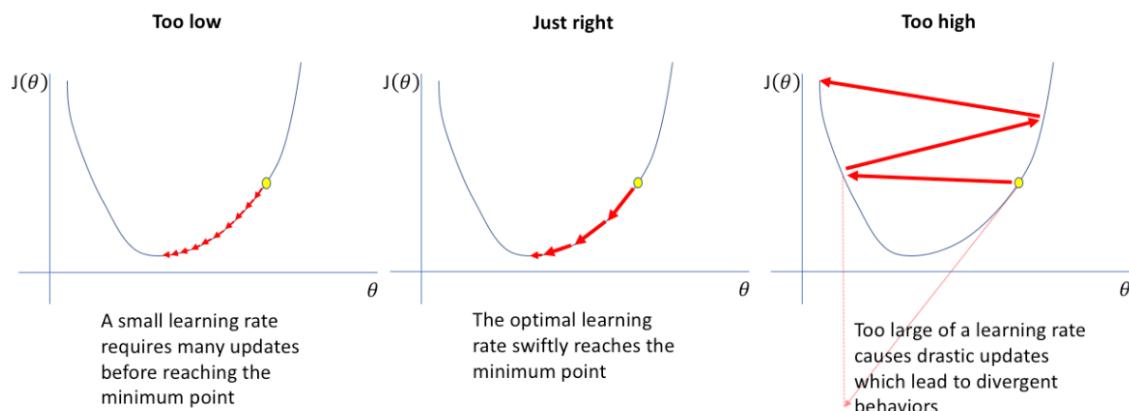
```

Nhận xét:

- Thuật toán hoạt động rất tốt trong trường hợp không thể tìm giá trị nhỏ nhất bằng đại số tuyến tính.
- Việc quan trọng nhất của thuật toán là tính đạo hàm của hàm số theo từng biến sau đó lặp lại bước 2.

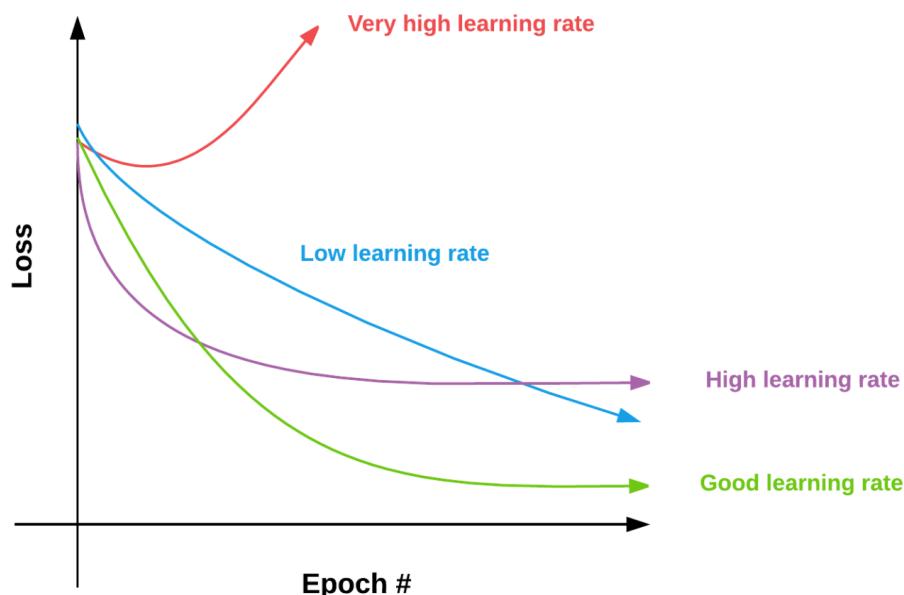
Việc chọn hệ số learning_rate cực kì quan trọng, có 3 trường hợp:

- Nếu learning_rate nhỏ: mỗi lần hàm số giảm rất ít nên cần rất nhiều lần thực hiện bước 2 để hàm số đạt giá trị nhỏ nhất.
- Nếu learning_rate hợp lý: sau một số lần lặp bước 2 vừa phải thì hàm sẽ đạt giá trị đủ nhỏ.
- Nếu learning_rate quá lớn: sẽ gây hiện tượng overshoot (như trong hình 3.8) và không bao giờ đạt được giá trị nhỏ nhất của hàm.



Hình 3.8: Các giá trị learning_rate khác nhau [13]

Cách tốt nhất để kiểm tra learning_rate hợp lý hay không là kiểm tra giá trị hàm $f(x)$ sau mỗi lần thực hiện bước 2 bằng cách vẽ đồ thị với trục x là số lần thực hiện bước 2, trục y là giá trị loss function tương ứng ở bước đấy.



Hình 3.9: Loss là giá trị của hàm cần tìm giá trị nhỏ nhất, Epoch ở đây là số cần thực hiện bước 2 [13]

3.3.3 Áp dụng vào bài toán

Ta cần tìm giá trị nhỏ nhất của hàm

$$J(w_0, w_1) = \frac{1}{2} * \frac{1}{N} * \left(\sum_{i=1}^N (\hat{y}_i - y_i)^2 \right) = \frac{1}{2} * \frac{1}{N} * \left(\sum_{i=1}^N (w_0 + w_1 * x_i - y_i)^2 \right)$$

Tuy nhiên do giá trị nhỏ nhất hàm $f(x)$ giống với hàm $\frac{f(x)}{N}$ (với $N > 0$) nên ta sẽ đi tìm giá trị nhỏ nhất của hàm:

$$J(w_0, w_1) = \frac{1}{2} * \left(\sum_{i=1}^N (w_0 + w_1 * x_i - y_i)^2 \right)$$

Việc tìm giá trị lớn nhất hàm này hoàn toàn có thể giải được bằng đại số tuyến tính nhưng để giới thiệu thuật toán Gradient descent cho bài Neural network nên tôi sẽ áp dụng Gradient descent luôn.

Việc quan trọng nhất của thuật toán gradient descent là tính đạo hàm của hàm số nên giờ ta sẽ đi tính đạo hàm của loss function theo từng biến.

Nhắc lại kiến thức $h'(x) = f(g(x))' = f'(g)*g'(x)$. Ví dụ:

$$h(x) = (3x + 1)^2 \text{ thì } f(x) = x^2, g(x) = 3x + 1 \Rightarrow h'(x) = f'(g) * g'(x) = f'(3x + 1) * g'(x) = 2 * (3x + 1) * 3 = 6 * (3x + 1).$$

$$\text{Tại 1 điểm } (x_i, y_i) \text{ gọi } f(w_0, w_1) = \frac{1}{2} * (w_0 + w_1 * x_i - y_i)^2$$

Ta có:

$$\frac{df}{dw_0} = w_0 + w_1 * x_i - y_i$$

$$\frac{df}{dw_1} = x_i * (w_0 + w_1 * x_i - y_i)$$

Do đó

$$\frac{dJ}{dw_0} = \sum_{i=1}^N (w_0 + w_1 * x_i - y_i)$$

$$\frac{dJ}{dw_1} = \sum_{i=1}^N x_i * (w_0 + w_1 * x_i - y_i)$$

3.4 Ma trận

3.4.1 Ma trận là gì

Ma trận là một mảng chữ nhật có m hàng và n cột, ta gọi là ma trận $m * n$ (số hàng nhân số cột).

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Hình 3.10: Ma trận kích thước $m*n$

Ví dụ về ma trận $3 * 4$

$$\begin{bmatrix} 2 & -5 & -11 & 0 \\ 4 & 2 & 1 & 4 \\ -1 & 3 & 0 & 5 \end{bmatrix}$$

Hình 3.11: Ma trận kích thước 3*4

Chỉ số ma trận thì hàng trước cột sau ví dụ $A[1, 2] = -5$ (hàng 1, cột 2); $A[3, 1] = 4$ (hàng 3, cột 1)

3.4.2 Phép nhân ma trận

Phép tính nhân ma trận $A * B$ chỉ thực hiện được khi số cột của A bằng số hàng của B , hay A có kích thước $m*n$ và B có kích thước $n*k$.

Ma trận $C = A * B$ thì C có kích thước $m * k$ và $C[i, j] = \sum_{k=1}^n A[i, k] * B[k, j]$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

Hình 3.12: Ví dụ phép nhân ma trận

Ví dụ:

$$C[1][1] = A[1][1] * B[1][1] + A[1][2] * B[2][1] = ax + cy$$

$$C[2][1] = A[2][1] * B[1][1] + A[2][2] * B[2][1] = bx + dy$$

3.4.3 Element-wise multiplication matrix

Ma trận A và B cùng kích thước $m*n$ thì phép tính này cho ra ma trận C cùng kích thước $m*n$ và $C[i,j] = A[i,j] * B[i,j]$. Hay là mỗi phần tử ở ma trận C bằng tích 2 phần tử tương ứng ở A và B . Kí hiệu $C = A \otimes B$, ví dụ:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} * b_{11} & a_{12} * b_{12} \\ a_{21} * b_{21} & a_{22} * b_{22} \end{bmatrix}$$

Hình 3.13: Ví dụ element-wise

3.4.4 Biểu diễn bài toán

Do với mỗi điểm x_i, y_i ta cần phải tính $(w_0 + w_1 * x_i - y_i)$ nên thay vì tính cho từng điểm dữ liệu một ta sẽ biểu diễn dưới dạng ma trận, X kích thước $n * 2$, Y kích thước $n * 1$ (n là số điểm dữ liệu trong tập dữ liệu mà ta có).

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots & \dots \\ 1 & x_n \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}, w = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

$$\hat{y} = X * W = \begin{bmatrix} w_0 + w_1 * x_1 \\ w_0 + w_1 * x_2 \\ \dots \\ w_0 + w_1 * x_n \end{bmatrix}$$

Hình 3.14: Biểu diễn bài toán dạng ma trận

$X[:, i]$ hiểu là ma trận kích thước $n*1$ lấy dữ liệu từ cột thứ i của ma trận X , nhưng do trong python chỉ số bắt đầu từ 0, nên cột đầu tiên là cột 0, cột thứ hai là cột 1, Phép tính $\text{sum}(X)$ là tính tổng tất cả các phần tử trong ma trận X .

$$X[:, 1] = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

$$\text{sum}(X[:, 1]) = x_1 + x_2 + \dots + x_n$$

$$\frac{dJ}{dw_0} = \text{sum}(\hat{y} - y)$$

$$\frac{dJ}{dw_1} = \text{sum}(X[:, 1] \otimes (\hat{y} - y))$$

$$\frac{dJ}{dw} = X^T * (\hat{y} - y)$$

Hình 3.15: Biểu diễn đạo hàm dạng ma trận

3.5 So sánh các loss function

3.5.1 Mean Absolute Error, L1 Loss

Mean Absolute Error (MAE) hay còn được gọi là L1 Loss là một loss function được sử dụng cho các mô hình hồi quy, đặc biệt cho các mô hình hồi quy tuyến tính. MAE được tính bằng tổng các trị tuyệt đối của hiệu giữa giá trị thực (y_i : target) và giá trị mà mô hình của chúng ra dự đoán (\hat{y}_i : predicted).

$$MAE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{n} \quad (3.1)$$

3.5.2 Mean Square Error, L2 Loss

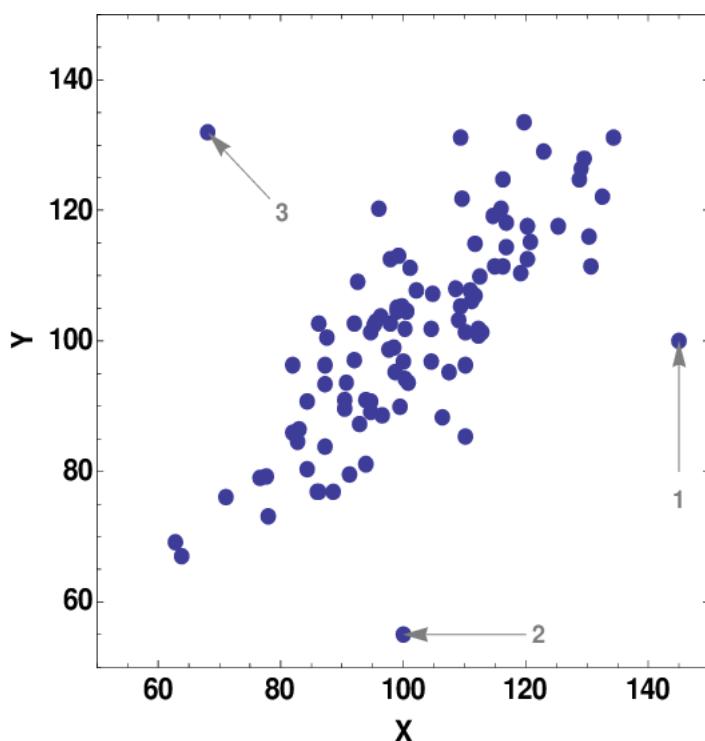
Mean Square Error (MSE) hay còn được gọi là L2 Loss là một loss function cũng được sử dụng cho các mô hình hồi quy, đặc biệt là các mô hình hồi quy tuyến tính. MSE được tính bằng tổng các

bình phương của hiệu giữa giá trị thực (y_i : target) và giá trị mà mô hình của chúng ra dự đoán (\hat{y}_i : predicted).

$$MSE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n} \quad (3.2)$$

3.5.3 So sánh MAE và MSE (L1 Loss và L2 Loss)

Để tìm điểm cực tiểu của các hàm số, thông thường cách đơn giản nhất chúng ta nghĩ đến chính là tìm đạo hàm của hàm số rồi tìm điểm mà tại đó đạo hàm của hàm số bằng 0. Như vậy sẽ có 1 bước đạo hàm, hiển nhiên tìm đạo hàm của MSE sẽ đơn giản hơn rất nhiều so với MAE. Tuy nhiên, MAE thì đem lại kết quả tốt hơn đối với các dữ liệu có outlier (điểm dị biệt - là những điểm có giá trị khác xa so với phần còn lại của dữ liệu).



Hình 3.16: Dữ liệu bị outlier

Đầu tiên hãy để ý đến MSE, ta có $y_i - \hat{y}_i = e$, e^2 sẽ vô cùng lớn nếu $e > 1$. Nếu bậc của e càng lớn thì giá trị hàm Loss cũng càng lớn hơn. Vì vậy nếu như chúng ta có 1 outlier trong bộ dữ liệu, giá trị của e sẽ vô cùng lớn, có thể tiến tới ∞ và khi đó, e^2 có thể sẽ tiến tới ∞ , khi đó giá trị MSE cực kì lớn.

Ngược lại với MSE, nếu ta có giá trị e lớn ($e > 1$) thì $|e|$ vẫn sẽ lớn, nhưng hiển nhiên nhỏ hơn nhiều so với e^2 .

Do đó khi tối ưu loss function, L2 phạt mạnh hơn với các điểm outlier và model sẽ bị kéo về phía outlier hơn. Do đó MSE bị ảnh hưởng bởi outlier và L1 tốt hơn đối với các dữ liệu có outlier.

Tuy vậy đạo hàm của MAE (hay L1 Loss) tính toán khó hơn nhiều so với MSE (hay L2 Loss). Cùng thử tính toán nhé.

Ta đặt $y_i - \hat{y}_i = e$.

$$|e|' = (\sqrt{e^2})' = \frac{1}{2\sqrt{e^2}} * 2e * e' = \frac{e}{\sqrt{e^2}} * e' \quad (3.3)$$

$$(e^2)' = 2e * e' \quad (3.4)$$

Để dễ hiểu hơn tại sao MAE và MSE có sự khác nhau như vậy với các outlier, hãy cùng làm 1 ví dụ đơn giản sau: Giả sử mình có 4 điểm dữ liệu 1, 2, 4, 33. Mình cùng tìm thử nghiệm theo L1 và L2 nhé.

Tìm min của:

- a) $L1 = |x - 1| + |x - 2| + |x - 4| + |x - 33|$
 - + $-\infty < x \leq 1$: $L1 = 40 - 4x$ đạt min = 36 tại $x = 1$
 - + $1 < x \leq 2$: $L1 = 38 - 2x$ đạt min = 34 tại $x = 2$
 - + $2 < x \leq 4$: $L1 = 34$
 - + $4 < x \leq 33$: $L1 = 2x + 26 > 34$
 - + $33 < x \leq +\infty$: $L1 = 4x - 40$ đạt min $> 4 * 33 - 40 = 92$
- Vậy $L1$ đạt min = 34 tại $2 \leq x \leq 4$

$$\begin{aligned} b) L2 &= (x - 1)^2 + (x - 2)^2 + (x - 4)^2 + (x - 33)^2 \\ L2' &= 2 * (x - 1 + x - 2 + x - 4 + x - 33) = 0 \Leftrightarrow x = 10 \\ L2 &\text{ đạt min} = 710 \text{ tại } x = 10. \end{aligned}$$

Trong dãy số 1, 2, 4, 33 thì có 33 là giá trị ngoại lai, L2 đạt min lớn hơn rất nhiều so với L1. Và để ý hơn thì thấy L2 đạt min tại trung bình (mean) của các giá trị $\frac{1+2+4+33}{4} = 10$ còn L1 đạt min tại trung vị (median) của các giá trị đó \Rightarrow Do đó L1 sẽ tốt hơn với dữ liệu có các outlier.

Tiêu chí xây dựng hàm loss function là không âm và loss càng nhỏ thì model càng tốt với dữ liệu. Từ đầu đến giờ có lẽ các bạn vẫn thắc mắc tại sao trong các hàm loss chỉ dùng trị tuyệt đối, bậc 2, thế bậc 2n thì sao? Câu trả lời là bạn chọn bậc càng cao thì càng bị ảnh hưởng bởi outlier. Ví dụ so sánh các loss function ở hình 3.17. Và 2 loss bạn hay gặp nhất sẽ là L1 (Mean Absolute Error) và L2 (Mean Square Error).

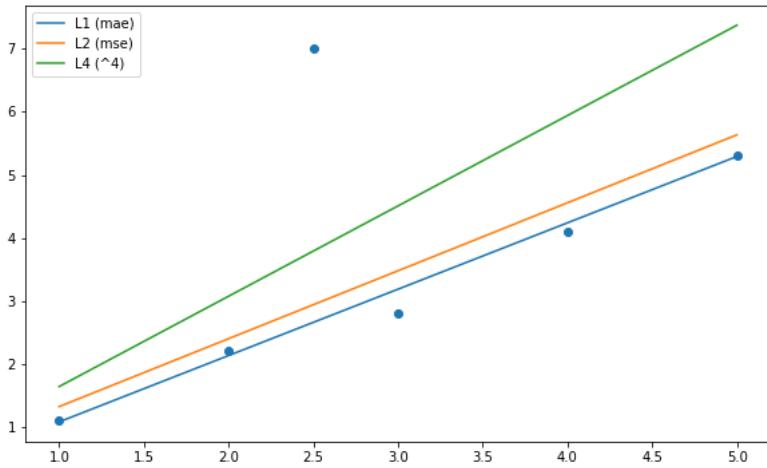
3.6 Giải bằng đại số tuyến tính

$w = [w_0, w_1, w_2, \dots, w_m]^T$ là vecto hệ số cần phải tối ưu, w_0 thường hay được gọi là bias.
 $x^{(i)} = [1, x_1^{(i)}, x_2^{(i)}, \dots, x_m^{(i)}]$ là dữ liệu thứ i trong bộ n số lượng dữ liệu quan sát được, mỗi dữ liệu có m giá trị.

$$\Rightarrow \hat{y}^{(i)} = x^{(i)}w$$

Biểu diễn tổng quát bài toán Linear Regression:

$$X \in \mathbb{R}^{n*(m+1)}, w \in \mathbb{R}^{(m+1)*1}, y \in \mathbb{R}^{n*1}$$



Hình 3.17: So sánh các loss function

$$X = \begin{pmatrix} x^{(1)} \\ x^{(2)} \\ \dots \\ x^{(n)} \end{pmatrix}, w = \begin{pmatrix} w_0 \\ w_1 \\ \dots \\ w_m \end{pmatrix}, \hat{y} = Xw = \begin{pmatrix} w_0 + w_1x_1^{(1)} + \dots + w_mx_m^{(1)} \\ w_0 + w_1x_1^{(2)} + \dots + w_mx_m^{(2)} \\ \dots \\ w_0 + w_1x_1^{(n)} + \dots + w_mx_m^{(n)} \end{pmatrix}, y - \hat{y} = \begin{pmatrix} y^{(1)} - (w_0 + w_1x_1^{(1)} + \dots + w_mx_m^{(1)}) \\ y^{(2)} - (w_0 + w_1x_1^{(2)} + \dots + w_mx_m^{(2)}) \\ \dots \\ y^{(n)} - (w_0 + w_1x_1^{(n)} + \dots + w_mx_m^{(n)}) \end{pmatrix}$$

$$\text{Hàm loss: } L = \frac{1}{2} * \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - \hat{y}^{(i)} \right)^2 = \frac{1}{2} * \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - x^{(i)}w \right)^2$$

$$\text{Định nghĩa Euclidean norm: } \|z\|_2 = (z_1^2 + z_2^2 + \dots + z_n^2)^{\frac{1}{2}} \implies \|z\|_2^2 = (z_1^2 + z_2^2 + \dots + z_n^2)$$

$$\implies L = \frac{1}{2} * \frac{1}{n} \|y - \hat{y}\|_2^2 = \frac{1}{2} * \frac{1}{n} \|y - Xw\|_2^2 = \frac{1}{2} * \frac{1}{n} * (y - \hat{y})^T * (y - \hat{y})$$

Giải thích chi tiết:

$$\begin{aligned}
 &+ > (Xw)^T = w^T X^T \\
 &+ > x^{(i)}w = \begin{bmatrix} 1 & x_1^{(i)} & \dots & x_m^{(i)} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_m \end{bmatrix} = w_0 + w_1x_1^{(i)} + \dots + w_mx_m^{(i)} \\
 &\implies \frac{d(x^{(i)}w)}{dw} = \begin{bmatrix} \frac{d(x^{(i)}w)}{dw_0} & \frac{d(x^{(i)}w)}{dw_1} & \dots & \frac{d(x^{(i)}w)}{dw_m} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(i)} & \dots & x_m^{(i)} \end{bmatrix} = x^{(i)} \\
 &\implies \frac{d(Xw)}{dw} = X
 \end{aligned}$$

$$\begin{aligned}
 + > wx^{(i)} &= \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_m \end{bmatrix} \begin{bmatrix} 1 & x_1^{(i)} & \dots & x_m^{(i)} \end{bmatrix} = w_0 + w_1 x_1^{(i)} + \dots + w_m x_m^{(i)} \\
 \implies \frac{d(wx^{(i)})}{dw} &= \begin{bmatrix} \frac{d(x^{(i)}w)}{dw_0} \\ \frac{d(x^{(i)}w)}{dw_1} \\ \dots \\ \frac{d(x^{(i)}w)}{dw_m} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \dots \\ x_m^{(i)} \end{bmatrix} = x^{(i)T} \\
 \implies \frac{d(wX)}{dw} &= X^T \\
 + > \frac{d(w^T X w)}{dw} &= w^T X + \frac{d((Xw)^T w)}{dw} = w^T X + w^T X^T = w^T (X + X^T) \quad (\text{Nếu } X \text{ đối xứng}) \\
 &= 2w^T X
 \end{aligned}$$

*Để ý $X^T X$ đối xứng

$$\implies \frac{dX}{dw} = 0, \frac{d(Xw)}{dw} = X, \frac{d(wX)}{dw} = X^T, \frac{d(w^T X w)}{dw} = w^T (X + X^T)$$

$$A = (y - \hat{y})^T * (y - \hat{y}) = (y - Xw)^T * (y - Xw)$$

$$= (y^T - w^T X^T) (y - Xw) = y^T y - y^T Xw - w^T X^T y + w^T X^T Xw$$

$$\begin{aligned}
 \implies A'_w &= 0 - y^T X - ((y^T X) w)' + 2w^T X^T X \\
 &= -y^T X - ((y^T X) w)' + 2w^T X^T X \\
 &= -y^T X - y^T X + 2w^T X^T X \\
 &= -y^T X - y^T X + 2w^T X^T X \\
 &= 0
 \end{aligned}$$

$$\Leftrightarrow 2w^T X^T X = 2y^T X$$

$$\Leftrightarrow w^T X^T X = y^T X$$

$$\Leftrightarrow (X^T X) * w = X^T y$$

$$\Leftrightarrow w = (X^T X)^{-1} * X^T y$$

Nếu $X^T X$ khả nghịch, thì L có nghiệm duy nhất: $w = (X^T X)^{-1} X^T y$

Nếu không khả nghịch, ta có thể sử dụng khái niệm giả nghịch đảo, bạn đọc có thể tự tìm hiểu thêm ở [đây](#).

3.7 Python code

```
# -*- coding: utf-8 -*-
```

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#đoạn code sinh ra dữ liệu
#numOfPoint = 30
#noise = np.random.normal(0, 1, numOfPoint).reshape(-1, 1)
#x = np.linspace(30, 100, numOfPoint).reshape(-1, 1)
#N = x.shape[0]
#y = 15*x + 8 + 20*noise

```

```
#plt.scatter(x, y)

data = pd.read_csv('data_linear.csv').values
N = data.shape[0]
x = data[:, 0].reshape(-1, 1)
y = data[:, 1].reshape(-1, 1)
plt.scatter(x, y)
plt.xlabel('mét vuông')
plt.ylabel('giá')

x = np.hstack((np.ones((N, 1)), x))

w = np.array([0.,1.]).reshape(-1,1)

numOfIteration = 100
cost = np.zeros((numOfIteration,1))
learning_rate = 0.000001
for i in range(1, numOfIteration):
    r = np.dot(x, w) - y
    cost[i] = 0.5*np.sum(r*r)
    w[0] -= learning_rate*np.sum(r)
    # correct the shape dimension
    w[1] -= learning_rate*np.sum(np.multiply(r, x[:,1].reshape(-1,1)))
    print(cost[i])
predict = np.dot(x, w)
plt.plot((x[0][1], x[N-1][1]),(predict[0], predict[N-1]), 'r')
plt.show()

x1 = 50
y1 = w[0] + w[1] * x1
print('Giá nhà cho 50m^2 là : ', y1)

# Lưu w với numpy.save(), định dạng '.npy'
np.save('weight.npy', w)
# Đọc file '.npy' chứa tham số weight
w = np.load('weight.npy')

# LinearRegression với thư viện sklearn
from sklearn.linear_model import LinearRegression

data = pd.read_csv('data_linear.csv').values
x = data[:, 0].reshape(-1, 1)
y = data[:, 1].reshape(-1, 1)

plt.scatter(x, y)
plt.xlabel('mét vuông')
plt.ylabel('giá')

# Tao mo hinh hooi quy tuyen tinh
```

```

lrg = LinearRegression()
# Train mô hình với data giá đất
lrg.fit(x, y)
# Đoán giá nhà đất
y_pred = lrg.predict(x)

plt.plot((x[0], x[-1]),(y_pred[0], y_pred[-1]), 'r')
plt.show()

# Lưu nhiều tham số với numpy.savez(), định dạng '.npz'
np.savez('w2.npz', a=lrg.intercept_, b=lrg.coef_)
# Lấy lại các tham số trong file .npz
k = np.load('w2.npz')
lrg.intercept_ = k['a']
lrg.coef_ = k['b']

```

3.8 Bài tập

1. Thực hiện các phép nhân ma trận sau:

(a)

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} * \begin{pmatrix} d & e & f \end{pmatrix}$$

(b)

$$\begin{pmatrix} a & b & c \end{pmatrix} * \begin{pmatrix} d \\ e \\ f \end{pmatrix}$$

(c)

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} * \begin{pmatrix} g & h & i \\ k & m & n \end{pmatrix}$$

(d)

$$\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots & \dots \\ 1 & x_n \end{pmatrix} * \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$$

2. Tính đạo hàm các hàm số sau:

(a) $f(x) = (2x+1)^2$

(b) $f(x) = \frac{1}{1+e^{-x}}$

(c) $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

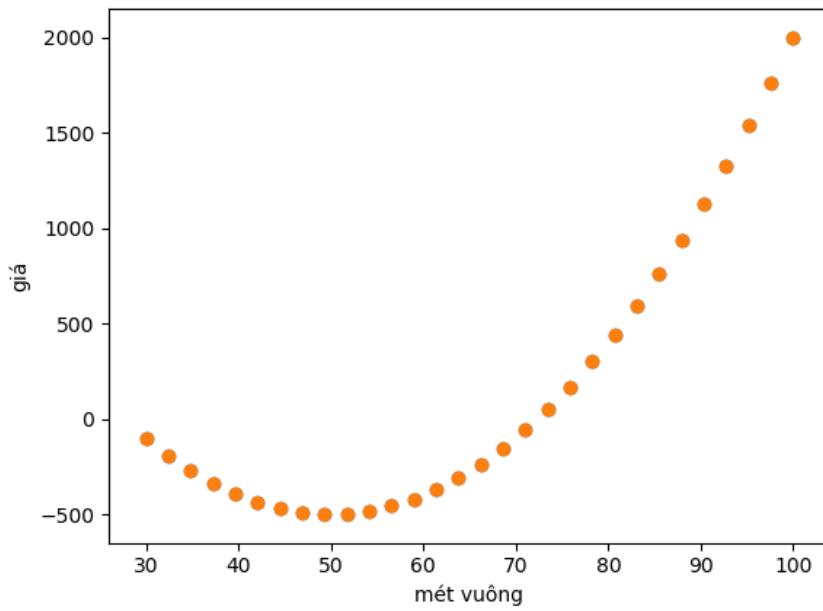
(d) $f(x) = x^x$

3. (a) Tự giải thích lại thuật toán gradient descent

- (b) Code python dùng thuật toán gradient descent để tìm giá trị nhỏ nhất của hàm $f(x) = x^2 + 2x + 5$

4. Tự tính đạo hàm của weight với loss function trong bài toán linear regression và biểu diễn lại dưới dạng ma trận.

5. Dựa vào code được cung cấp chỉnh 1 số tham số như learning_rate (tăng, giảm), số iteration xem loss function sẽ thay đổi thế nào.
6. Giả sử giá nhà phụ thuộc vào diện tích và số phòng ngủ. Thiết kế 1 linear regression model và định nghĩa loss function cho bài toán trên.
7. Giả sử bài toán vẫn như trên nhưng khi bạn vẽ đồ thị dữ liệu sẽ như thế này



Model cho bài toán bạn chọn như thế nào? Code python cho bài toán đầy tương tự như bài toán linear regression. Dữ liệu ở file data_square.csv trên github.

4. Logistic regression

Bài trước học về linear regression với đầu ra là giá trị thực, thì ở bài này sẽ giới thiệu thuật toán logistic regression với đầu ra là giá trị nhị phân (0 hoặc 1), ví dụ: email gửi đến hộp thư của bạn có phải spam hay không; u là u lành tính hay ác tính,...

4.1 Bài toán

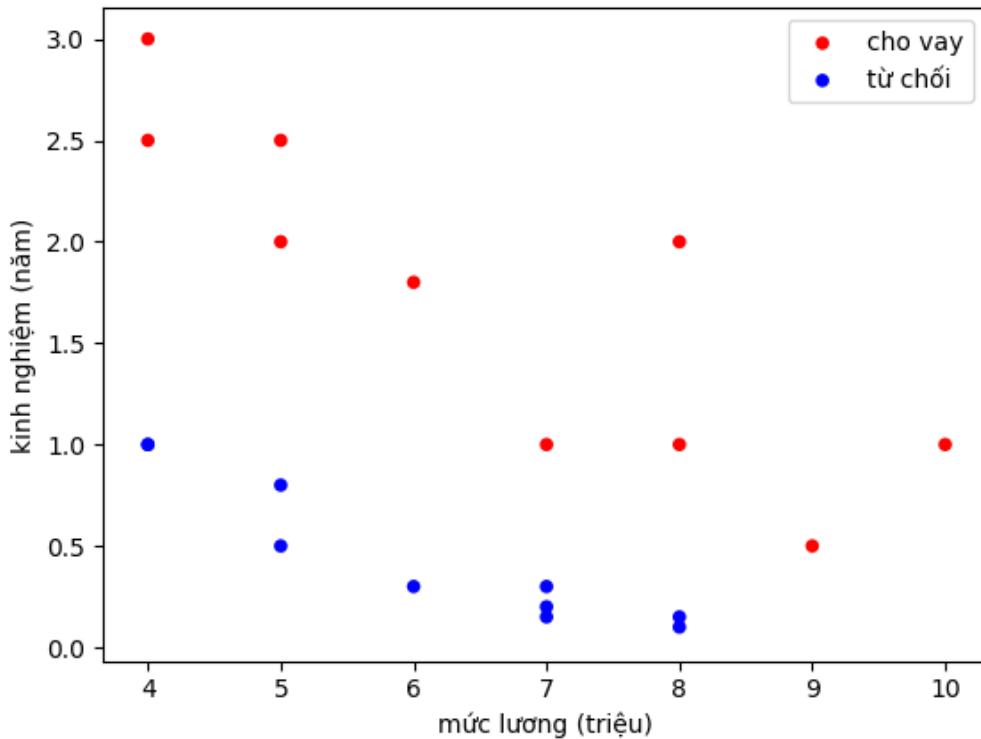
Ngân hàng bạn đang làm có chương trình cho vay ưu đãi cho các đối tượng mua chung cư. Tuy nhiên gần đây có một vài chung cư rất hấp dẫn (giá tốt, vị trí đẹp,...) nên lượng hồ sơ người nộp cho chương trình ưu đãi tăng đáng kể. Bình thường bạn có thể duyệt 10-20 hồ sơ một ngày để quyết định hồ sơ có được cho vay hay không, tuy nhiên gần đây bạn nhận được 1000-2000 hồ sơ mỗi ngày. Bạn không thể xử lý hết hồ sơ và bạn cần có một giải pháp để có thể dự đoán hồ sơ mới là có nên cho vay hay không.

Sau khi phân tích thì bạn nhận thấy là hai yếu tố chính quyết định đến việc được vay tiền đó là mức lương và thời gian công tác. Đây là dữ liệu bạn có từ trước đến nay:

Lương	Thời gian làm việc	Cho vay
10	1	1
9	0.5	1
5	2	1
...
8	0.1	0
6	0.3	0
7	0.15	0
...

Khi có dữ liệu bạn visualize dữ liệu lên như hình 4.1

Về mặt logic, giờ ta cần tìm đường thẳng phân chia giữa các điểm cho vay và từ chối. Rồi quyết định hồ sơ mới có nên có vay hay không từ đường đấy như hình 4.2



Hình 4.1: Đồ thị giữa mức lương, số năm kinh nghiệm và kết quả cho vay

Ví dụ đường xanh là đường phân chia. Dự đoán cho hồ sơ của người có mức lương 6 triệu và 1 năm kinh nghiệm là không cho vay.

Tuy nhiên, do ngân hàng đang trong thời kỳ khó khăn nên việc cho vay bị thắt lại, chỉ những hồ sơ nào chắc chắn trên 80% mới được vay.

Vậy nên bây giờ bạn không những tìm là hồ sơ ấy cho vay hay không cho vay mà cần tìm xác suất nên cho hồ sơ ấy vay là bao nhiêu.

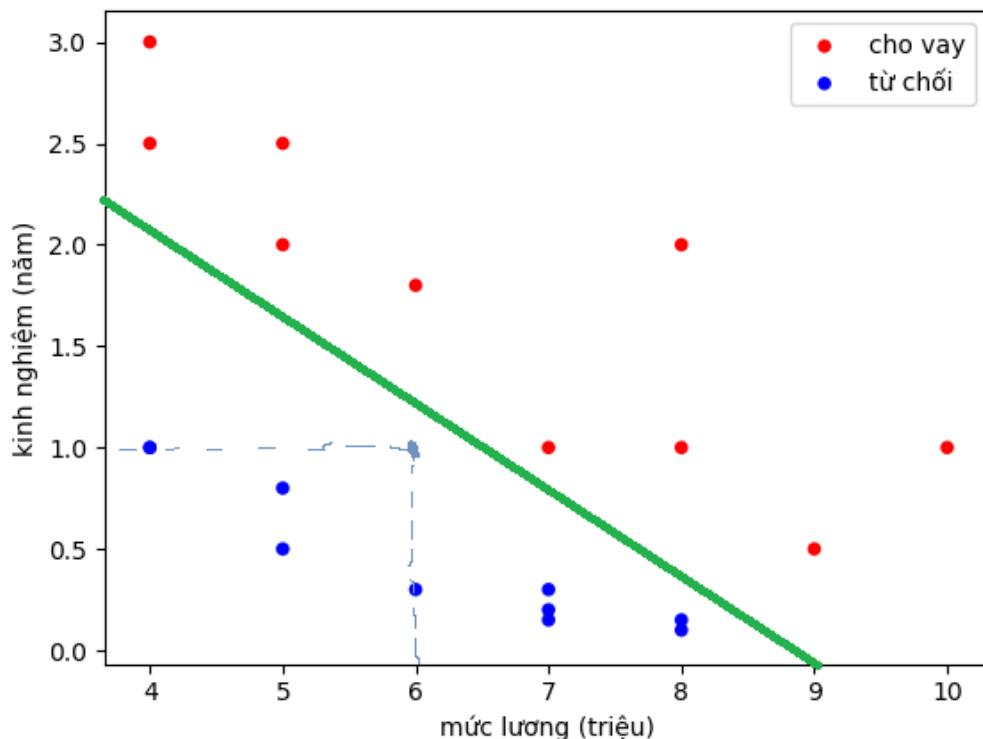
Hay trong nhiều trường hợp khác trong bài toán phân loại người ta quan tâm hơn đến xác suất hay vì chỉ 1 hay 0. Ví dụ: bác sĩ sẽ thông báo ca mổ này 80% thành công cho người nhà bệnh nhân.

4.2 Xác suất

Bạn được học xác suất từ cấp hai, cấp ba rồi đến toán cao cấp, nhưng có bao giờ bạn hỏi tại sao lại có xác suất không? Vì trong cuộc sống này có những sự việc không chắc chắn, ví dụ ngày mai trời có mưa không. Vậy nên xác suất ra đời để đo lường sự không chắc chắn ấy.

Vậy xác suất là gì? "Các nhà toán học coi xác suất là các số trong khoảng [0,1], được gán tương ứng với một biến cố mà khả năng xảy ra hoặc không xảy ra là ngẫu nhiên" [28]. Ví dụ bạn tung đồng xu có 2 mặt, thì xác suất bạn tung được mặt ngửa là 50% ($= 50/100 = 0.5$).

Nhận xét:



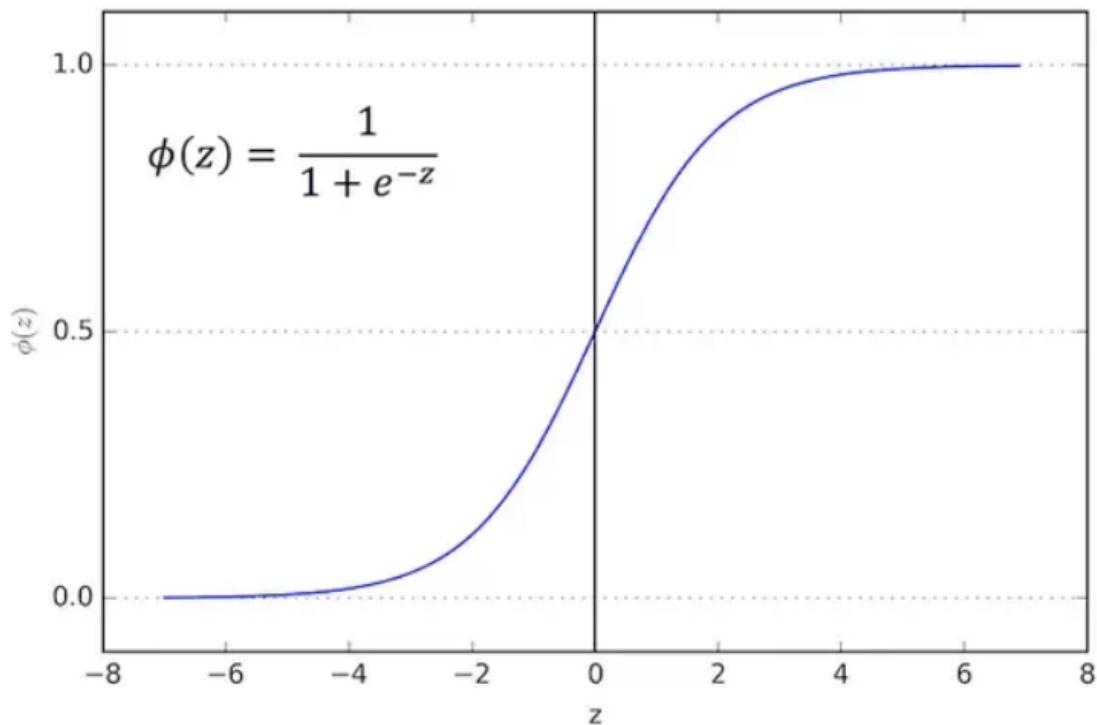
Hình 4.2: Đường phân chia và dự đoán điểm dữ liệu mới

- Xác suất của 1 sự kiện trong khoảng [0,1]
- Sự kiện bạn càng chắc chắn xảy ra thì xác suất càng cao. Ví dụ bạn lương cao và còn đi làm lâu lăm thì xác suất bạn được vay mua chung cư là cao.
- Tổng xác suất của sự kiện A và sự kiện phủ định của A là 100% (hay 1). Ví dụ sự kiện A: tung đồng xu mặt ngửa, xác suất 50%; phủ định của sự kiện A: tung đồng xu mặt sấp, xác suất 50% => tổng 100%.

Bạn sẽ thấy xác suất quan trọng hơn là chỉ 0 hay 1, ví dụ trước mỗi ca mổ khó, bác sĩ không thể chắc chắn là sẽ thất bại hay thành công mà chỉ có thể nói xác suất thành công là bao nhiêu (ví dụ 80%).

4.3 Hàm sigmoid

Giờ ta cần tìm xác suất của hồ sơ mới nên cho vay. Hay giá trị của hàm cần trong khoảng [0,1]. Rõ ràng là giá trị của phương trình đường thẳng như bài trước có thể ra ngoài khoảng [0,1] nên cần một hàm mới luôn có giá trị trong khoảng [0,1]. Đó là hàm sigmoid.



Hình 4.3: Đồ thị hàm sigmoid

Nhận xét:

- Hàm số liên tục, nhận giá trị thực trong khoảng (0,1).
- Hàm có đạo hàm tại mọi điểm (để áp dụng gradient descent).

4.4 Thiết lập bài toán

Mọi người có để ý các bước trong bài linear regression không nhỉ, các bước bao gồm:

1. Visualize dữ liệu
2. Thiết lập model
3. Thiết lập loss function
4. Tìm tham số bằng việc tối ưu loss function
5. Dự đoán dữ liệu mới bằng model vừa tìm được

Đây là mô hình chung cho bài toán trong Deep Learning.

4.4.1 Model

Với dòng thứ i trong bảng dữ liệu, gọi $x_1^{(i)}$ là lương và $x_2^{(i)}$ là thời gian làm việc của hồ sơ thứ i .

$p(x^{(i)} = 1) = \hat{y}_i$ là xác suất mà model dự đoán hồ sơ thứ i được cho vay.

$p(x^{(i)} = 0) = 1 - \hat{y}_i$ là xác suất mà model dự đoán hồ sơ thứ i không được cho vay.

$$\Rightarrow p(x^{(i)} = 1) + p(x^{(i)} = 0) = 1$$

$$\text{Hàm sigmoid: } \sigma(x) = \frac{1}{1 + e^{-x}}.$$

Như bài trước công thức của linear regression là: $\hat{y}_i = w_0 + w_1 * x_i$ thì giờ công thức của logistic regression là:

$$\hat{y}_i = \sigma(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}) = \frac{1}{1 + e^{-(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)})}}$$

Ở phần cuối mọi người sẽ thấy được quan hệ giữa xác suất và đường thẳng.

4.4.2 Loss function

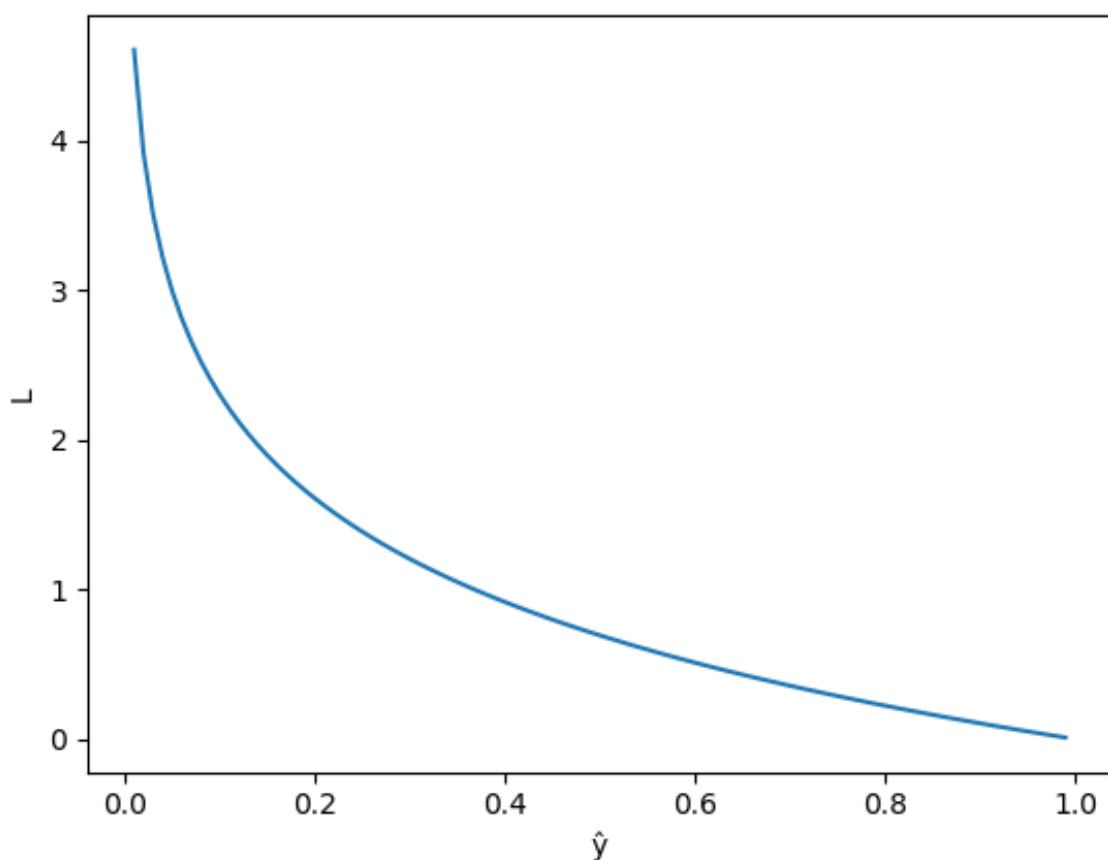
Giờ cũng cần một hàm để đánh giá độ tốt của model. Như bài trước là \hat{y} càng gần y càng tốt, giờ cũng vậy:

- Nếu hồ sơ thứ i là cho vay, tức $y_i = 1$ thì ta cũng mong muốn \hat{y}_i càng gần 1 càng tốt hay model dự đoán xác suất người thứ i được vay vốn càng cao càng tốt.
- Nếu hồ sơ thứ i không được vay, tức $y_i = 0$ thì ta cũng mong muốn \hat{y}_i càng gần 0 càng tốt hay model dự đoán xác suất người thứ i được vay vốn càng thấp càng tốt.

Với mỗi điểm $(x^{(i)}, y_i)$, gọi hàm loss function $L = -(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$, loss function này có tên gọi là binary_crossentropy

Mặc định trong machine learning nói chung hay deep learning thì viết log hiểu là ln

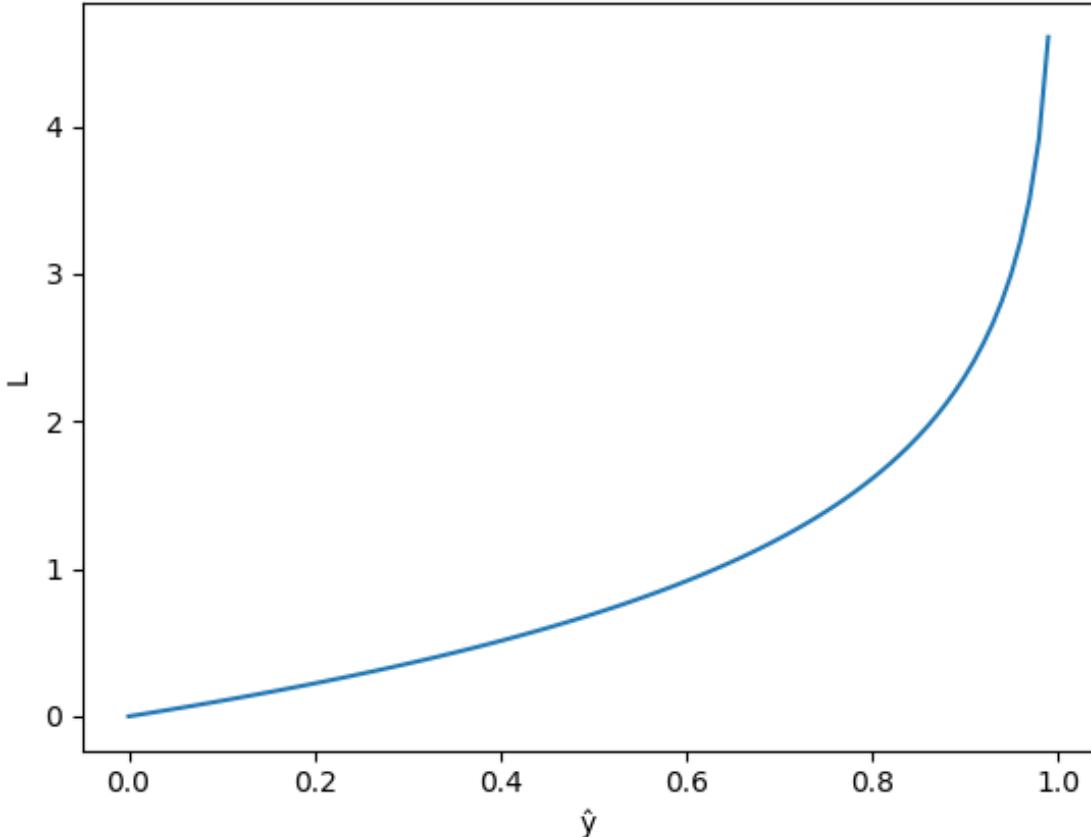
Thử đánh giá hàm L nhé. Nếu $y_i = 1 \Rightarrow L = -\log(\hat{y}_i)$



Hình 4.4: Đồ thị hàm loss function trong trường hợp $y_i = 1$

Nhận xét:

- Hàm L giảm dần từ 0 đến 1.
 - Khi model dự đoán \hat{y}_i gần 1, tức giá trị dự đoán gần với giá trị thật y_i thì L nhỏ, xấp xỉ 0
 - Khi model dự đoán \hat{y}_i gần 0, tức giá trị dự đoán ngược lại giá trị thật y_i thì L rất lớn
- Ngược lại, nếu $y_i = 0 \Rightarrow L = -\log(1 - \hat{y}_i)$



Hình 4.5: Đồ thị hàm loss function trong trường hợp $y_i = 0$

Nhận xét:

- Hàm L tăng dần từ 0 đến 1
 - Khi model dự đoán \hat{y}_i gần 0, tức giá trị dự đoán gần với giá trị thật y_i thì L nhỏ, xấp xỉ 0
 - Khi model dự đoán \hat{y}_i gần 1, tức giá trị dự đoán ngược lại giá trị thật y_i thì L rất lớn
- => Hàm L nhỏ khi giá trị model dự đoán gần với giá trị thật và rất lớn khi model dự đoán sai, hay nói cách khác L càng nhỏ thì model dự đoán càng gần với giá trị thật. => Bài toán tìm model trở thành tìm giá trị nhỏ nhất của L

Hàm loss function trên toàn bộ dữ liệu $J = -\frac{1}{N} * \sum_{i=1}^N (y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$

4.5 Chain rule

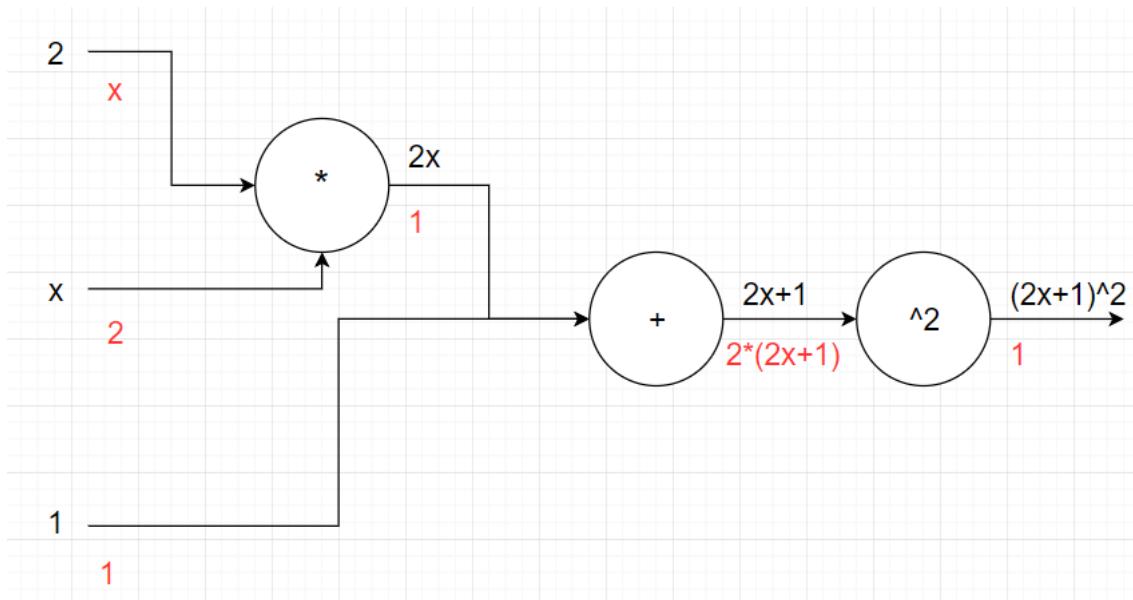
Chain rule là gì? Nếu $z = f(y)$ và $y = g(x)$ hay $z = f(g(x))$ thì $\frac{dz}{dx} = \frac{dz}{dy} * \frac{dy}{dx}$

Ví dụ cần tính đạo hàm $z(x) = (2x + 1)^2$, có thể thấy $z = f(g(x))$ trong đó $f(x) = x^2, g(x) = 2x + 1$. Do đó áp dụng chain rule ta có:

$$\frac{dz}{dx} = \frac{dz}{dy} * \frac{dy}{dx} = \frac{d(2x+1)^2}{d(2x+1)} * \frac{d(2x+1)}{dx} = 2 * (2x+1) * 2 = 4 * (2x+1)$$

Mọi người biết $\frac{dt^2}{dt} = 2t \Rightarrow \frac{d(2x+1)^2}{d(2x+1)} = 2(2x+1)$ bằng cách đặt $t = 2x+1$.

Có một cách dễ nhìn hơn để tính chain rule là dùng biểu đồ



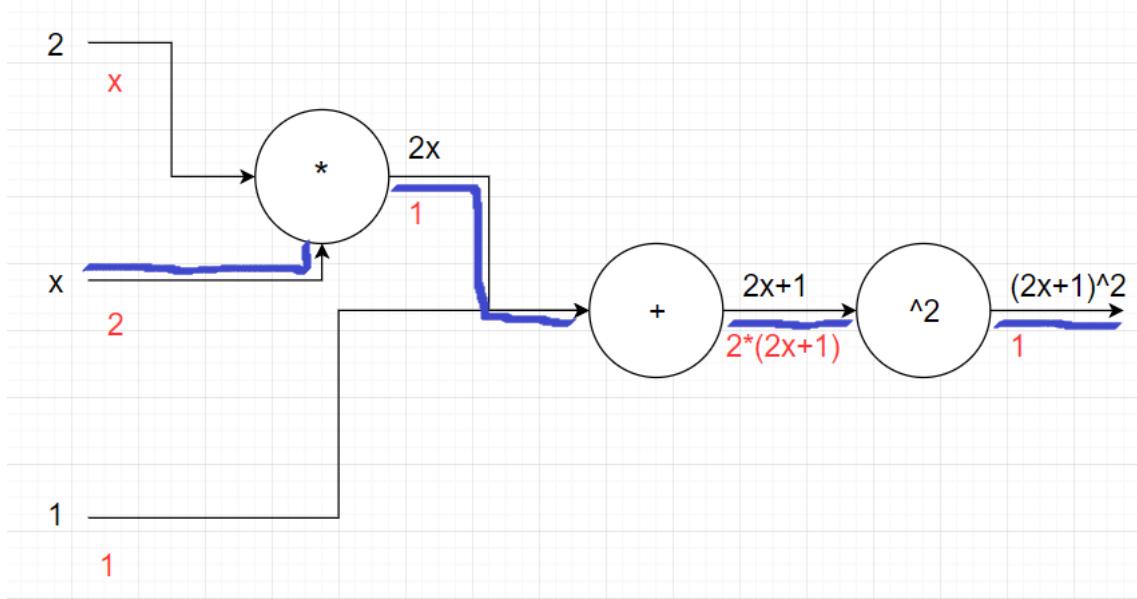
Hình 4.6: Graph cho chain rule

Số và biến ($2, x, 1$) viết ở bên trái và không có hình tròn bao quanh, các hình tròn là các phép tính ($*$, $+$, $\hat{2}$)

Giá trị của biểu thức sau khi thực hiện phép tính được viết màu đen ở phía trên đường sau phép tính. ví dụ, phép cộng của $2x$ và 1 có giá trị $2x+1$, phép bình phương $2x+1$ có giá trị là $(2x+1)\hat{2}$.

Giá trị của đạo hàm qua thực hiện phép tính được viết ở bên dưới với mực đỏ, ví dụ qua phép bình phương, $\frac{d(2x+1)^2}{d(2x+1)} = 2(2x+1)$, hay qua phép cộng $\frac{d(2x+1)}{d(2x)} = 1$ và qua phép nhân $\frac{d(2x)}{d(x)} = 2$.

Ở bước cuối cùng đạo hàm được viết là 1 (có hay không cũng được vì nhân với 1 vẫn thế nhưng để bạn biết đây là điểm kết thúc). Giờ cần tính $\frac{d(2x+1)^2}{dx}$ thì bạn nhân tất cả các giá trị màu đỏ trên đường đi từ x đến $(2x+1)^2$



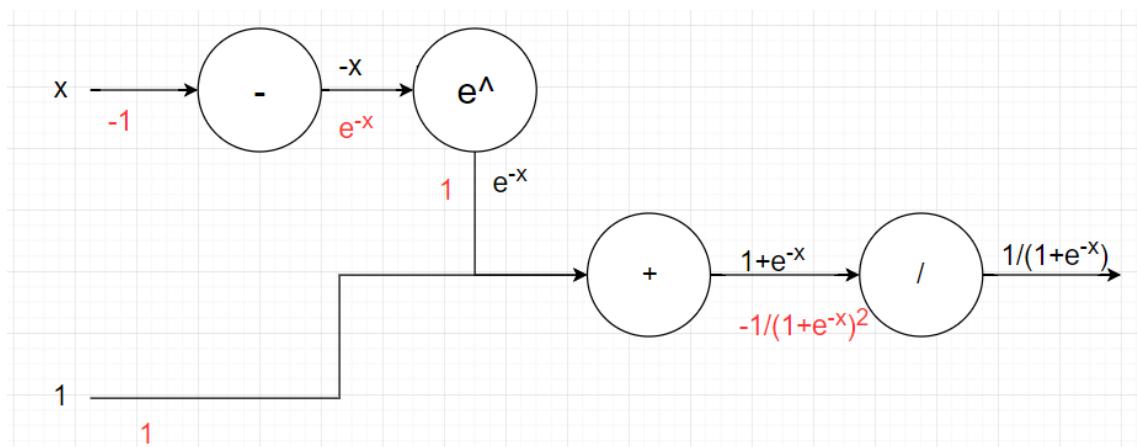
Do đó $\frac{d(2x+1)^2}{dx} = 2 * (2x+1) * 2 * 1 = 4 * (2x+1)$

Thực ra nếu bạn để ý biểu đồ chính là chain rule: $\frac{d(2x+1)^2}{dx} = \frac{d(2x+1)^2}{d(2x+1)} * \frac{d(2x+1)}{d(2x)} * \frac{d(2x)}{dx} = 2 * (2x+1) * 2 = 4 * (2x+1)$

Chỉ là biểu đồ dễ nhìn hơn.

Thử áp dụng tính đạo hàm của hàm sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$.

Nhắc lại kiến thức đạo hàm cơ bản $\frac{d(\frac{1}{x})}{dx} = \frac{-1}{x^2}$, $\frac{d(e^x)}{dx} = e^x$

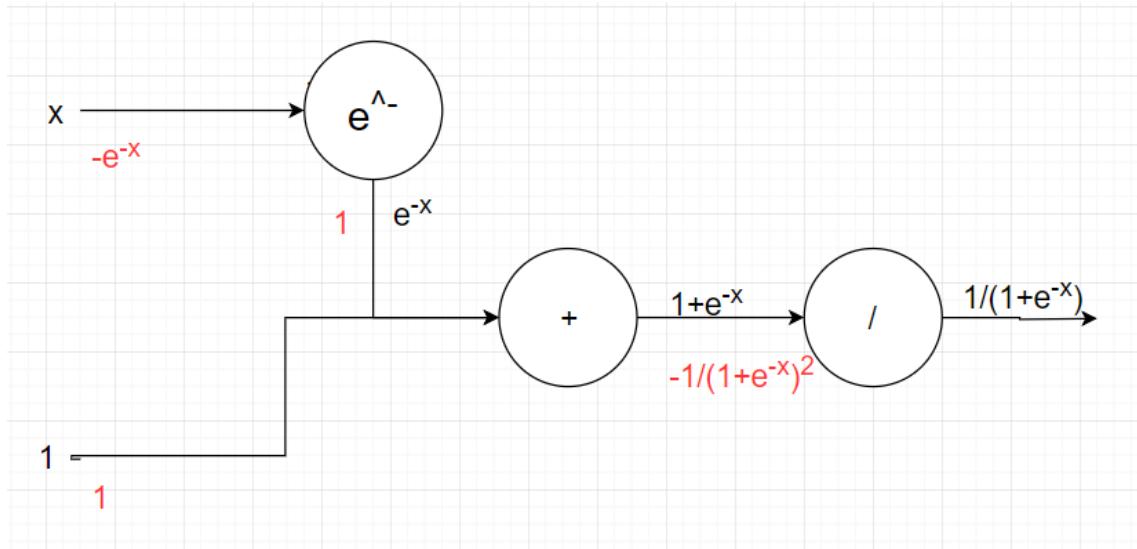


Do đó:

$$\frac{d(\sigma(x))}{dx} = \frac{d(\frac{1}{1+e^{-x}})}{dx} = \frac{-1}{(1+e^{-x})^2} * 1 * e^{-x} * (-1) = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \frac{1}{1+e^{-x}} * \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}} * \left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(x) * (1 - \sigma(x))$$

Thực ra mọi người không cần vẽ đồ thị quá chi tiết như trên chỉ dùng để tách phép tính khi mà nó phức tạp



4.5.1 Áp dụng gradient descent

Để áp dụng thuật toán gradient descent tìm tối ưu loss function mình cần tính đạo hàm của loss function với w.

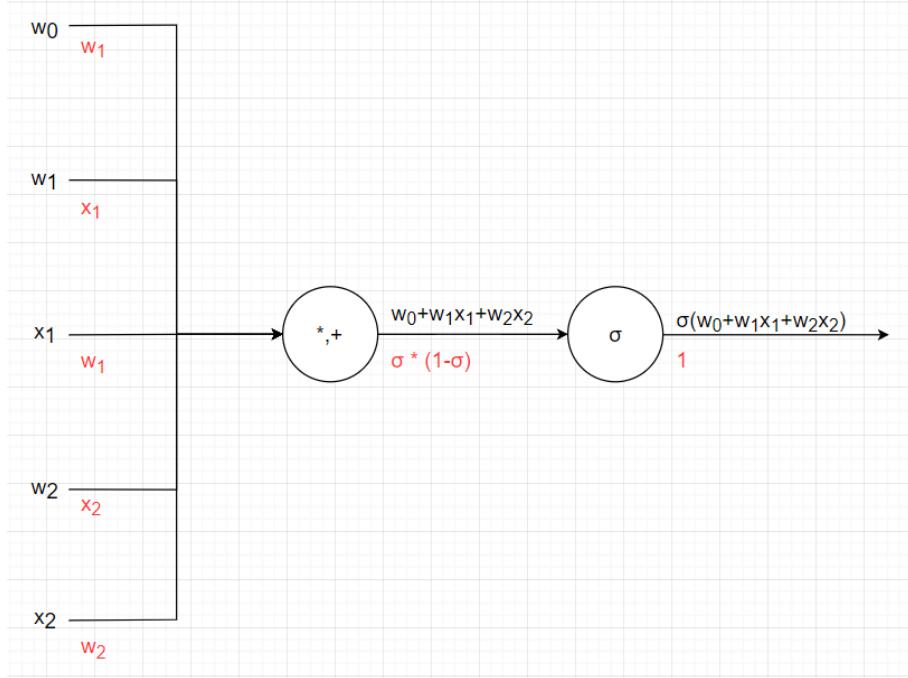
Với mỗi điểm $(x^{(i)}, y_i)$, gọi hàm loss function

$L = -(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$ trong đó $\hat{y}_i = \sigma(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}) = \sigma(z)$ là giá trị mà model dự đoán, còn y_i là giá trị thật của dữ liệu, $z = w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}$.

Nhắc lại đạo hàm cơ bản, $\frac{d(\log(x))}{dx} = \frac{1}{x}$

Áp dụng chain rule ta có: $\frac{dL}{dw_0} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dz} * \frac{dz}{dw_0}$

$$\frac{dL}{d\hat{y}_i} = -\frac{d(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))}{d\hat{y}_i} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) = \frac{\hat{y}_i - y_i}{\hat{y}_i * (1 - \hat{y}_i)} (*)$$



$$\text{Từ đó thi ta thấy: } \frac{d\hat{y}_i}{dw_0} = \frac{d(\sigma(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}))}{dw_0} = \hat{y}_i * (1 - \hat{y}_i)$$

$$\frac{d\hat{y}_i}{dw_1} = \frac{d(\sigma(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}))}{dw_1} = x_1^{(i)} * \hat{y}_i * (1 - \hat{y}_i)$$

$$\frac{d\hat{y}_i}{dw_2} = \frac{d(\sigma(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}))}{dw_2} = x_2^{(i)} * \hat{y}_i * (1 - \hat{y}_i)$$

Do đó

$$\frac{dL}{dw_0} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dw_0} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right) * \hat{y}_i * (1 - \hat{y}_i) = -(y_i * (1 - \hat{y}_i) - (1 - y_i) * \hat{y}_i) = \hat{y}_i - y_i$$

Tương tự

$$\begin{aligned} \frac{dL}{dw_1} &= x_1^{(i)} * (\hat{y}_i - y_i) \\ \frac{dL}{dw_2} &= x_2^{(i)} * (\hat{y}_i - y_i) \end{aligned}$$

Đây là trên một điểm dữ liệu, trên toàn bộ dữ liệu

$$\frac{dL}{dw_0} = \frac{1}{N} * \sum_{i=1}^N (\hat{y}_i - y_i)$$

$$\frac{dL}{dw_1} = \frac{1}{N} * \sum_{i=1}^N x_1^{(i)} * (\hat{y}_i - y_i)$$

$$\frac{dL}{dw_2} = \frac{1}{N} * \sum_{i=1}^N x_2^{(i)} * (\hat{y}_i - y_i)$$

4.5.2 Biểu diễn bài toán dưới ma trận

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ \dots & \dots & \dots \\ 1 & x_1^{(n)} & x_2^{(n)} \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix}, w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

$$\hat{y} = \sigma(X * W)$$

$$J = -\frac{1}{N} * \text{sum}(y \otimes \log(\hat{y}) + (1-y) \otimes \log(1-\hat{y}))$$

$$\frac{dJ}{dw} = \frac{1}{N} * X^T * (\hat{y} - y)$$

Nếu mọi người thấy các công thức biểu diễn dưới ma trận vẫn lạ lạ thì nên xem lại bài 1 và lấy giấy bút tự tính và biểu diễn lại.

Sau khi thực hiện thuật toán gradient descent ta sẽ tìm được w_0, w_1, w_2 . Với mỗi hồ sơ mới $x^{(t)}$ ta sẽ tính được phần trăm nêu cho vay $\hat{y}_t = \sigma(w_0 + w_1 * x_1^{(t)} + w_2 * x_2^{(t)})$ rồi so sánh với ngưỡng cho vay của công ty t (bình thường là $t = 0.5$, thời kì thiết chặt thì là $t = 0.8$) nếu $\hat{y}_t \geq t$ thì cho vay, không thì không cho vay.

4.5.3 Hàm sigmoid được chọn để đạo hàm đẹp

$$\text{Ta có: } \frac{dL}{dw_0} = \frac{dL}{d\hat{y}_i} * \frac{d\hat{y}_i}{dz} * \frac{dz}{dw_0}$$

$$\frac{dL}{d\hat{y}_i} = -\frac{d(y_i * \log(\hat{y}_i) + (1-y_i) * \log(1-\hat{y}_i))}{d\hat{y}_i} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}\right) = \frac{\hat{y}_i - y_i}{\hat{y}_i * (1-\hat{y}_i)} (*)$$

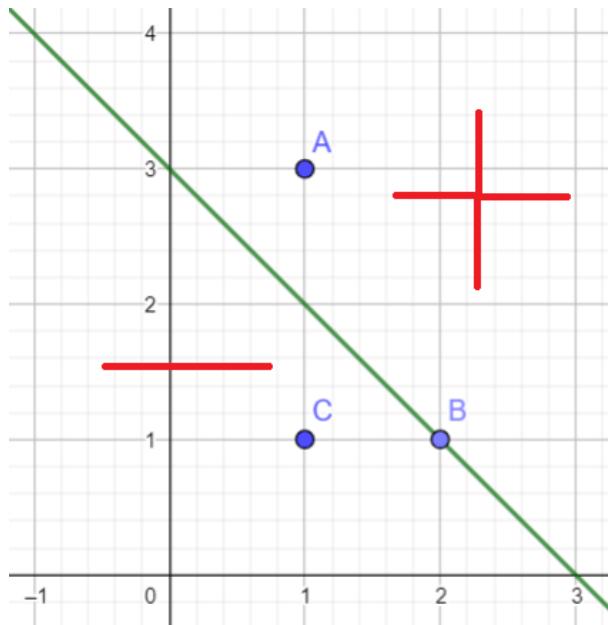
Ta có $\hat{y} = \theta(z)$ nên ta cần tìm 1 hàm $\theta(z)$ có đạo hàm $\frac{\partial \hat{y}}{\partial z}$ triệt tiêu mẫu số biểu thức (*):

$$\begin{aligned} &\Rightarrow \frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y}) \\ &\Leftrightarrow \frac{\partial \hat{y}}{\hat{y}(1-\hat{y})} = \partial z \\ &\Leftrightarrow \int \frac{\partial \hat{y}}{\hat{y}(1-\hat{y})} = \int \partial z \\ &\Leftrightarrow \int \left(\frac{1}{\hat{y}} + \frac{1}{1-\hat{y}}\right) \partial \hat{y} = \int \partial z \\ &\Leftrightarrow \log \hat{y} - \log(1-\hat{y}) = z \\ &\Leftrightarrow \log \frac{\hat{y}}{1-\hat{y}} = z \\ &\Leftrightarrow \frac{\hat{y}}{1-\hat{y}} = e^z \\ &\Leftrightarrow \hat{y} = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}} = \sigma(z) \end{aligned}$$

Mọi người có thể thấy hàm sigmoid ngoài việc liên tục có đạo hàm tại mọi điểm, cho giá trị trong khoảng $(0, 1)$ thì còn làm đạo hàm loss function đẹp hơn.

4.6 Quan hệ giữa phần trăm và đường thẳng

Xét đường thẳng $y = ax + b$, gọi $f(x,y) = y - (ax + b)$ thì đường thẳng chia mặt phẳng thành 2 miền, 1 miền hàm f có giá trị dương, 1 miền hàm f có giá trị âm và giá trị f các điểm trên đường thẳng bằng 0.



Hình 4.7: Đường thẳng $y = 3 - x$

Ví dụ, xét đường thẳng $y = 3 - x$, hàm $f(x,y) = y - 3 + x$.

- Tại điểm A(1,3) $\Rightarrow f(1,3) = 3 - 3 + 1 = 1 > 0$
- Tại điểm B(2,1) nằm trên đường thẳng $\Rightarrow f(2,1) = 2 - 3 + 1 = 0$
- Tại điểm C(1,1) $\Rightarrow f(1,1) = 1 - 3 + 1 = -1 < 0$

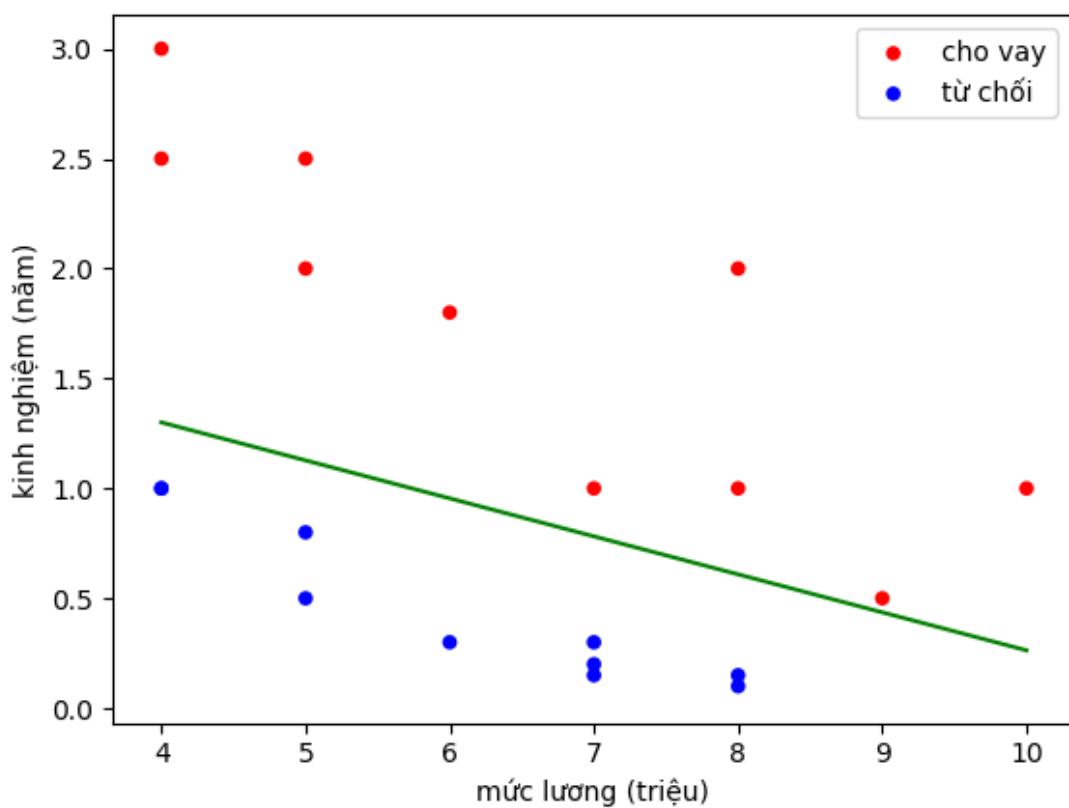
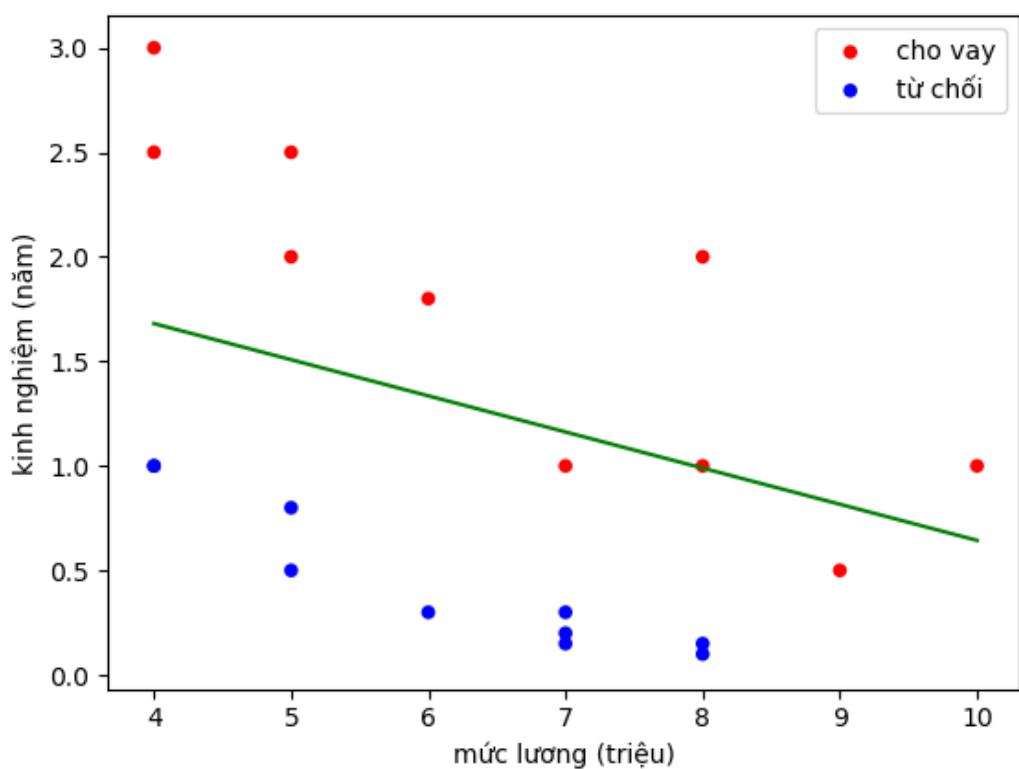
Giả sử bạn lấy mốc ở chính giữa là 50% tức là nếu hồ sơ mới dự đoán $\hat{y}_i \geq 0.5$ thì cho vay, còn nhỏ hơn 0.5 thì không cho vay.

$$\begin{aligned}\hat{y}_i \geq 0.5 &\Leftrightarrow \frac{1}{1 + e^{-(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)})}} \geq 0.5 \\ &\Leftrightarrow 2 \geq 1 + e^{-(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)})} \\ &\Leftrightarrow e^{-(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)})} \leq 1 = e^0 \\ &\Leftrightarrow -(w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)}) \leq 0 \\ &\Leftrightarrow w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)} \geq 0\end{aligned}$$

$$\text{Tương tự } \hat{y}_i < 0.5 \Leftrightarrow w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)} < 0$$

\Rightarrow đường thẳng $w_0 + w_1 * x + w_2 * y = 0$ chính là đường phân cách giữa các điểm cho vay và từ chối.

Trong trường hợp tổng quát bạn lấy xác suất lớn hơn t ($0 < t < 1$) thì mới cho vay tiền $\hat{y}_i > t \Leftrightarrow w_0 + w_1 * x_1^{(i)} + w_2 * x_2^{(i)} > -\ln(\frac{1}{t} - 1)$

Hình 4.8: Đường phân chia $t = 0.5$ Hình 4.9: Đường phân chia của $t = 0.8$

Ví dụ $t = 0.8$, bạn thấy đường phân chia gần với điểm màu đỏ hơn so với $t = 0.5$ thậm chí một số hồ sơ cũ trước được cho vay nhưng nếu giờ nộp lại cũng từ chối. Đồng nghĩa với việc công ty thắt chặt việc cho vay lại.

4.7 Ứng dụng

- Spam detection: Dự đoán mail gửi đến hộp thư của bạn có phải spam hay không.
- Credit card fraud: Dự đoán giao dịch ngân hàng có phải gian lận không.
- Health: Dự đoán 1 u là u lành hay u ác tính.
- Banking: Dự đoán khoản vay có trả được hay không.
- Investment: Dự đoán khoản đầu tư vào start-up có sinh lợi hay không.

4.8 Python code

```
# Thêm thư viện
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Hàm sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Load data từ file csv
data = pd.read_csv('dataset.csv').values
N, d = data.shape
x = data[:, 0:d-1].reshape(-1, d-1)
y = data[:, 2].reshape(-1, 1)

# Vẽ data bằng scatter
plt.scatter(x[:10, 0], x[:10, 1], c='red', edgecolors='none', s=30, label='cho vay')
plt.scatter(x[10:, 0], x[10:, 1], c='blue', edgecolors='none', s=30, label='từ chối')
plt.legend(loc=1)
plt.xlabel('mức lương (triệu)')
plt.ylabel('kinh nghiệm (năm)')

# Thêm cột 1 vào dữ liệu x
x = np.hstack((np.ones((N, 1)), x))

w = np.array([0., 0.1, 0.1]).reshape(-1, 1)

# Số lần lặp bước 2
numOfIteration = 1000
cost = np.zeros((numOfIteration, 1))
learning_rate = 0.01

for i in range(1, numOfIteration):

    # Tính giá trị dự đoán
    y_predict = sigmoid(np.dot(x, w))
```

```
cost[i] = -np.sum(np.multiply(y, np.log(y_predict)) + \
                  np.multiply(1-y, np.log(1-y_predict)))
# Gradient descent
w = w - learning_rate * np.dot(x.T, y_predict-y)
print(cost[i])

# Vẽ đường phân cách.
t = 0.5
plt.plot((4, 10),(-(w[0]+4*w[1]+ np.log(1/t-1))/w[2], -(w[0] + 10*w[1]+ \
                  np.log(1/t-1))/w[2]), 'g')
plt.show()

# Lưu weight dùng numpy.save(), định dạng '.npy'
np.save('weight logistic.npy', w)
# Load weight từ file '.npy'
w = np.load('weight logistic.npy')

# Logistic Regression dùng thư viện sklearn
from sklearn.linear_model import LogisticRegression

# Load data từ file csv
data = pd.read_csv('dataset.csv').values
N, d = data.shape
x = data[:, 0:d-1].reshape(-1, d-1)
y = data[:, 2].reshape(-1, 1)

# Vẽ data bằng scatter
plt.scatter(x[:10, 0], x[:10, 1], c='red', edgecolors='none', s=30, label='cho vay')
plt.scatter(x[10:, 0], x[10:, 1], c='blue', edgecolors='none', s=30, label='từ chối')
plt.legend(loc=1)
plt.xlabel('mức lương (triệu)')
plt.ylabel('kinh nghiệm (năm)')

# Tạo mô hình Logistic Regression và train
logreg = LogisticRegression()
logreg.fit(x, y)

# Lưu các biến của mô hình vào mảng
wg = np.zeros( (3, 1) )
wg[0, 0] = logreg.intercept_
wg[1:, 0] = logreg.coef_

# Vẽ đường phân cách
t = 0.5
plt.plot((4, 10),(-(wg[0]+4*[1]+ np.log(1/t-1))/wg[2], \
                  -(wg[0] + 10*wg[1]+ np.log(1/t-1))/wg[2]), 'g')
plt.show()

# Lưu các tham số dùng numpy.savetxt(), định dạng '.npz'
```

```

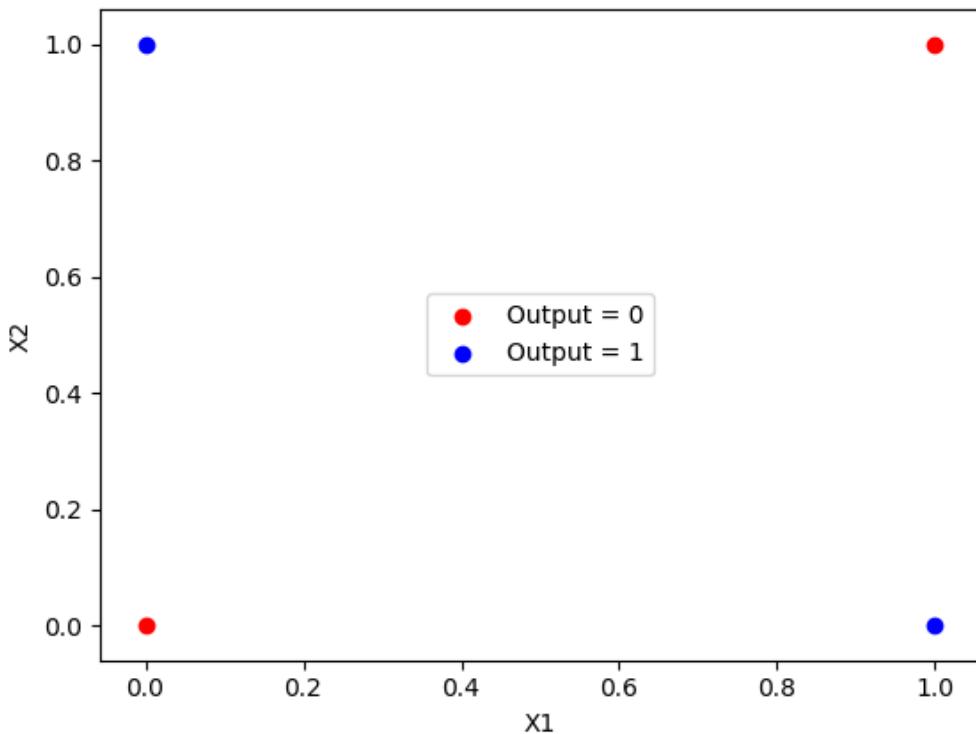
np.savez('w logistic.npz', a=logreg.intercept_, b=logreg.coef_)
# Load các tham số dùng numpy.load(), file '.npz'
k = np.load('w logistic.npz')
logreg.intercept_ = k['a']
logreg.coef_ = k['b']

```

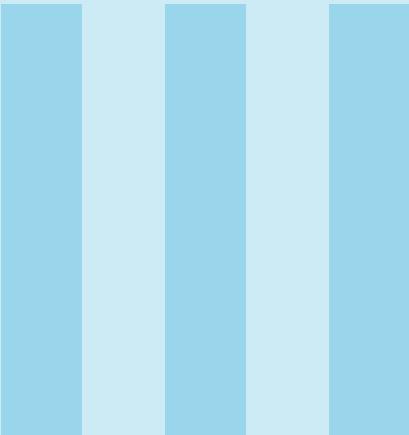
4.9 Bài tập

- Tại sao hàm sigmoid được chọn trong bài toán logistic regression? Dùng chain rule tính đạo hàm hàm sigmoid.
- Dùng chain rule tính đạo hàm loss function của logistic regression với từng biến.
- Biểu diễn dưới dạng ma trận cho bài toán logistic regression.
- Chỉnh 1 số tham số như learning_rate (tăng, giảm), số iteration xem loss function sẽ thay đổi thế nào.
- Dùng logistic regression giải bài toán XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



Hình 4.10: Bài toán XOR



Neural Network

5	Neural network	83
5.1	Neural network là gì	
5.2	Mô hình neural network	
5.3	Feedforward	
5.4	Logistic regression với toán tử XOR	
5.5	Bài tập	
6	Backpropagation	97
6.1	Bài toán XOR với neural network	
6.2	Mô hình tổng quát	
6.3	Python code	
6.4	Bài tập	

5. Neural network

Bài trước học về thuật toán logistic regression với giá trị đầu ra là giá trị nhị phân. Tuy nhiên, logistic regression là một mô hình neural network đơn giản, bài này sẽ học mô hình neural network đầy đủ.

Bạn nên hoàn thành 2 bài trước linear regression và logistic regression trước khi vào bài này. Trong bài này có khá nhiều kí hiệu và công thức, nên bạn nên chuẩn bị giấy bút để bắt đầu.

5.1 Neural network là gì

Con chó có thể phân biệt được người thân trong gia đình và người lạ hay đứa trẻ có thể phân biệt được các con vật. Những việc tưởng chừng như rất đơn giản nhưng lại cực kì khó để thực hiện bằng máy tính. Vậy sự khác biệt nằm ở đâu? Câu trả lời nằm ở cấu trúc bộ não với lượng lớn các nơ-ron thần kinh liên kết với nhau. Liệu máy tính có thể mô phỏng lại cấu trúc bộ não để giải các bài toán trên ???

Neural là tính từ của neuron (nơ-ron), network chỉ cấu trúc, cách các nơ-ron đó liên kết với nhau, nên neural network (NN) là một hệ thống tính toán lấy cảm hứng từ sự hoạt động của các nơ-ron trong hệ thần kinh.

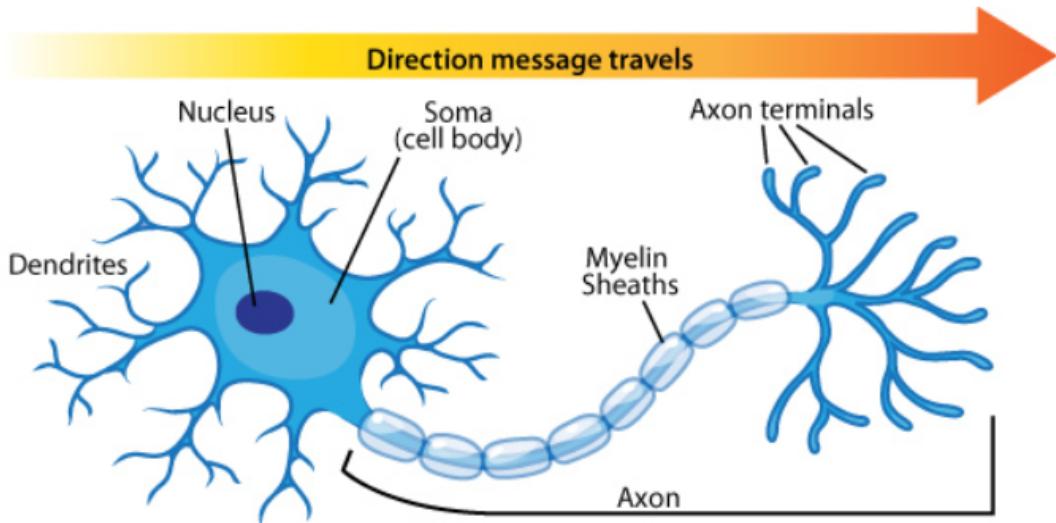
5.1.1 Hoạt động của các nơ-ron

Nơ-ron là đơn vị cơ bản cấu tạo hệ thống thần kinh và là thành phần quan trọng nhất của não. Đầu chúng ta gồm khoảng 10 triệu nơ-ron và mỗi nơ-ron lại liên kết với tầm 10.000 nơ-ron khác.

Ở mỗi nơ-ron có phần thân (soma) chứa nhân, các tín hiệu đầu vào qua sợi nhánh (dendrites) và các tín hiệu đầu ra qua sợi trực (axon) kết nối với các nơ-ron khác. Hiểu đơn giản mỗi nơ-ron nhận dữ liệu đầu vào qua sợi nhánh và truyền dữ liệu đầu ra qua sợi trực, đến các sợi nhánh của các nơ-ron khác.

Mỗi nơ-ron nhận xung điện từ các nơ-ron khác qua sợi nhánh. Nếu các xung điện này đủ lớn để kích hoạt nơ-ron, thì tín hiệu này đi qua sợi trực đến các sợi nhánh của các nơ-ron khác. => Ở

Neuron Anatomy



Hình 5.1: Tế bào thần kinh [14]

mỗi nơ-ron cần quyết định có kích hoạt nơ-ron đấy hay không. Tương tự các hoạt động của hàm sigmoid bài trước.

Tuy nhiên NN chỉ là lấy cảm hứng từ não bộ và cách nó hoạt động, chứ không phải bắt chước toàn bộ các chức năng của nó. Việc chính của chúng ta là dùng mô hình đấy để giải quyết các bài toán chúng ta cần.

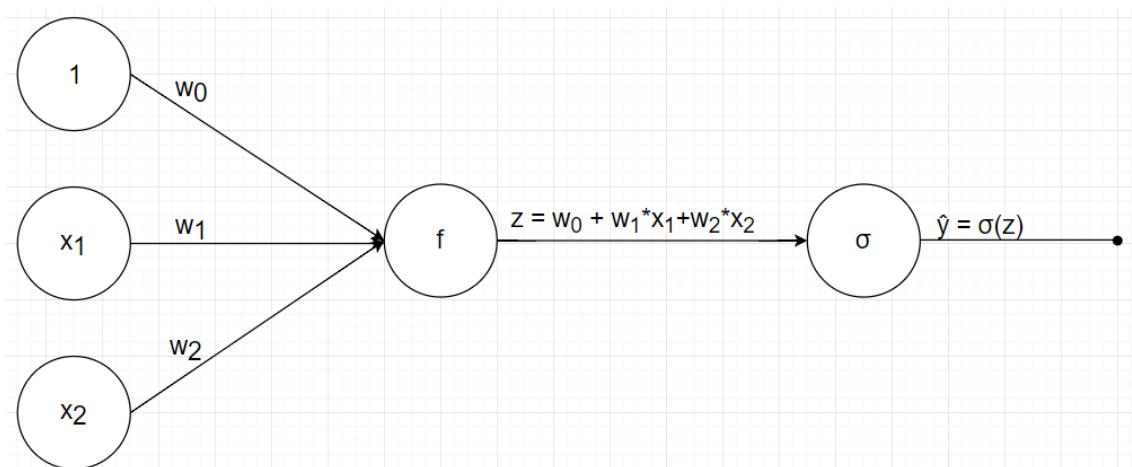
5.2 Mô hình neural network

5.2.1 Logistic regression

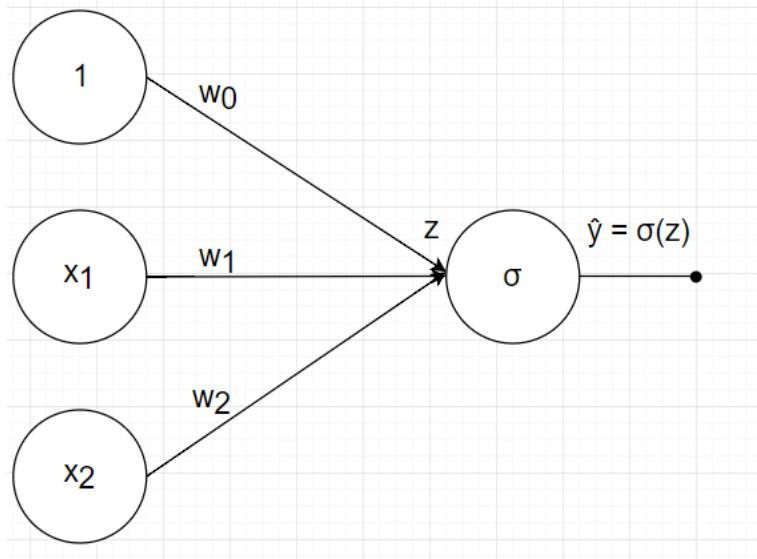
Logistic regression là mô hình neural network đơn giản nhất chỉ với input layer và output layer.

Mô hình của logistic regression từ bài trước là: $\hat{y} = \sigma(w_0 + w_1 * x_1 + w_2 * x_2)$. Có 2 bước:

- Tính tổng linear: $z = w_0 + x_1 * w_1 + x_2 * w_2$
- Áp dụng sigmoid function: $\hat{y} = \sigma(z)$



Để biểu diễn gọn lại ta sẽ gộp hai bước trên thành một trên biểu đồ như hình 5.2.



Hình 5.2: Mô hình logistic regression

Hệ số w_0 được gọi là bias. Để ý từ những bài trước đến giờ dữ liệu khi tính toán luôn được thêm 1 để tính hệ số bias w_0 . Tại sao lại cần hệ số bias? Quay lại với bài 1, phương trình đường thẳng sẽ thế nào nếu bỏ w_0 , phương trình giờ có dạng: $y = w_1 * x$, sẽ luôn đi qua gốc tọa độ và nó không tổng quát hóa phương trình đường thẳng nên có thể không tìm được phương trình mong muốn. => Việc thêm bias (hệ số tự do) là rất quan trọng.

Hàm sigmoid ở đây được gọi là activation function.

5.2.2 Mô hình tổng quát

Layer đầu tiên là input layer, các layer ở giữa được gọi là hidden layer, layer cuối cùng được gọi là output layer. Các hình tròn được gọi là node.

Mỗi mô hình luôn có 1 input layer, 1 output layer, có thể có hoặc không các hidden layer. Tổng số layer trong mô hình được quy ước là số layer - 1 (không tính input layer).

Ví dụ như ở hình trên có 1 input layer, 2 hidden layer và 1 output layer. Số lượng layer của mô hình là 3 layer.

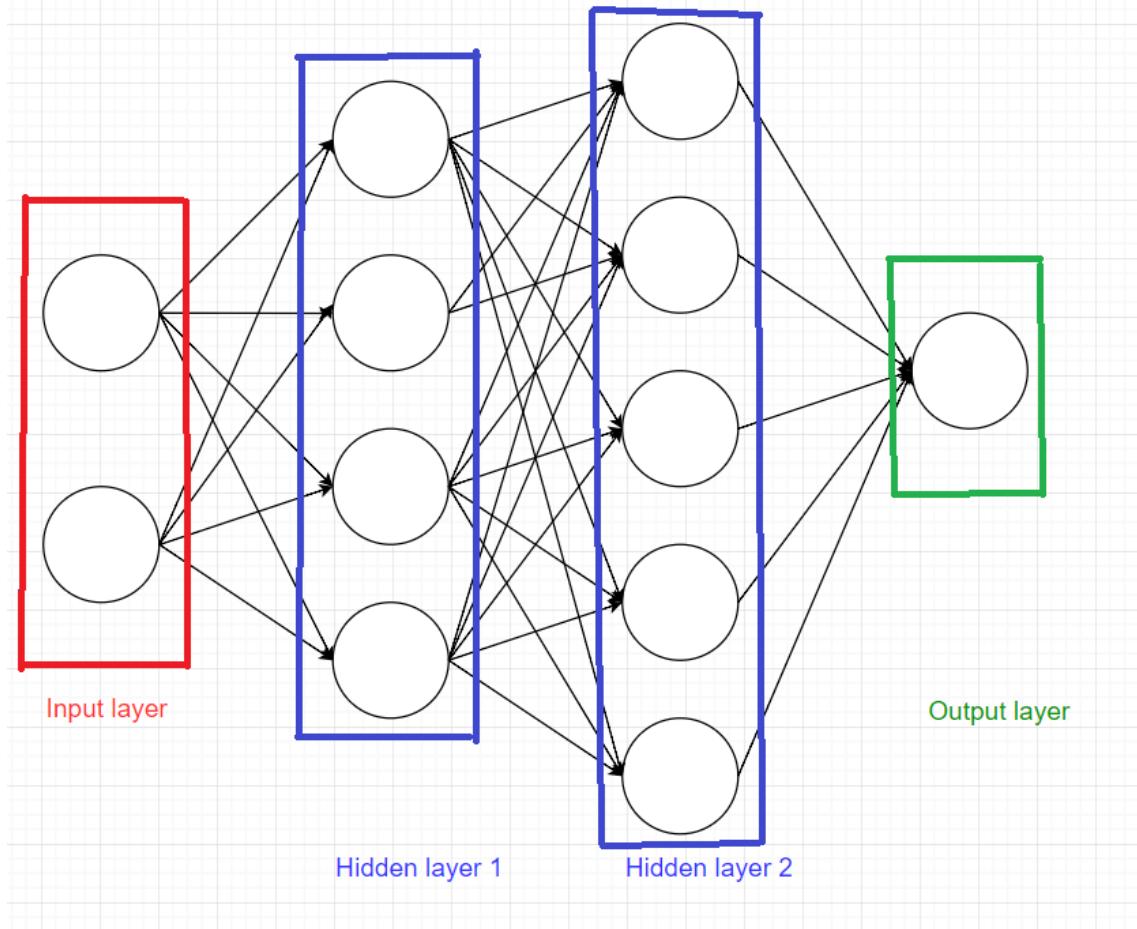
Mỗi node trong hidden layer và output layer :

- Liên kết với tất cả các node ở layer trước đó với các hệ số w riêng.
- Mỗi node có 1 hệ số bias b riêng.
- Diễn ra 2 bước: tính tổng linear và áp dụng activation function.

5.2.3 Kí hiệu

Số node trong hidden layer thứ i là $l^{(i)}$.

Ma trận $W^{(k)}$ kích thước $l^{(k-1)} * l^{(k)}$ là ma trận hệ số giữa layer $(k-1)$ và layer k , trong đó $w_{ij}^{(k)}$ là hệ số kết nối từ node thứ i của layer $k-1$ đến node thứ j của layer k .



Hình 5.3: Mô hình neural network

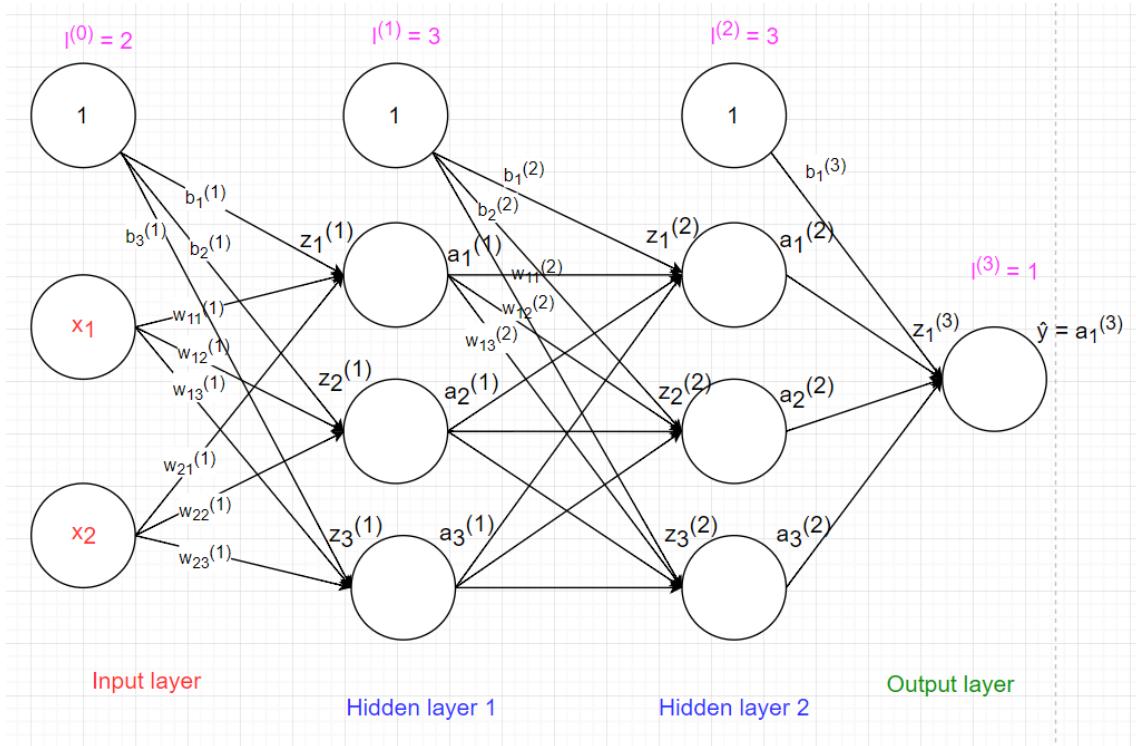
Vector $b^{(k)}$ kích thước $l^k * 1$ là hệ số bias của các node trong layer k, trong đó $b_i^{(k)}$ là bias của node thứ i trong layer k.

Với node thứ i trong layer 1 có bias $b_i^{(l)}$ thực hiện 2 bước:

- Tính tổng linear: $z_i^{(l)} = \sum_{j=1}^{l^{(l-1)}} a_j^{(l-1)} * w_{ji}^{(l)} + b_i^{(l)}$, là tổng tất cả các node trong layer trước nhân với hệ số w tương ứng, rồi cộng với bias b.
- Áp dụng activation function: $a_i^{(l)} = \sigma(z_i^{(l)})$

Vector $z^{(k)}$ kích thước $l^{(k)} * 1$ là giá trị các node trong layer k sau bước tính tổng linear.

Vector $a^{(k)}$ kích thước $l^{(k)} * 1$ là giá trị của các node trong layer k sau khi áp dụng hàm activation function.



Mô hình neural network trên gồm 3 layer. Input layer có 2 node ($l^{(0)} = 2$), hidden layer 1 có 3 node, hidden layer 2 có 3 node và output layer có 1 node.

Do mỗi node trong hidden layer và output layer đều có bias nên trong input layer và hidden layer cần thêm node 1 để tính bias (nhưng không tính vào tổng số node layer có).

Tại node thứ 2 ở layer 1, ta có:

- $z_2^{(1)} = x_1 * w_{12}^{(1)} + x_2 * w_{22}^{(1)} + b_2^{(1)}$
- $a_2^{(1)} = \sigma(z_2^{(1)})$

Hay ở node thứ 3 layer 2, ta có:

- $z_3^{(2)} = a_1^{(1)} * w_{13}^{(2)} + a_2^{(1)} * w_{23}^{(2)} + a_3^{(1)} * w_{33}^{(2)} + b_3^{(2)}$
- $a_3^{(2)} = \sigma(z_3^{(2)})$

5.3 Feedforward

Để nhất quán về mặt ký hiệu, gọi input layer là $a^{(0)} (= x)$ kích thước $2*1$.

$$\begin{aligned}
 z^{(1)} &= \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix} = \begin{bmatrix} a_1^{(0)} * w_{11}^{(1)} + a_2^{(0)} * w_{21}^{(1)} + a_3^{(0)} * w_{31}^{(1)} + b_1^{(1)} \\ a_1^{(0)} * w_{12}^{(1)} + a_2^{(0)} * w_{22}^{(1)} + a_3^{(0)} * w_{32}^{(1)} + b_2^{(1)} \\ a_1^{(0)} * w_{13}^{(1)} + a_2^{(0)} * w_{23}^{(1)} + a_3^{(0)} * w_{33}^{(1)} + b_3^{(1)} \end{bmatrix} \\
 &= (W^{(1)})^T * a^{(0)} + b^{(1)}
 \end{aligned}$$

$$a^{(1)} = \sigma(z^{(1)})$$

Tương tự ta có:

$$\begin{aligned} z^{(2)} &= (W^{(2)})^T * a^{(1)} + b^{(2)} \\ a^{(2)} &= \sigma(z^{(2)}) \\ z^{(3)} &= (W^{(3)})^T * a^{(2)} + b^{(3)} \\ \hat{y} &= a^{(3)} = \sigma(z^{(3)}) \end{aligned}$$



Hình 5.4: Feedforward

5.3.1 Biểu diễn dưới dạng ma trận

Tuy nhiên khi làm việc với dữ liệu ta cần tính dự đoán cho nhiều dữ liệu một lúc, nên gọi X là ma trận $n \times d$, trong đó n là số dữ liệu và d là số trường trong mỗi dữ liệu, trong đó $x_j^{[i]}$ là giá trị trường dữ liệu thứ j của dữ liệu thứ i . Ví dụ dataset bài trước

Lương	Thời gian làm việc
10	1
5	2
7	0.15
6	1.8

thì $n = 4, d = 2, x_1^{[1]} = 10, x_2^{[1]} = 1, x_1^{[3]} = 6, x_2^{[2]} = 2$.

$$X = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_d^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_d^{[2]} \\ \dots & \dots & \dots & \dots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_d^{[n]} \end{bmatrix} = \begin{bmatrix} -(x^{[1]})^T - \\ -(x^{[2]})^T - \\ \dots \\ -(x^{[n]})^T - \end{bmatrix}$$

Hình 5.5: Biểu diễn dạng ma trận của nhiều dữ liệu trong dataset

Do $x^{[1]}$ là vector kích thước $d \times 1$ tuy nhiên ở X mỗi dữ liệu được viết theo hàng nên cần transpose $x^{[1]}$ thành kích thước $1 \times d$, kí hiệu: $-(x^{[1]})^T -$

Gọi ma trận $Z^{(i)}$ kích thước $N \times l^{(i)}$ trong đó $z_j^{(i)[k]}$ là giá trị thứ j trong layer i sau bước tính tổng linear của dữ liệu thứ k trong dataset.

*** Kí hiệu (i) là layer thứ i và kí hiệu $[k]$ là dữ liệu thứ k trong dataset.

Tương tự, gọi ma trận $A^{(i)}$ kích thước $N \times l^{(i)}$ trong đó $a_j^{(i)[k]}$ là giá trị thứ j trong layer i sau khi áp dụng activation function của dữ liệu thứ k trong dataset.

$$Z^{(i)} = \begin{bmatrix} z_1^{(i)[1]} & z_2^{(i)[1]} & \dots & z_{l^{(i)}}^{(i)[1]} \\ z_1^{(i)[2]} & z_2^{(i)[2]} & \dots & z_{l^{(i)}}^{(i)[2]} \\ \dots & \dots & \dots & \dots \\ z_1^{(i)[n]} & z_2^{(i)[n]} & \dots & z_{l^{(i)}}^{(i)[n]} \end{bmatrix} = \begin{bmatrix} -(z^{(i)[1]})^T \\ -(z^{(i)[2]})^T \\ \dots \\ -(z^{(i)[n]})^T \end{bmatrix}$$

Do đó

$$\begin{aligned} Z^{(1)} &= \begin{bmatrix} (z^{(1)[1]})^T \\ (z^{(1)[2]})^T \\ \dots \\ (z^{(1)[n]})^T \end{bmatrix} = \begin{bmatrix} (x^{[1]})^T * w^{(1)} + (b^{(1)})^T \\ (x^{[2]})^T * w^{(1)} + (b^{(1)})^T \\ \dots \\ (x^{[n]})^T * w^{(1)} + (b^{(1)})^T \end{bmatrix} = \\ X * W^{(1)} + \begin{bmatrix} (b^{(1)})^T \\ (b^{(1)})^T \\ \dots \\ (b^{(1)})^T \end{bmatrix} &= X * W^{(1)} + b^{(1)} \end{aligned}$$

Hình 5.6: Phép tính cuối cùng không đúng nhưng để viết công thức cho gọn lại.

$$A^{(1)} = \sigma(Z^{(1)})$$

$$Z^{(2)} = A^{(1)} * W^{(2)}, A^{(2)} = \sigma(Z^{(2)})$$

$$Z^{(3)} = A^{(2)} * W^{(3)}$$

$$\hat{Y} = A^{(3)} = \sigma(Z^{(3)})$$

Vậy là có thể tính được giá trị dự đoán của nhiều dữ liệu một lúc dưới dạng ma trận.

Giờ từ input X ta có thể tính được giá trị dự đoán \hat{Y} , tuy nhiên việc chính cần làm là đi tìm hệ số W và b. Có thể nghĩ ngay tới thuật toán gradient descent và việc quan trọng nhất trong thuật toán gradient descent là đi tìm đạo hàm của các hệ số đối với loss function. Và việc tính đạo hàm của các hệ số trong neural network được thực hiện bởi thuật toán backpropagation, sẽ được giới thiệu ở bài sau. Vì vì bài này có quá nhiều công thức sơ mài người rồi nên code sẽ được để ở bài sau.

5.4 Logistic regression với toán tử XOR

Phần này không bắt buộc, nó giúp giải thích việc có nhiều layer hơn thì mô hình sẽ giải quyết được các bài toán phức tạp hơn. Cụ thể là mô hình logistic regression bài trước không biểu diễn được toán

tử XOR nhưng nếu thêm 1 hidden layer với 2 node ở giữa input layer và output layer thì có thể biểu diễn được toán tử XOR.

AND, OR, XOR là các phép toán thực hiện phép tính trên bit. Thế bit là gì? bạn không cần quan tâm, chỉ cần biết mỗi bit nhận 1 trong 2 giá trị là 0 hoặc 1.

5.4.1 NOT

Phép tính NOT của 1 bit cho ra giá trị ngược lại.

A	NOT(A)
0	1
1	0

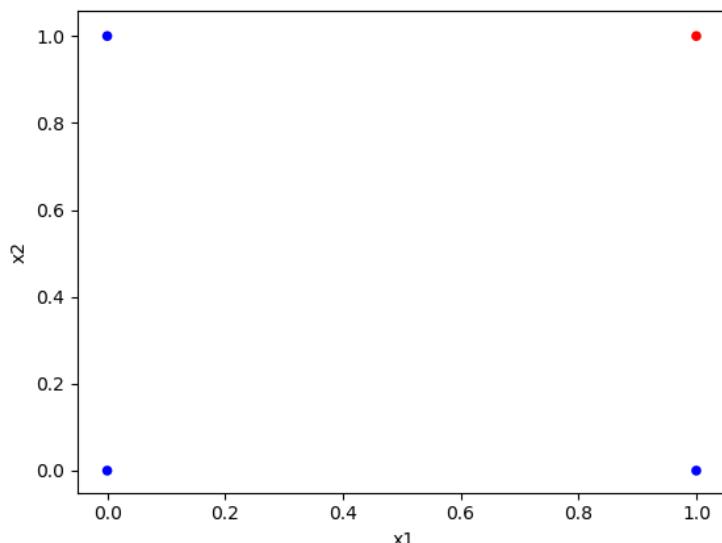
5.4.2 AND

Phép tính AND của 2 bit cho giá trị 1 nếu cả 2 bit bằng 1 và cho giá trị bằng 0 trong các trường hợp còn lại. Bảng chân lý

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

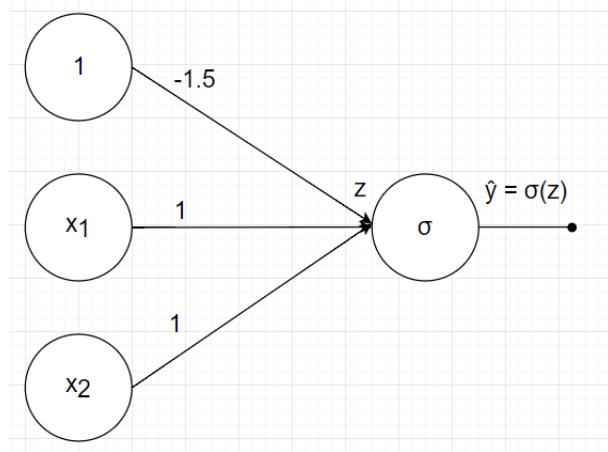
Giờ muốn máy tính học toán tử AND, ta thấy là kết quả là 0 và 1, nên nghĩ ngay đến logistic regression với dữ liệu

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

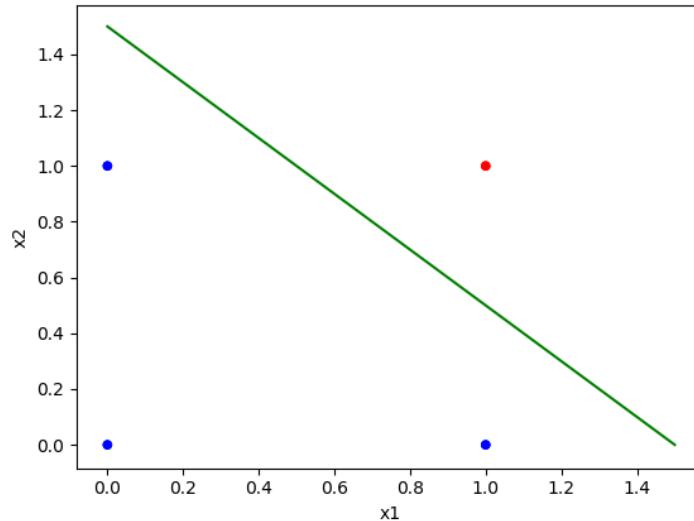


Hình 5.7: Chấm xanh là giá trị 0, chấm đỏ là giá trị 1

Theo bài trước, thì logistic regression chính là đường thẳng phân chia giữa các điểm nên áp dụng thuật toán trong bài logistic regression ta tìm được $w_0 = -1.5, w_1 = 1, w_2 = 1$

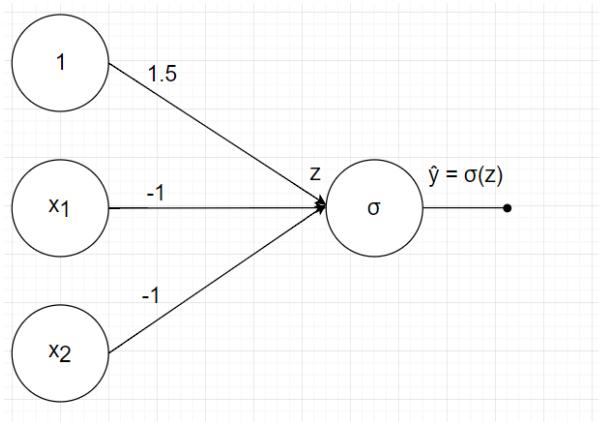


Hình 5.8: x_1 AND x_2



Hình 5.9: Đường thẳng $y = 1.5 - x$ phân chia 2 miền dữ liệu

Nhận xét, do phép tính NOT là đổi giá trị của bit, nên phép tính NOT(A AND B) có thể biểu diễn như hình trên với việc đổi màu các điểm từ đỏ thành xanh và xanh thành đỏ. Do đó đường phân chia không thay đổi và 2 miền giá trị đổi dấu cho nhau \Rightarrow giá trị các tham số đổi dấu $w_0 = 1.5, w_1 = -1, w_2 = -1$

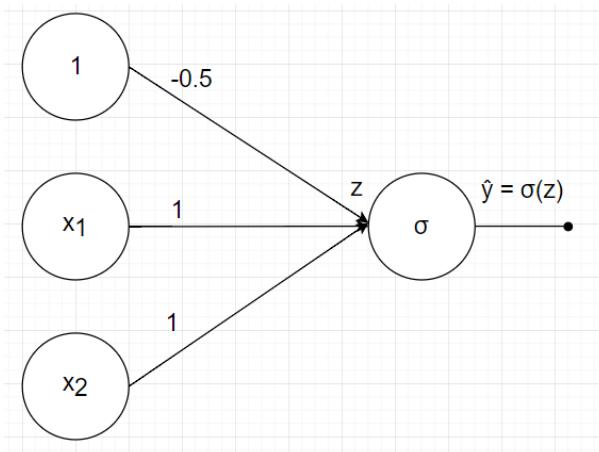
Hình 5.10: NOT (x_1 AND x_2)

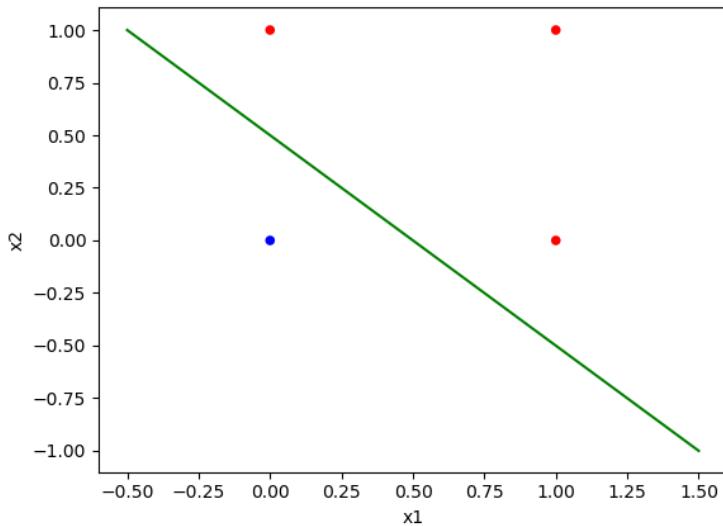
5.4.3 OR

Phép tính OR của 2 bit cho giá trị 1 nếu 1 trong 2 bit bằng 1 và cho giá trị bằng 0 trong các trường hợp còn lại. Bảng chân lý

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Tương tự ta cũng tìm được $w_0 = -0.5, w_1 = 1, w_2 = 1$

Hình 5.11: x_1 OR x_2

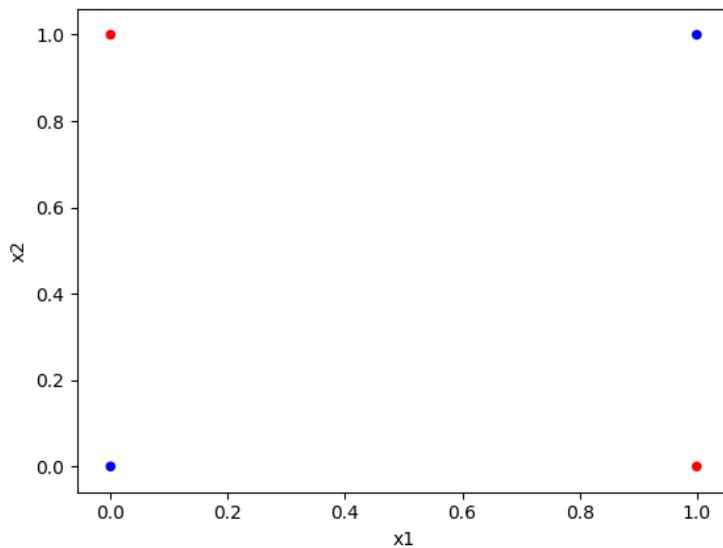


5.4.4 XOR

Phép tính XOR của 2 bit cho giá trị 1 nếu đúng 1 trong 2 bit bằng 1 và cho giá trị bằng 0 trong các trường hợp còn lại. Bảng chân lý

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Khi thiết lập bài toán logistic regression, ta có đồ thị

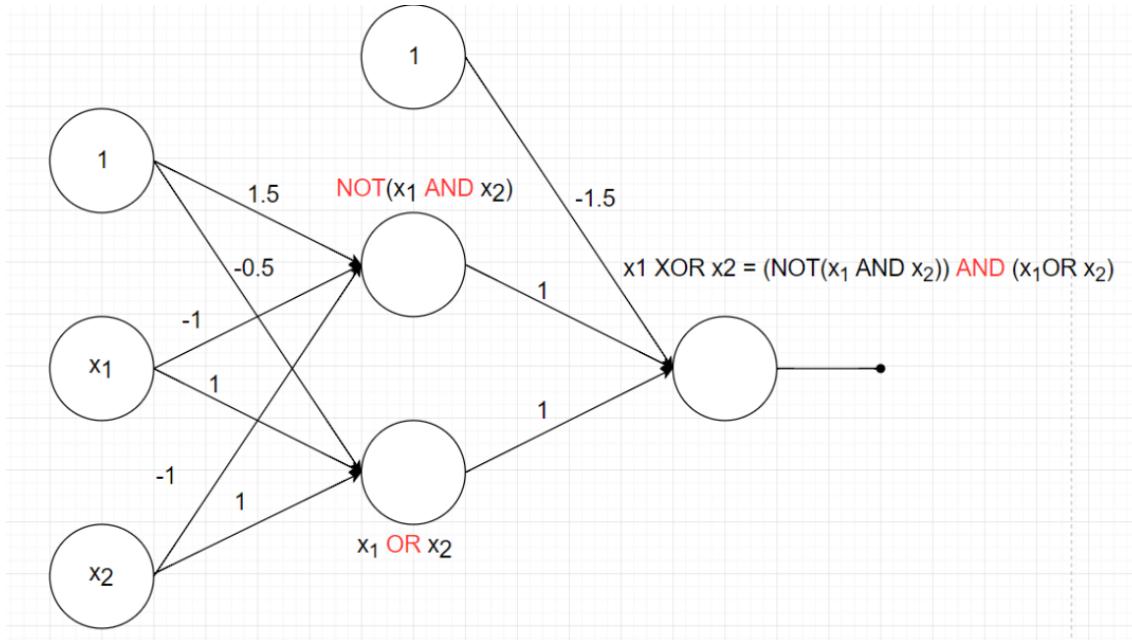


Rõ ràng là không thể dùng một đường thẳng để phân chia dữ liệu thành 2 miền. Nên khi bạn dùng gradient descent vào bài toán XOR thì bất kể bạn chạy bước 2 bao nhiêu lần hay chỉnh learning_rate thế nào thì vẫn không ra được kết quả như mong muốn. Logistic regression như bài trước không thể giải quyết được vấn đề này, giờ cần một giải pháp mới !!!

Áp dụng các kiến thức về bit ở trên lại, ta có:

A	B	A XOR B	A AND B	NOT (A AND B)	A OR B	(NOT(A AND B) AND (A OR B))
0	0	0	0	1	0	0
0	1	1	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	0	1	0

Do đó: $A \text{ XOR } B = (\text{NOT}(A \text{ AND } B) \text{ AND } (A \text{ OR } B))$, vậy để tính được XOR ta kết hợp NOT(AND) và OR sau đó tính phép tính AND.



Hình 5.12: Mô hình XOR

Nhìn có vẻ rõ ràng, cùng phân tích nhé:

- node $\text{NOT}(x_1 \text{ AND } x_2)$ chính là từ hình 5.10, với 3 mũi tên chỉ đến từ 1, x_1, x_2 với hệ số w_0, w_1, w_2 tương ứng là 1.5, -1, -1.
- node tính $x_1 \text{ OR } x_2$ là từ hình 5.11
- node trong output layer là phép tính AND từ 2 node của layer trước, giá trị hệ số từ hình 1 mang xuống.

Nhận xét: mô hình logistic regression không giải quyết được bài toán XOR nhưng mô hình mới thì giải quyết được bài toán XOR. Đâu là sự khác nhau:

- Logistic regression chỉ có input layer và output layer
- Mô hình mới có 1 hidden layer có 2 node ở giữa input layer và output layer.

=> **Càng nhiều layer và node thì càng giải quyết được các bài toán phức tạp hơn.**

5.5 Bài tập

1. (a) Tại sao hàm activation phải non-linear? Điều gì xảy ra nếu hàm linear activation được sử dụng?

- (b) Tính output 1 Neural Network đơn giản (2 nodes input layer, 2 nodes hidden layer, 1 node output layer) với hàm activation $f(x) = x + 1$ cho tất cả các node, tất cả các hệ số $w = 1$ và $b = 0$.
2. Tại sao cần nhiều layer và nhiều node trong 1 hidden layer?
 3. Code python cho mạng neural network với 1 hidden layer, sigmoid activation.

6. Backpropagation

Bài trước đã học về mô hình neural network và feedforward, giờ ta cần đi tìm hệ số W và b. Có thể nghĩ ngay tới thuật toán gradient descent và việc quan trọng nhất trong thuật toán gradient descent là đi tìm đạo hàm của các hệ số đối với loss function. Bài này sẽ tính đạo hàm của các hệ số trong neural network với thuật toán backpropagation.

Bạn nên hoàn thành bài neural network trước khi bắt đầu bài này và bài này là không bắt buộc để theo các bài tiếp theo trong sách.

6.1 Bài toán XOR với neural network

Bảng chân lý cho toán tử XOR

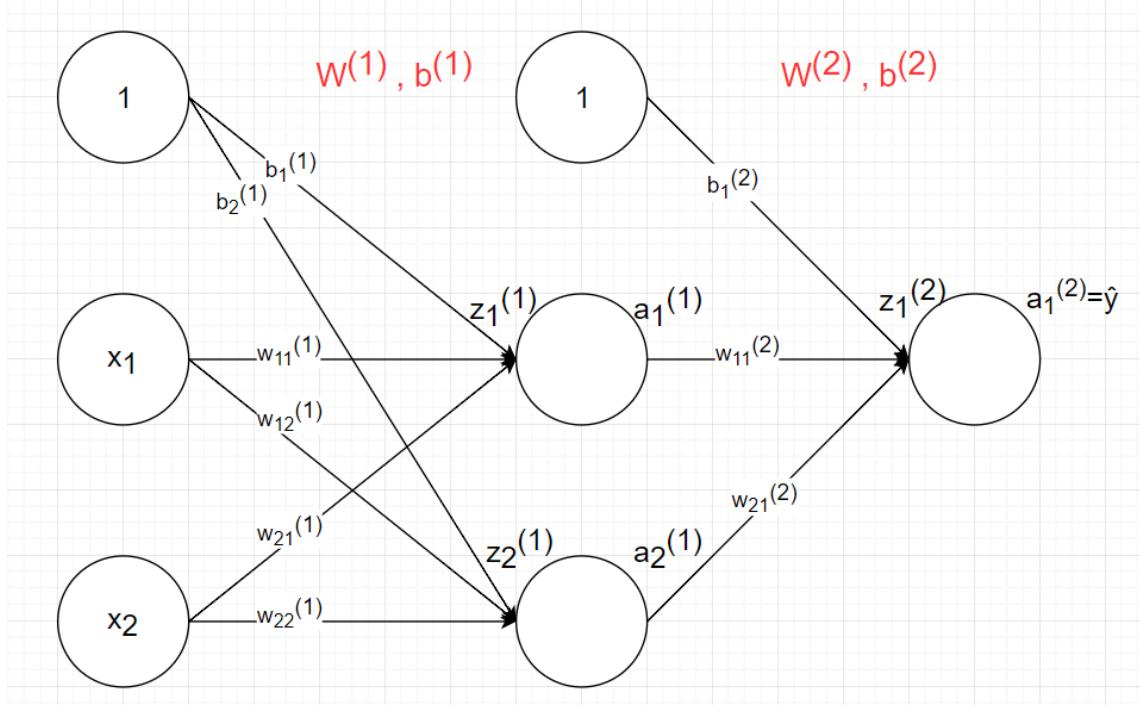
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Bài trước đã chứng minh là mô hình logistic regression trong bài 2 không thể biểu diễn được toán tử XOR. Để biểu diễn toán tử XOR ta cần thêm 1 hidden layer với 2 node.

6.1.1 Model

Nhắc lại kiến thức bài trước:

- Mô hình trên có 2 layer (số lượng layer của mô hình không tính input layer)
- Mô hình: 2-2-1, nghĩa là 2 node trong input layer, 1 hidden layer có 2 node và output layer có 1 node.
- Input layer và hidden layer luôn thêm node 1 để tính bias cho layer sau, nhưng không tính vào số lượng node trong layer
- Ở mỗi node trong hidden layer và output layer đều thực hiện 2 bước: tính tổng linear và áp dụng activation function.



- Các hệ số và bias tương ứng được ký hiệu như trong hình

Feedforward

- $z_1^{(1)} = b_1^{(1)} + x_1 * w_{11}^{(1)} + x_2 * w_{21}^{(1)}$
- $a_1^{(1)} = \sigma(z_1^{(1)})$
- $z_2^{(1)} = b_2^{(1)} + x_1 * w_{12}^{(1)} + x_2 * w_{22}^{(1)}$
- $a_2^{(1)} = \sigma(z_2^{(1)})$
- $z_1^{(2)} = b_1^{(2)} + a_1^{(1)} * w_{11}^{(2)} + a_2^{(1)} * w_{21}^{(2)}$
- $\hat{y} = a_1^{(2)} = \sigma(z_1^{(2)})$

Viết dưới dạng ma trận

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

$$Z^{(1)} = X * W^{(1)} + b^{(1)}$$

$$A^{(1)} = \sigma(Z^{(1)})$$

$$Z^{(2)} = A^{(1)} * W^{(2)} + b^{(2)}$$

$$\hat{Y} = A^{(2)} = \sigma(Z^{(2)})$$

6.1.2 Loss function

Hàm loss function vẫn dùng giống như trong bài 2

Với mỗi điểm $(x^{[i]}, y_i)$, gọi hàm loss function

$$L = -(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$$

Hàm loss function trên toàn bộ dữ liệu

$$J = -\sum_{i=1}^N (y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$$

6.1.3 Gradient descent

Để áp dụng gradient descent ta cần tính được đạo hàm của các hệ số W và bias b với hàm loss function.

*** Kí hiệu chuẩn về đạo hàm

- Khi hàm $f(x)$ là hàm 1 biến x , ví dụ: $f(x) = 2*x + 1$. Đạo hàm của f đối với biến x kí hiệu là $\frac{df}{dx}$
- Khi hàm $f(x, y)$ là hàm nhiều biến, ví dụ $f(x, y) = x^2 + y^2$. Đạo hàm f với biến x kí hiệu là $\frac{\partial f}{\partial x}$

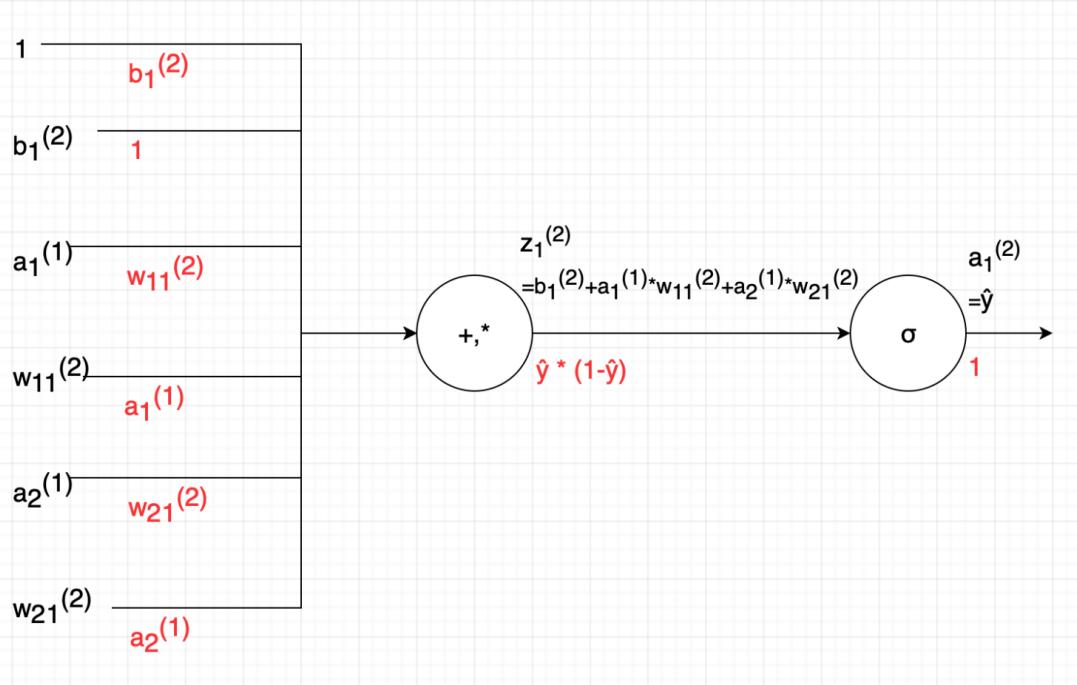
Với mỗi điểm $(x^{[i]}, y_i)$, hàm loss function

$L = -(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$ trong đó $\hat{y}_i = a_1^{(2)} = \sigma(a_1^{(1)} * w_{11}^{(2)} + a_2^{(1)} * w_{21}^{(2)} + b_1^{(2)})$ là giá trị mà model dự đoán, còn y_i là giá trị thật của dữ liệu.

$$\frac{\partial L}{\partial \hat{y}_i} = -\frac{\partial(y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))}{\partial \hat{y}_i} = -(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i})$$

Tính đạo hàm L với $W^{(2)}, b^{(2)}$

$$\text{Áp dụng chain rule ta có: } \frac{\partial L}{\partial b_1^{(2)}} = \frac{dL}{d\hat{y}_i} * \frac{\partial \hat{y}_i}{\partial b_1^{(2)}}$$



Từ đồ thị ta thấy:

$$\begin{aligned}\frac{\partial \hat{y}_i}{\partial b_1^{(2)}} &= \hat{y}_i * (1 - \hat{y}_i) \\ \frac{\partial \hat{y}_i}{\partial w_{11}^{(2)}} &= a_1^{(1)} * \hat{y}_i * (1 - \hat{y}_i) \\ \frac{\partial \hat{y}_i}{\partial w_{21}^{(2)}} &= a_2^{(1)} * \hat{y}_i * (1 - \hat{y}_i) \\ \frac{\partial \hat{y}_i}{\partial a_1^{(1)}} &= w_{11}^{(2)} * \hat{y}_i * (1 - \hat{y}_i) \\ \frac{\partial \hat{y}_i}{\partial a_2^{(1)}} &= w_{21}^{(2)} * \hat{y}_i * (1 - \hat{y}_i)\end{aligned}$$

Do đó

$$\frac{\partial L}{\partial b_1^{(2)}} = \frac{\partial L}{\partial \hat{y}_i} * \frac{\partial \hat{y}_i}{\partial b_1^{(2)}} = -\left(\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{(1-\hat{y}_i)}\right) * \hat{y}_i * (1 - \hat{y}_i) = -(y_i * (1 - \hat{y}_i) - (1 - y_i) * \hat{y}_i) = \hat{y}_i - y_i$$

Tương tự

$$\frac{\partial L}{\partial w_{11}^{(2)}} = a_1^{(1)} * (\hat{y}_i - y_i)$$

$$\frac{\partial L}{\partial w_{21}^{(2)}} = a_2^{(1)} * (\hat{y}_i - y_i)$$

$$\frac{\partial L}{\partial a_1^{(1)}} = w_{11}^{(2)} * (\hat{y}_i - y_i)$$

$$\frac{\partial L}{\partial a_2^{(1)}} = w_{21}^{(2)} * (\hat{y}_i - y_i)$$

Biểu diễn dưới dạng ma trận

*** Lưu ý: đạo hàm của L đối với ma trận W kích thước m*n cũng là một ma trận cùng kích thước m*n.

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}} & \dots & \frac{\partial L}{\partial w_{1n}} \\ \frac{\partial L}{\partial w_{21}} & \dots & \frac{\partial L}{\partial w_{2n}} \\ \dots & \dots & \dots \\ \frac{\partial L}{\partial w_{m1}} & \dots & \frac{\partial L}{\partial w_{mn}} \end{bmatrix}$$

Do đó, $\frac{\partial J}{\partial W^{(2)}} = (A^{(1)})^T * (\hat{Y} - Y)$, $\frac{\partial J}{\partial b^{(2)}} = (sum(\hat{Y} - Y))^T$, $\frac{\partial J}{\partial A^{(1)}} = (\hat{Y} - Y) * (W^{(2)})^T$, phép tính sum tính tổng các cột của ma trận.

$$\begin{aligned}W &= \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \Rightarrow sum(W) = (w_{11} + w_{21} + w_{31}, w_{12} + w_{22} + w_{32}) \\ (sum(W))^T &= \begin{bmatrix} w_{11} + w_{21} + w_{31} \\ w_{12} + w_{22} + w_{32} \end{bmatrix}\end{aligned}$$

Vậy là đã tính xong đạo hàm của L với hệ số $W^{(2)}, b^{(2)}$. Giờ sẽ đi tính đạo hàm của L với hệ số $W^{(1)}, b^{(1)}$. Khoan, tưởng chỉ cần tính đạo hàm của L với các hệ số W và bias b, tại sao cần tính đạo hàm của L với $A^{(1)}$??? Khi tính đạo hàm của hệ số và bias trong layer trước đây sẽ cần dùng đến.

Tính đạo hàm L với $W^{(1)}, b^{(1)}$

Do $a_1^{(1)} = \sigma(b_1^{(1)} + x_1 * w_{11}^{(1)} + x_2 * w_{21}^{(1)})$

Áp dụng chain rule ta có: $\frac{\partial L}{\partial b_1^{(1)}} = \frac{\partial L}{\partial a_1^{(1)}} * \frac{\partial a_1^{(1)}}{\partial b_1^{(1)}}$

Ta có:

$$\frac{\partial a_1^{(1)}}{\partial b_1^{(1)}} = \frac{\partial a_1^{(1)}}{z_1^{(1)}} * \frac{z_1^{(1)}}{\partial b_1^{(1)}} = a_1^{(1)} * (1 - a_1^{(1)})$$

Do đó

$$\frac{\partial L}{\partial b_1^{(1)}} = a_1^{(1)} * (1 - a_1^{(1)}) * w_{11}^{(2)} * (\hat{y}_i - y_i)$$

Tương tự

$$\frac{\partial L}{\partial w_{11}^{(1)}} = x_1 * a_1^{(1)} * (1 - a_1^{(1)}) * w_{11}^{(2)} * (\hat{y}_i - y_i)$$

$$\frac{\partial L}{\partial w_{12}^{(1)}} = x_1 * a_2^{(1)} * (1 - a_2^{(1)}) * w_{11}^{(2)} * (\hat{y}_i - y_i)$$

$$\frac{\partial L}{\partial w_{21}^{(1)}} = x_2 * a_1^{(1)} * (1 - a_1^{(1)}) * w_{21}^{(2)} * (\hat{y}_i - y_i)$$

$$\frac{\partial L}{\partial w_{22}^{(1)}} = x_2 * a_2^{(1)} * (1 - a_2^{(1)}) * w_{21}^{(2)} * (\hat{y}_i - y_i)$$

Viết dưới dạng ma trận

Có thể tạm viết dưới dạng chain rule là $\frac{\partial J}{\partial W^{(1)}} = \frac{\partial J}{\partial A^{(1)}} * \frac{\partial A^{(1)}}{\partial Z^{(1)}} * \frac{\partial Z^{(1)}}{\partial W^{(1)}}$ (1).

Từ trên đã tính được $\frac{\partial J}{\partial A^{(1)}} = (\hat{Y} - Y) * (W^{(2)})^T$

Đạo hàm của hàm sigmoid $\frac{d\sigma(x)}{dx} = \sigma(x) * (1 - \sigma(x))$ và $A^{(1)} = \sigma(Z^{(1)})$, nên trong (1) có thể hiểu là $\frac{\partial A^{(1)}}{\partial Z^{(1)}} = A^{(1)} * (1 - A^{(1)})$

Cuối cùng, $Z^{(1)} = X * W^{(1)} + b^{(1)}$ nên có thể tạm hiểu $\frac{\partial Z^{(1)}}{\partial W^{(1)}} = X$, nó giống như $f(x) = a * x + b \Rightarrow \frac{df}{dx} = a$ vậy.

Kết hợp tất cả lại $\frac{\partial J}{\partial W^{(1)}} = X^T * (((\hat{Y} - Y) * (W^{(2)})^T) \otimes A^{(1)} \otimes (1 - A^{(1)}))$

Thế khi nào thì dùng element-wise (\otimes), khi nào dùng nhân ma trận (*) ?

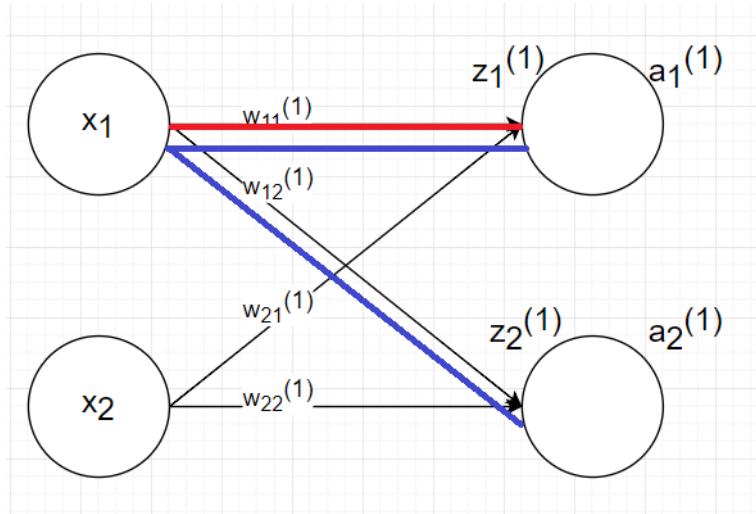
- Khi tính đạo hàm ngược lại qua bước activation thì dùng (\otimes)
- Khi có phép tính nhân ma trận thì dùng (*), nhưng đặc biệt chú ý đến kích thước ma trận và dùng transpose nếu cần thiết. Ví dụ: ma trận X kích thước $N*3$, W kích thước $3*4$, $Z = X * W$

W sẽ có kích thước $N \times 4$ thì $\frac{\partial J}{\partial W} = X^T * (\frac{\partial J}{\partial Z})$ và $\frac{\partial J}{\partial X} = (\frac{\partial J}{\partial Z}) * W^T$

Tương tự, $\frac{\partial L}{\partial b^{(1)}} = \text{sum}(((\hat{Y} - Y) * (W^{(2)})^T) \otimes A^{(1)})^T$

Vậy là đã tính xong hết đạo hàm của loss function với các hệ số W và bias b, giờ có thể áp dụng gradient descent để giải bài toán.

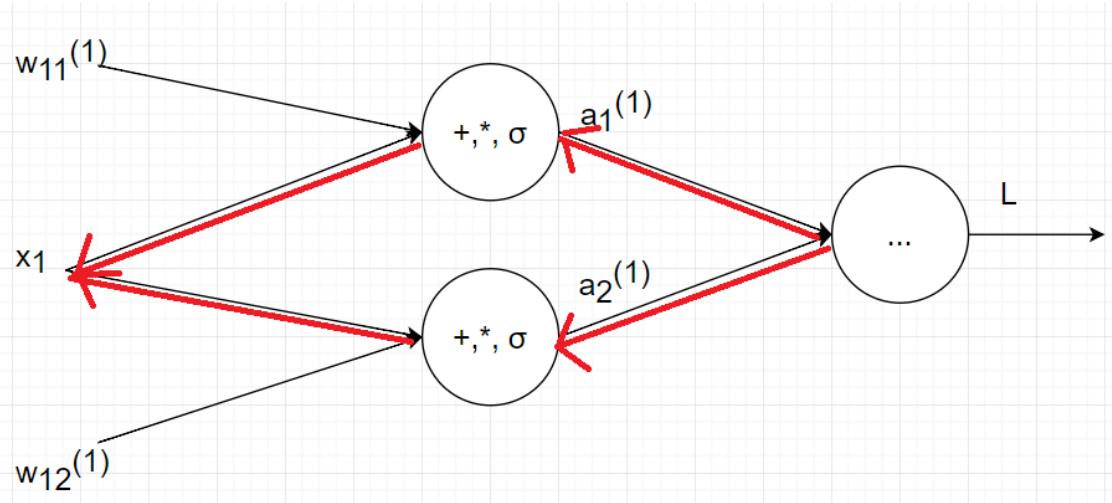
Giờ thử tính $\frac{\partial L}{\partial x_1}$, ở bài này thì không cần vì chỉ có 1 hidden layer, nhưng nếu nhiều hơn 1 hidden layer thì bạn cần tính bước này để tính đạo hàm với các hệ số trước đó.



Hình 6.1: Đường màu đỏ cho $w_{11}^{(1)}$, đường màu xanh cho x_1

Ta thấy $w_{11}^{(1)}$ chỉ tác động đến $a_1^{(1)}$, cụ thể là $a_1^{(1)} = \sigma(b_1^{(1)} + x_1 * w_{11}^{(1)} + x_2 * w_{21}^{(1)})$

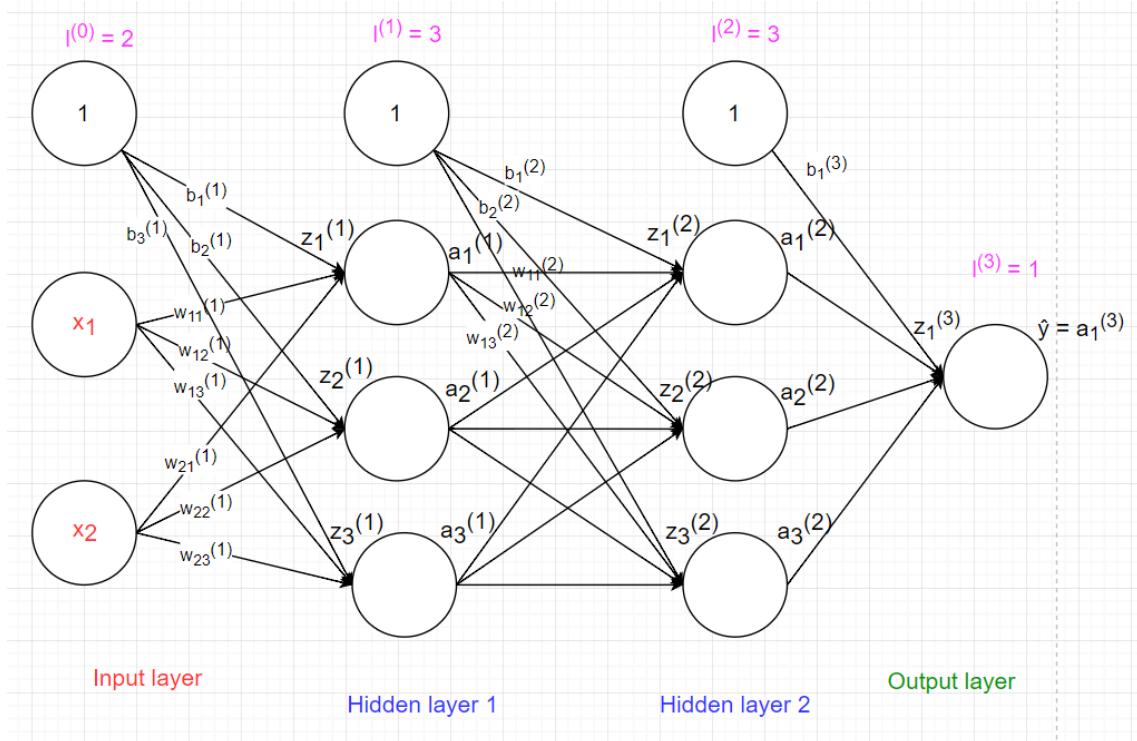
Tuy nhiên x_1 không những tác động đến $a_1^{(1)}$ mà còn tác động đến $a_2^{(1)}$, nên khi áp dụng chain rule tính đạo hàm của L với x_1 cần tính tổng đạo hàm qua cả $a_1^{(1)}$ và $a_2^{(1)}$.



Do đó:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial a_1^{(1)}} * \frac{\partial a_1^{(1)}}{\partial x_1} + \frac{\partial L}{\partial a_2^{(1)}} * \frac{\partial a_2^{(1)}}{\partial x_1} = w_{11}^{(1)} * a_1^{(1)} * (1 - a_1^{(1)}) * w_{11}^{(2)} * (y_i - \hat{y}_i) + w_{12}^{(1)} * a_2^{(1)} * (1 - a_2^{(1)}) * w_{21}^{(2)} * (y_i - \hat{y}_i)$$

6.2 Mô hình tổng quát



Hình 6.2: Mô hình neural network

Bạn có thể xem lại biểu diễn dạng ma trận với neural network ở bài trước.

1. **Bước 1:** Tính $\frac{\partial J}{\partial \hat{Y}}$, trong đó $\hat{Y} = A^{(3)}$
2. **Bước 2:** Tính $\frac{\partial J}{\partial W^{(3)}} = (A^{(2)})^T * (\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(3)}}{\partial Z^{(3)}})$, $\frac{\partial J}{\partial b^{(3)}} = (\text{sum}(\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(3)}}{\partial Z^{(3)}}))^T$ và tính $\frac{\partial J}{\partial A^{(2)}} = (\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial A^{(3)}}{\partial Z^{(3)}}) * (W^{(3)})^T$
3. **Bước 3:** Tính $\frac{\partial J}{\partial W^{(2)}} = (A^{(1)})^T * (\frac{\partial J}{\partial A^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}})$, $\frac{\partial J}{\partial b^{(2)}} = (\text{sum}(\frac{\partial J}{\partial A^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}}))^T$ và tính $\frac{\partial J}{\partial A^{(1)}} = (\frac{\partial J}{\partial A^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}}) * (W^{(2)})^T$
4. **Bước 4:** Tính $\frac{\partial J}{\partial W^{(1)}} = (A^{(0)})^T * (\frac{\partial J}{\partial A^{(1)}} \otimes \frac{\partial A^{(1)}}{\partial Z^{(1)}})$, $\frac{\partial J}{\partial b^{(1)}} = (\text{sum}(\frac{\partial J}{\partial A^{(1)}} \otimes \frac{\partial A^{(1)}}{\partial Z^{(1)}}))^T$, trong đó $A^{(0)} = X$

Nếu network có nhiều layer hơn thì cứ tiếp tục cho đến khi tính được đạo hàm của loss function J với tất cả các hệ số W và bias b.

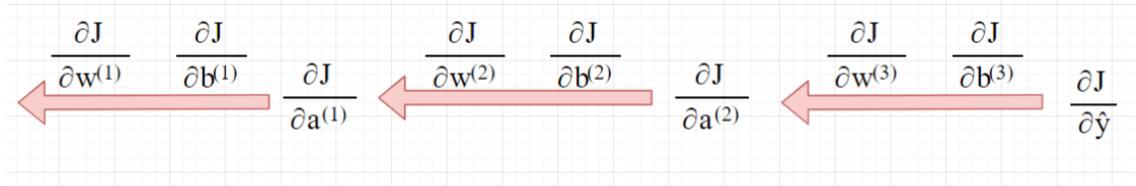
Nếu hàm activation là sigmoid thì $\frac{\partial A^{(i)}}{\partial Z^{(i)}} = A^{(i)} \otimes (1 - A^{(i)})$

Ở bài trước quá trình feedforward



Hình 6.3: Quá trình feedforward

Thì ở bài này quá trình tính đạo hàm ngược lại



Hình 6.4: Quá trình backpropagation

Đây là vì sao thuật toán được gọi là backpropagation (lan truyền ngược)

6.3 Python code

```
# Thêm thư viện
import numpy as np
import pandas as pd

# Hàm sigmoid
def sigmoid(x):
    return 1/(1+np.exp(-x))

# Đạo hàm hàm sigmoid
def sigmoid_derivative(x):
    return x*(1-x)

# Lớp neural network
class NeuralNetwork:
    def __init__(self, layers, alpha=0.1):
        # Mô hình layer ví dụ [2,2,1]
        self.layers = layers

        # Hệ số learning rate
        self.alpha = alpha

        # Tham số W, b
        self.W = []
```

```
self.b = []

# Khởi tạo các tham số ở mỗi layer
for i in range(0, len(layers)-1):
    w_ = np.random.randn(layers[i], layers[i+1])
    b_ = np.zeros((layers[i+1], 1))
    self.W.append(w_/layers[i])
    self.b.append(b_)

# Tóm tắt mô hình neural network
def __repr__(self):
    return "Neural network [{}]\n".format("-".join(str(l) for l in self.layers))

# Train mô hình với dữ liệu
def fit_partial(self, x, y):
    A = [x]

    # quá trình feedforward
    out = A[-1]
    for i in range(0, len(self.layers) - 1):
        out = sigmoid(np.dot(out, self.W[i]) + (self.b[i].T))
        A.append(out)

    # quá trình backpropagation
    y = y.reshape(-1, 1)
    dA = [-(y/A[-1] - (1-y)/(1-A[-1]))]
    dW = []
    db = []
    for i in reversed(range(0, len(self.layers)-1)):
        dw_ = np.dot((A[i]).T, dA[-1] * sigmoid_derivative(A[i+1]))
        db_ = (np.sum(dA[-1] * sigmoid_derivative(A[i+1]), 0)).reshape(-1,1)
        dA_ = np.dot(dA[-1] * sigmoid_derivative(A[i+1]), self.W[i].T)
        dW.append(dw_)
        db.append(db_)
        dA.append(dA_)

    # Đảo ngược dW, db
    dW = dW[::-1]
    db = db[::-1]

    # Gradient descent
    for i in range(0, len(self.layers)-1):
        self.W[i] = self.W[i] - self.alpha * dW[i]
        self.b[i] = self.b[i] - self.alpha * db[i]

def fit(self, X, y, epochs=20, verbose=10):
    for epoch in range(0, epochs):
```

```
self.fit_partial(X, y)
if epoch % verbose == 0:
    loss = self.calculate_loss(X, y)
    print("Epoch {}, loss {}".format(epoch, loss))

# Dự đoán
def predict(self, X):
    for i in range(0, len(self.layers) - 1):
        X = sigmoid(np.dot(X, self.W[i]) + (self.b[i]).T)
    return X

# Tính loss function
def calculate_loss(self, X, y):
    y_predict = self.predict(X)
#return np.sum((y_predict-y)**2)/2
    return -(np.sum(y*np.log(y_predict) + (1-y)*np.log(1-y_predict)))
```

6.4 Bài tập

1. Tính backpropagation với neural netwrok có 1 hidden layer đã implement từ bài trước.

Convolutional Neural Network

IV

7 Giới thiệu về xử lý ảnh 109

- 7.1 Ảnh trong máy tính
- 7.2 Phép tính convolution
- 7.3 Bài tập

8 Convolutional neural network 123

- 8.1 Thiết lập bài toán
- 8.2 Convolutional neural network
- 8.3 Mạng VGG 16
- 8.4 Visualizing Convolutional Neural Network
- 8.5 Bài tập

9 Giới thiệu keras và bài toán phân loại ảnh 139

- 9.1 Giới thiệu về keras
- 9.2 MNIST Dataset
- 9.3 Python code
- 9.4 Ứng dụng của việc phân loại ảnh
- 9.5 Bài tập

10 Ứng dụng CNN cho ô tô tự lái 149

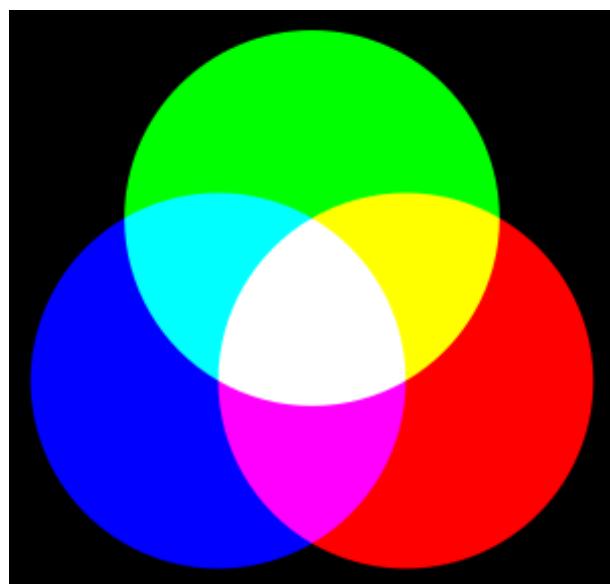
- 10.1 Giới thiệu mô phỏng ô tô tự lái
- 10.2 Bài toán ô tô tự lái
- 10.3 Python code
- 10.4 Áp dụng model cho ô tô tự lái
- 10.5 Bài tập

7. Giới thiệu về xử lý ảnh

7.1 Ảnh trong máy tính

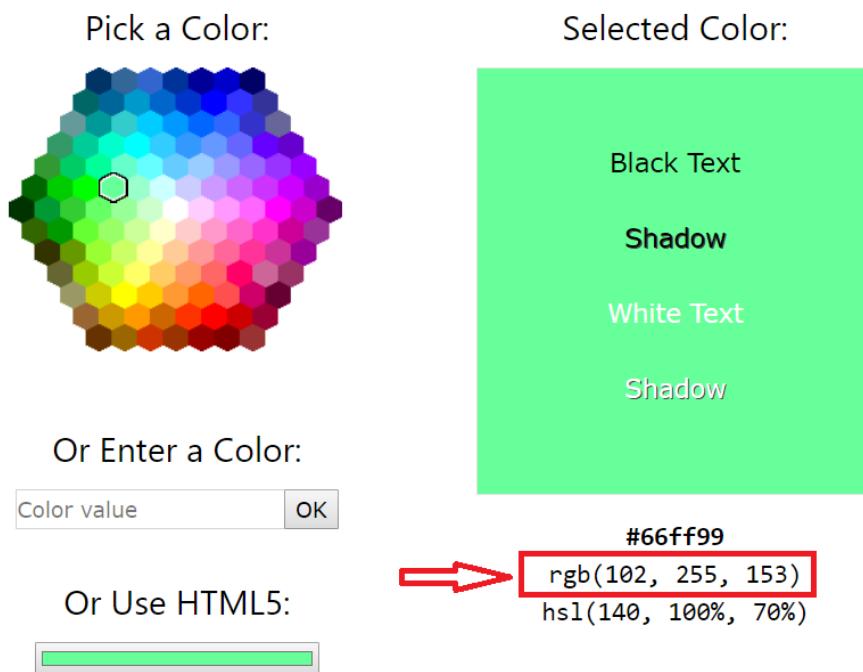
7.1.1 Hệ màu RGB

RGB viết tắt của Red (đỏ), Green (xanh lục), Blue (xanh lam), là ba màu chính của ánh sáng khi tách ra từ lăng kính. Khi trộn ba màu trên theo tỉ lệ nhất định có thể tạo thành các màu khác nhau.



Hình 7.1: Thêm đỏ vào xanh lá cây tạo ra vàng; thêm vàng vào xanh lam tạo ra trắng [19]

Ví dụ khi bạn chọn màu ở [đây](#). Khi bạn chọn một màu thì sẽ ra một bộ ba số tương ứng (r,g,b) Với mỗi bộ 3 số r, g, b nguyên trong khoảng [0, 255] sẽ cho ra một màu khác nhau. Do có 256 cách chọn r, 256 cách chọn màu g, 256 cách chọn b => tổng số màu có thể tạo ra bằng hệ màu RGB là: $256 * 256 * 256 = 16777216$ màu !!!

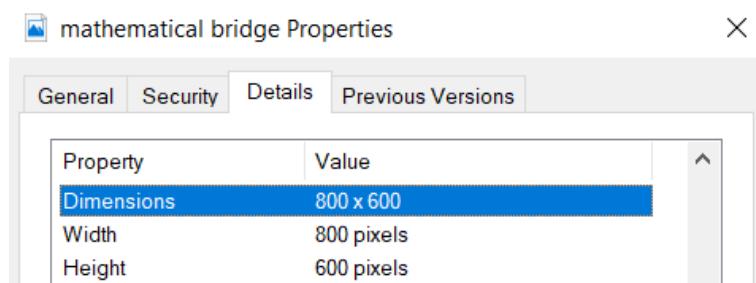


Hình 7.2: màu được chọn là $rgb(102, 255, 153)$, nghĩa là $r=102, g=255, b=153$.

7.1.2 Ảnh màu

Ví dụ về ảnh màu trong hình 7.3

Khi bạn kích chuột phải vào ảnh trong máy tính, bạn chọn properties (mục cuối cùng), rồi chọn tab details



Bạn sẽ thấy chiều dài ảnh là 800 pixels (viết tắt px), chiều rộng 600 pixels, kích thước là $800 * 600$. Trước giờ chỉ học đơn vị đo là mét hay centimet, pixel là gì nhỉ ?

Theo wiki, pixel (hay điểm ảnh) là một khối màu rất nhỏ và là đơn vị cơ bản nhất để tạo nên một bức ảnh kỹ thuật số.

Vậy bức ảnh trên kích thước $800 \text{ pixel} * 600 \text{ pixel}$, có thể biểu diễn dưới dạng một ma trận kích thước $600 * 800$ (vì định nghĩa ma trận là số hàng nhân số cột).



Hình 7.3: Mathematical bridge, Cambridge

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,800} \\ w_{2,1} & w_{2,2} & \dots & w_{2,800} \\ \dots & \dots & \dots & \dots \\ w_{600,1} & w_{600,2} & \dots & w_{600,800} \end{bmatrix}$$

Trong đó mỗi phần tử w_{ij} là một pixel.

Như vậy có thể hiểu là mỗi pixel thì biểu diễn một màu và bức ảnh trên là sự kết hợp rất nhiều pixel. Hiểu đơn giản thì in bức ảnh ra, kẻ ô vuông như chơi cờ ca rô với 800 đường thẳng ở chiều dài, 600 đường ở chiều rộng, thì mỗi ô vuông là một pixel, biểu diễn một chấm màu.

Tuy nhiên để biểu diễn 1 màu ta cần 3 thông số (r,g,b) nên gọi $w_{ij} = (r_{ij}, g_{ij}, b_{ij})$ để biểu diễn dưới dạng ma trận thì sẽ như sau:

$$\begin{bmatrix} (100, 100, 50) & (101, 112, 3) & (131, 20, 80) \\ (150, 210, 130) & (10, 120, 130) & (111, 120, 130) \\ (10, 260, 30) & (200, 20, 30) & (100, 20, 3) \end{bmatrix}$$

Hình 7.4: Ảnh màu kích thước 3*3 biểu diễn dạng ma trận, mỗi pixel biểu diễn giá trị (r,g,b)

Để tiện lưu trữ và xử lý không thể lưu trong 1 ma trận như thế kia mà sẽ tách mỗi giá trị màu trong mỗi pixel ra một ma trận riêng.

$$\begin{bmatrix} 100 & 101 & 131 \\ 150 & 10 & 111 \\ 10 & 200 & 100 \end{bmatrix}, \begin{bmatrix} 100 & 112 & 20 \\ 210 & 120 & 120 \\ 260 & 20 & 20 \end{bmatrix}, \begin{bmatrix} 50 & 3 & 80 \\ 130 & 130 & 130 \\ 30 & 30 & 3 \end{bmatrix}$$

R **G** **B**

Hình 7.5: Tách ma trận trên thành 3 ma trận cùng kích thước: mỗi ma trận lưu giá trị từng màu khác nhau red, green, blue

Tổng quát

$$\begin{bmatrix} (r_{1,1}, g_{1,1}, b_{1,1}) & (r_{1,2}, g_{1,2}, b_{1,2}) & \dots & (r_{1,800}, g_{1,800}, b_{1,800}) \\ (r_{2,1}, g_{2,1}, b_{2,1}) & (r_{2,2}, g_{2,2}, b_{2,2}) & \dots & (r_{2,800}, g_{2,800}, b_{2,800}) \\ \dots & \dots & \dots & \dots \\ (r_{600,1}, g_{600,1}, b_{600,1}) & (r_{600,2}, g_{600,2}, b_{600,2}) & \dots & (r_{600,800}, g_{600,800}, b_{600,800}) \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,800} \\ r_{2,1} & r_{2,2} & \dots & r_{2,800} \\ \dots & \dots & \dots & \dots \\ r_{600,1} & r_{600,2} & \dots & r_{600,800} \end{bmatrix}, \begin{bmatrix} g_{1,1} & g_{1,2} & \dots & g_{1,800} \\ g_{2,1} & g_{2,2} & \dots & g_{2,800} \\ \dots & \dots & \dots & \dots \\ g_{600,1} & g_{600,2} & \dots & g_{600,800} \end{bmatrix}, \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,800} \\ b_{2,1} & b_{2,2} & \dots & b_{2,800} \\ \dots & \dots & \dots & \dots \\ b_{600,1} & b_{600,2} & \dots & b_{600,800} \end{bmatrix},$$

Hình 7.6: Tách ma trận biểu diễn màu ra 3 ma trận, mỗi ma trận lưu giá trị 1 màu.

Mỗi ma trận được tách ra được gọi là 1 channel nên ảnh màu được gọi là 3 channel: channel red, channel green, channel blue.

Tóm tắt: Ảnh màu là một ma trận các pixel mà mỗi pixel biểu diễn một điểm màu. Mỗi điểm màu được biểu diễn bằng bộ 3 số (r,g,b). Để tiện cho việc xử lý ảnh thì sẽ tách ma trận pixel ra 3 channel red, green, blue.

7.1.3 Tensor là gì

Khi dữ liệu biểu diễn dạng 1 chiều, người ta gọi là vector, mặc định khi viết vector sẽ viết dưới dạng cột.

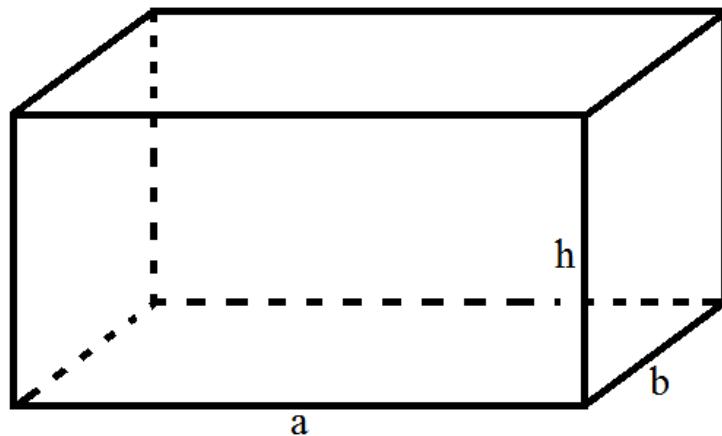
Khi dữ liệu dạng 2 chiều, người ta gọi là ma trận, kích thước là số hàng * số cột.

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix}, W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \dots & \dots & \dots & \dots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix}$$

Hình 7.7: Vector v kích thước n, ma trận W kích thước m*n

Khi dữ liệu nhiều hơn 2 chiều thì sẽ được gọi là tensor, ví dụ như dữ liệu có 3 chiều.

Để ý thì thấy là ma trận là sự kết hợp của các vector cùng kích thước. Xếp n vector kích thước m cạnh nhau thì sẽ được ma trận $m \times n$. Thì tensor 3 chiều cũng là sự kết hợp của các ma trận cùng kích thước, xếp k ma trận kích thước $m \times n$ lên nhau sẽ được tensor kích thước $m \times n \times k$.



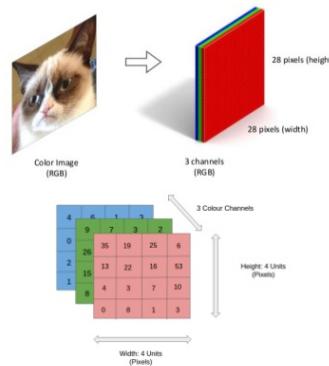
Hình 7.8: Hình hộp chữ nhật kích thước $a \times b \times h$

Tưởng tượng mặt đáy là một ma trận kích thước $a \times b$, được tạo bởi b vector kích thước a . Cả hình hộp là tensor 3 chiều kích thước $a \times b \times h$, được tạo bởi xếp h ma trận kích thước $a \times b$ lên nhau.

Do đó biểu diễn ảnh màu trên máy tính ở phần trên sẽ được biểu diễn dưới dạng tensor 3 chiều kích thước $600 \times 800 \times 3$ do có 3 ma trận (channel) màu red, green, blue kích thước 600×800 chồng lên nhau.

Ví dụ biểu diễn ảnh màu kích thước 28×28 , biểu diễn dưới dạng tensor $28 \times 28 \times 3$

color image is 3rd-order tensor



Hình 7.9: Ảnh màu biểu diễn dưới dạng tensor [1]

7.1.4 Ảnh xám



Hình 7.10: Ảnh xám của mathematical bridge

Tương tự ảnh màu, ảnh xám cũng có kích thước 800 pixel * 600 pixel, có thể biểu diễn dưới dạng một ma trận kích thước 600 * 800 (vì định nghĩa ma trận là số hàng nhân số cột).

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,800} \\ w_{2,1} & w_{2,2} & \dots & w_{2,800} \\ \dots & \dots & \dots & \dots \\ w_{600,1} & w_{600,2} & \dots & w_{600,800} \end{bmatrix}$$

Tuy nhiên mỗi pixel trong ảnh xám chỉ cần biểu diễn bằng một giá trị nguyên trong khoảng từ [0,255] thay vì (r,g,b) như trong ảnh màu. Do đó khi biểu diễn ảnh xám trong máy tính chỉ cần một ma trận là đủ.

$$\begin{bmatrix} 0 & 215 & \dots & 250 \\ 12 & 156 & \dots & 1 \\ \dots & \dots & \dots & \dots \\ 244 & 255 & \dots & 12 \end{bmatrix}$$

Hình 7.11: Biểu diễn ảnh xám

Giá trị 0 là màu đen, 255 là màu trắng và giá trị pixel càng gần 0 thì càng tối và càng gần 255 thì càng sáng.

7.1.5 Chuyển hệ màu của ảnh

Mỗi pixel trong ảnh màu được biểu diễn bằng 3 giá trị (r,g,b) còn trong ảnh xám chỉ cần 1 giá trị x để biểu diễn.

Khi chuyển từ ảnh màu sang ảnh xám ta có thể dùng công thức: $x = r * 0.299 + g * 0.587 + b * 0.114$.

Tuy nhiên khi chuyển ngược lại, bạn chỉ biết giá trị x và cần đi tìm r,g,b nên sẽ không chính xác.

7.2 Phép tính convolution

7.2.1 Convolution

Để cho dễ hình dung tôi sẽ lấy ví dụ trên ảnh xám, tức là ảnh được biểu diễn dưới dạng ma trận A kích thước $m*n$.

Ta định nghĩa kernel là một ma trận vuông kích thước $k*k$ trong đó k là số lẻ. k có thể bằng 1, 3, 5, 7, 9,... Ví dụ kernel kích thước 3*3

$$W = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Kí hiệu phép tính convolution (\otimes), kí hiệu $Y = X \otimes W$

Với mỗi phần tử x_{ij} trong ma trận X lấy ra một ma trận có kích thước bằng kích thước của kernel W có phần tử x_{ij} làm trung tâm (đây là vì sao kích thước của kernel thường lẻ) gọi là ma trận A. Sau đó tính tổng các phần tử của phép tính element-wise của ma trận A và ma trận W, rồi viết vào ma trận kết quả Y.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

\otimes

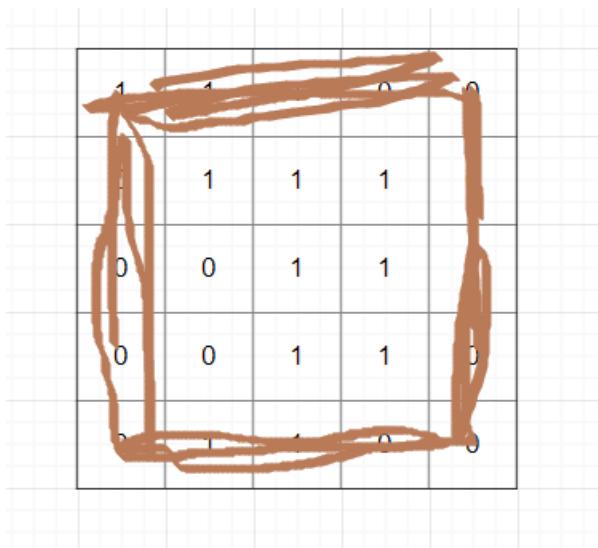
1	0	1
0	1	0
1	0	1

=

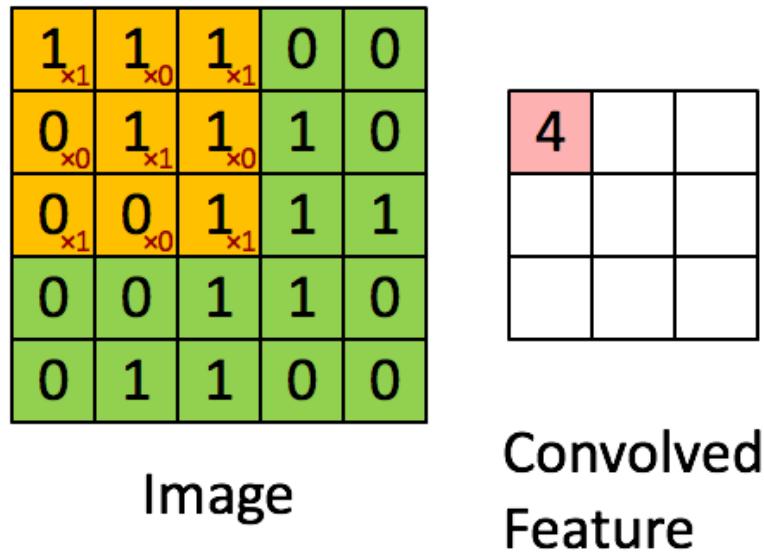
X W Y

Ví dụ khi tính tại x_{22} (ô khoanh đỏ trong hình), ma trận A cùng kích thước với W, có x_{22} làm trung tâm có màu nền da cam như trong hình. Sau đó tính $y_{11} = \text{sum}(A \otimes W) = x_{11} * w_{11} + x_{12} * w_{12} + x_{13} * w_{13} + x_{21} * w_{21} + x_{22} * w_{22} + x_{23} * w_{23} + x_{31} * w_{31} + x_{32} * w_{32} + x_{33} * w_{33} = 4$. Và làm tương tự với các phần tử còn lại trong ma trận.

Thế thì sẽ xử lý thế nào với phần tử ở viền ngoài như x_{11} ? Bình thường khi tính thì sẽ bỏ qua các phần tử ở viền ngoài, vì không tìm được ma trận A ở trong X.



Nên bạn để ý thấy ma trận Y có kích thước nhỏ hơn ma trận X. Kích thước của ma trận Y là $(m-k+1) * (n-k+1)$.



Hình 7.12: Các bước thực hiện phép tính convolution cho ma trận X với kernel K ở trên

7.2.2 Padding

Như ở trên thì mỗi lần thực hiện phép tính convolution xong thì kích thước ma trận Y đều nhỏ hơn X. Tuy nhiên giờ ta muốn ma trận Y thu được có kích thước bằng ma trận X \Rightarrow Tìm cách giải quyết cho các phần tử ở viền \Rightarrow Thêm giá trị 0 ở viền ngoài ma trận X.

0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	1	1	0	0	0

Hình 7.13: Ma trận X khi thêm viền 0 bên ngoài

Rõ ràng là giờ đã giải quyết được vấn đề tìm A cho phần tử x_{11} , và ma trận Y thu được sẽ bằng

kích thước ma trận X ban đầu.

Phép tính này gọi là convolution với padding=1. Padding=k nghĩa là thêm k vector 0 vào mỗi phía (trên, dưới, trái, phải) của ma trận.

7.2.3 Stride

Như ở trên ta thực hiện tuần tự các phần tử trong ma trận X, thu được ma trận Y cùng kích thước ma trận X, ta gọi là stride=1.

0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	0	0	0	0	0

Hình 7.14: stride=1, padding=1

Tuy nhiên nếu stride=k ($k > 1$) thì ta chỉ thực hiện phép tính convolution trên các phần tử $x_{1+i*k, 1+j*k}$. Ví dụ $k = 2$.

0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	0	0	0	0	0

Hình 7.15: padding=1, stride=2

Hiểu đơn giản là bắt đầu từ vị trí x_{11} sau đó nhảy k bước theo chiều dọc và ngang cho đến hết ma trận X.

Kích thước của ma trận Y là $3*3$ đã giảm đi đáng kể so với ma trận X.

Công thức tổng quát cho phép tính convolution của ma trận X kích thước $m*n$ với kernel kích thước $k*k$, stride = s, padding = p ra ma trận Y kích thước $(\frac{m-k+2p}{s}+1) * (\frac{n-k+2p}{s}+1)$.

Stride thường dùng để giảm kích thước của ma trận sau phép tính convolution.

Mọi người có thể xem thêm trực quan hơn ở [đây](#).

7.2.4 Ý nghĩa của phép tính convolution

Mục đích của phép tính convolution trên ảnh là làm mờ, làm nét ảnh; xác định các đường;... Mỗi kernel khác nhau thì sẽ phép tính convolution sẽ có ý nghĩa khác nhau. Ví dụ:

Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Hình 7.16: Một số kernel phổ biến [12]

7.3 Bài tập

1. Cài thư viện opencv trên python và thực hiện công việc sau:
 - (a) Load và hiện thị ảnh màu/xám
 - (b) Chuyển từ ảnh màu sang ảnh xám
 - (c) Resize ảnh
 - (d) Rotate ảnh
 - (e) Threshold trên ảnh

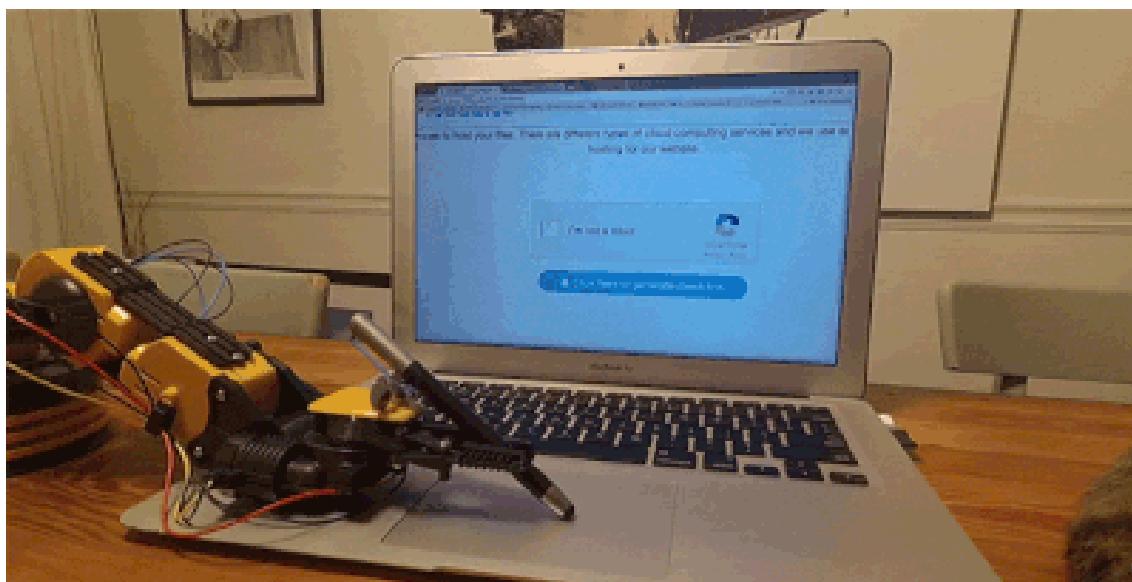
- (f) Tìm các đường thẳng, đường tròn trên ảnh.
2. Thử các kernel khác nhau trên ảnh (identity, edge detection, sharpen, blue).
 3. Tự implement lại phép tính convolution với input là ảnh, kernel, padding, stride.
 4. Tự implement lại thuật toán sobel tìm edge của ảnh.
 5. Tự implement lại HoughLine, HoughCircle, Canny bằng code python thuần.

8. Convolutional neural network

Bài này sẽ giới thiệu về convolutional neural network, sẽ được dùng khi input của neural network là ảnh. Mọi người nên đọc trước bài neural network và xử lý ảnh trước khi bắt đầu bài này.

8.1 Thiết lập bài toán

Gần đây việc kiểm tra mã captcha để xác minh không phải robot của google bị chính robot vượt qua



Hình 8.1: Robot vượt qua kiểm tra captcha

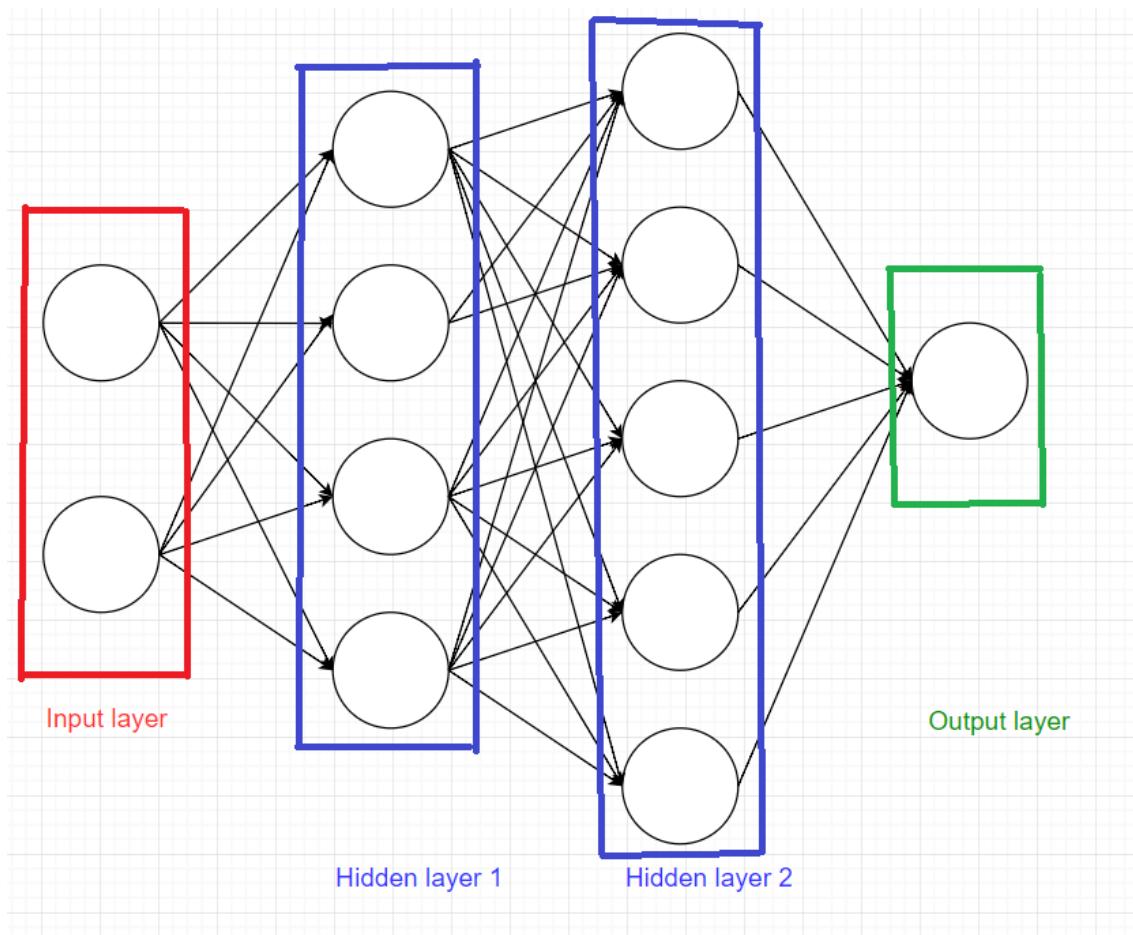
Thế nên google quyết định cho ra thuật toán mới, dùng camera chụp ảnh người dùng và dùng deep learning để xác minh xem ảnh có chứa mặt người không thay cho hệ thống captcha cũ.

Bài toán: Input một ảnh màu kích thước $64*64$, output ảnh có chứa mặt người hay không.

8.2 Convolutional neural network

8.2.1 Convolutional layer

Mô hình neural network từ những bài trước

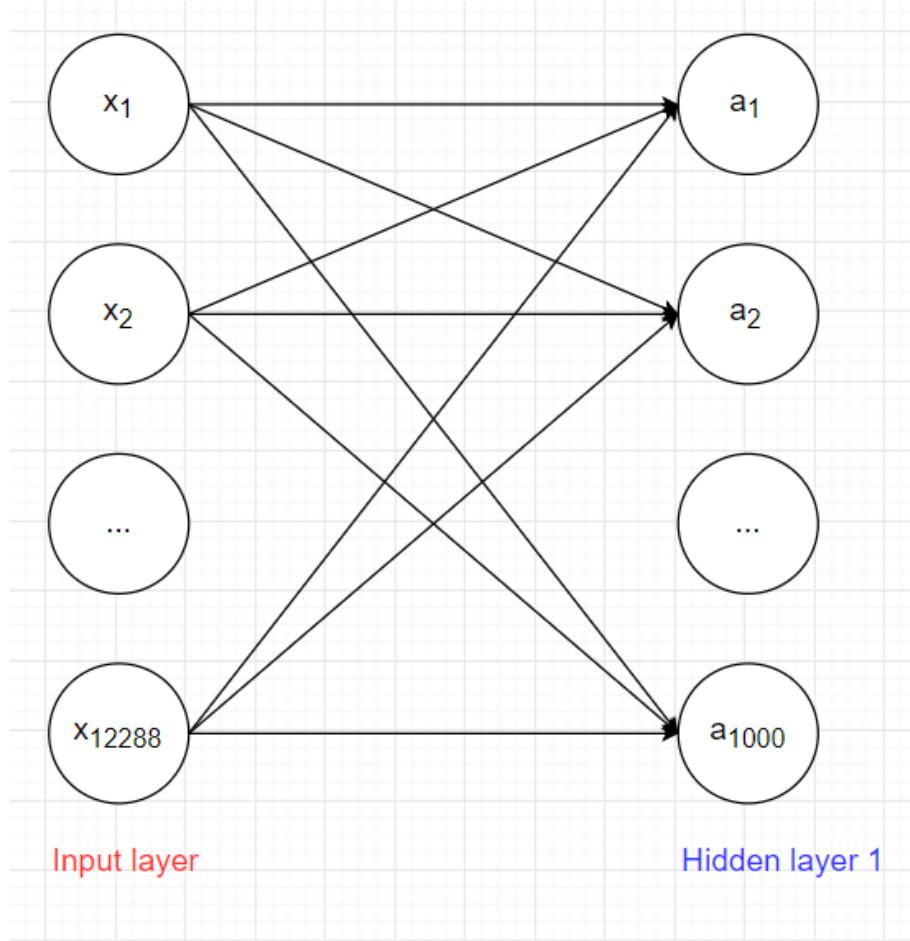


Hình 8.2: Mô hình neural network.

Mỗi hidden layer được gọi là fully connected layer, tên gọi theo đúng ý nghĩa, mỗi node trong hidden layer được kết nối với tất cả các node trong layer trước. Cả mô hình được gọi là fully connected neural network (FCN).

Vấn đề của fully connected neural network với xử lý ảnh

Như bài trước về xử lý ảnh, thì ảnh màu $64*64$ được biểu diễn dưới dạng 1 tensor $64*64*3$. Nên để biểu thị hết nội dung của bức ảnh thì cần truyền vào input layer tất cả các pixel ($64*64*3 = 12288$). Nghĩa là input layer giờ có 12288 nodes.



Hình 8.3: Input layer và hidden layer 1

Giả sử số lượng node trong hidden layer 1 là 1000. Số lượng weight W giữa input layer và hidden layer 1 là $12288 * 1000 = 12288000$, số lượng bias là 1000 \Rightarrow tổng số parameter là: 12289000. Đây mới chỉ là số parameter giữa input layer và hidden layer 1, trong model còn nhiều layer nữa, và nếu kích thước ảnh tăng, ví dụ $512 * 512$ thì số lượng parameter tăng cực kì nhanh \Rightarrow Cần giải pháp tối ưu !!!

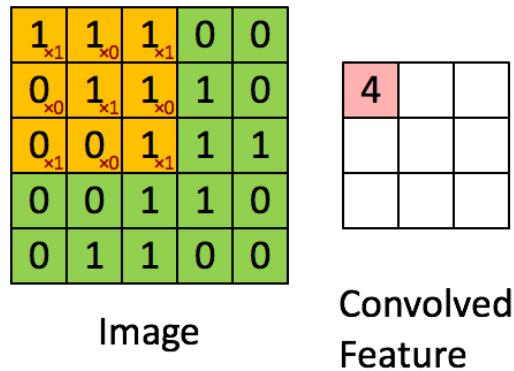
Nhận xét:

- Trong ảnh các pixel ở cạnh nhau thường có liên kết với nhau hơn là những pixel ở xa. Ví dụ như phép tính convolution trên ảnh ở bài trước. Để tìm các đường cong trong ảnh, ta áp dụng sobel kernel trên mỗi vùng kích thước $3 * 3$. Hay làm nét ảnh ta áp dụng sharpen kernel cũng trên vùng có kích thước $3 * 3$.
- Với phép tính convolution trong ảnh, chỉ 1 kernel được dùng trên toàn bộ bức ảnh. Hay nói cách khác là các pixel ảnh chia sẻ hệ số với nhau.

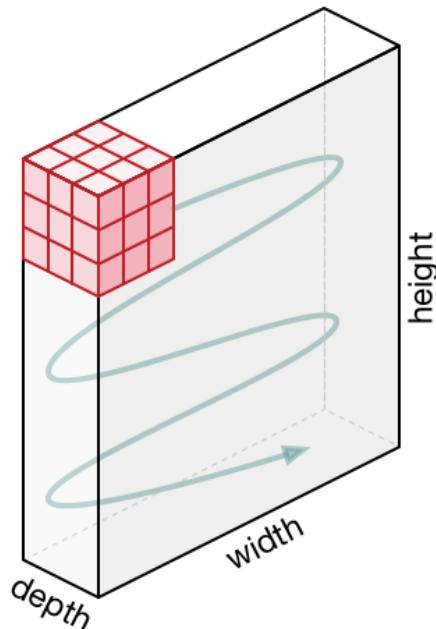
\Rightarrow Áp dụng phép tính convolution vào layer trong neural network ta có thể giải quyết được vấn đề lượng lớn parameter mà vẫn lấy ra được các đặc trưng của ảnh.

Convolutional layer đầu tiên

Bài trước phép tính convolution thực hiện trên ảnh xám với biểu diễn ảnh dạng ma trận

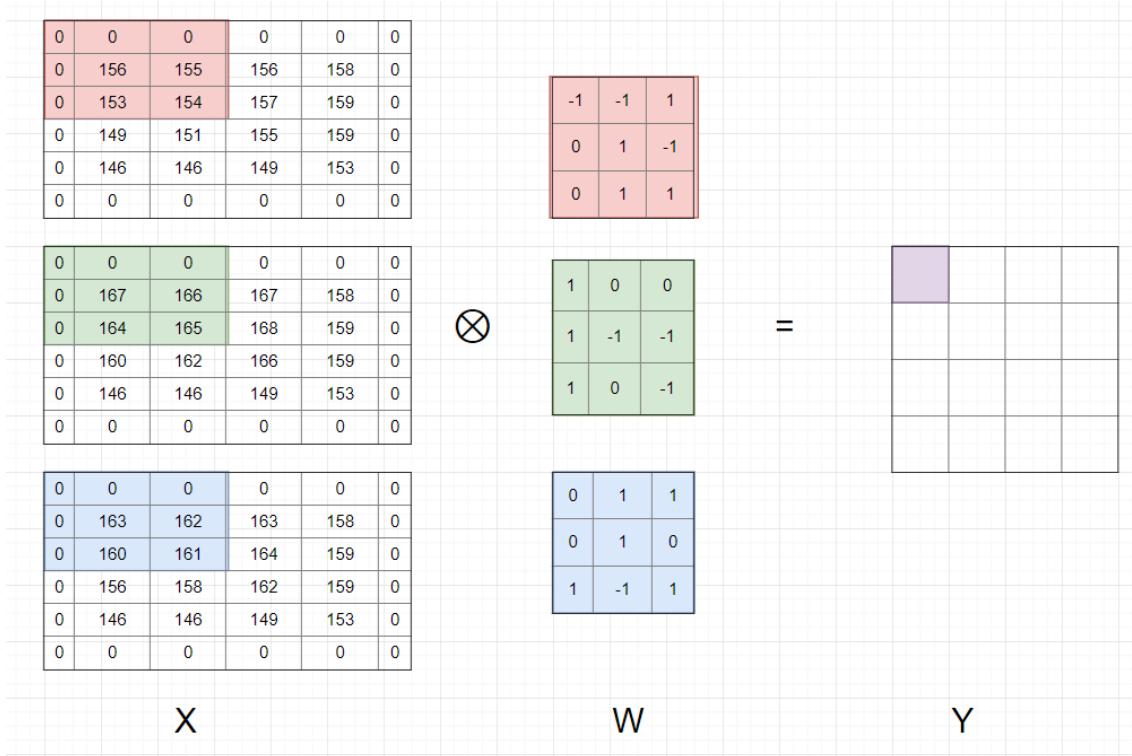


Tuy nhiên ảnh màu có tới 3 channels red, green, blue nên khi biểu diễn ảnh dưới dạng tensor 3 chiều. Nên ta cũng sẽ định nghĩa kernel là 1 tensor 3 chiều kích thước $k*k*3$.



Hình 8.4: Phép tính convolution trên ảnh màu với $k=3$.

Ta định nghĩa kernel có cùng độ sâu (depth) với biểu diễn ảnh, rồi sau đó thực hiện di chuyển khôi kernel tương tự như khi thực hiện trên ảnh xám.



Hình 8.5: Tensor X, W 3 chiều được viết dưới dạng 3 matrix.

Khi biểu diễn ma trận ta cần 2 chỉ số hàng và cột: i và j, thì khi biểu diễn ở dạng tensor 3 chiều cần thêm chỉ số độ sâu k. Nên chỉ số mỗi phần tử trong tensor là x_{ijk} .

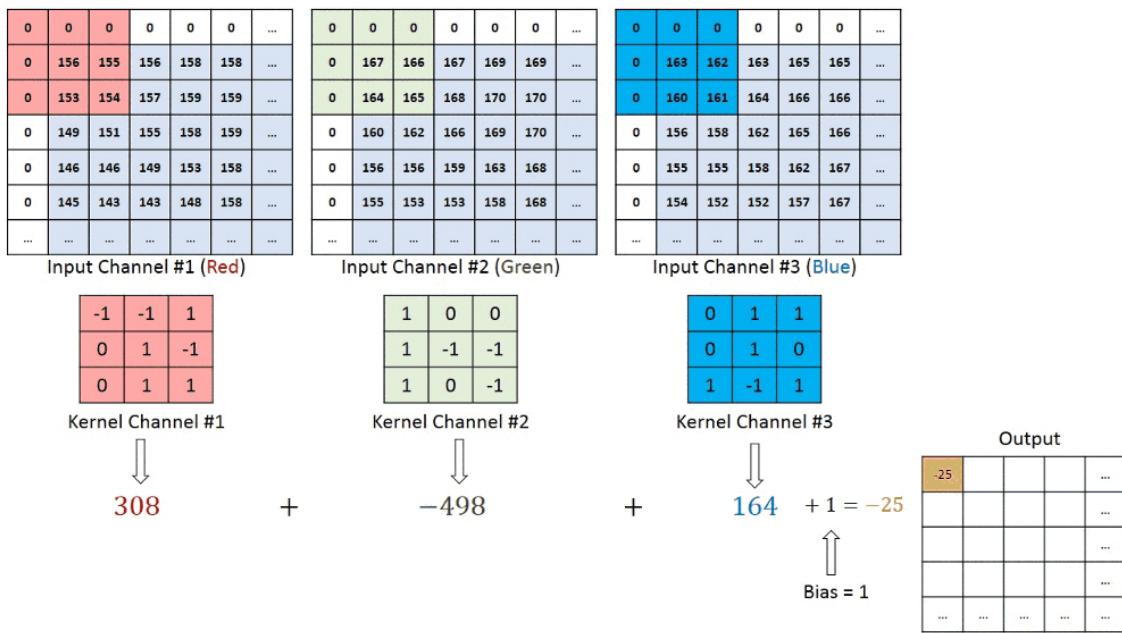
$$y_{11} = b + (x_{111} * w_{111} + x_{121} * w_{121} + x_{131} * w_{131} + x_{211} * w_{211} + x_{221} * w_{221} + x_{231} * w_{231} + x_{311} * w_{311} + x_{321} * w_{321} + x_{331} * w_{331}) + (x_{112} * w_{112} + x_{122} * w_{122} + x_{132} * w_{132} + x_{212} * w_{212} + x_{222} * w_{222} + x_{232} * w_{232} + x_{312} * w_{312} + x_{322} * w_{322} + x_{332} * w_{332}) + (x_{113} * w_{113} + x_{123} * w_{123} + x_{133} * w_{133} + x_{213} * w_{213} + x_{223} * w_{223} + x_{233} * w_{233} + x_{313} * w_{313} + x_{323} * w_{323} + x_{333} * w_{333}) = -25$$

Nhận xét:

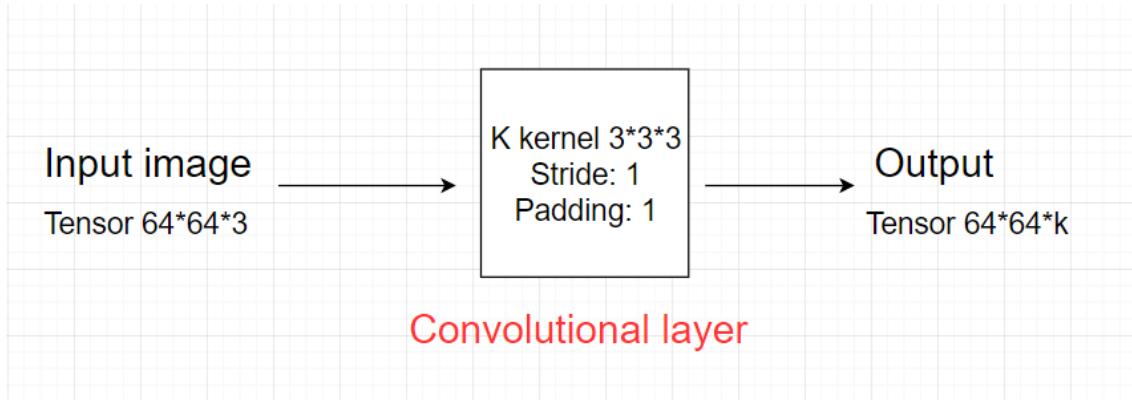
- Output Y của phép tính convolution trên ảnh màu là 1 matrix.
- Có 1 hệ số bias được cộng vào sau bước tính tổng các phần tử của phép tính element-wise

Các quy tắc đối với padding và stride toàn hoàn tương tự như ở bài trước.

Với mỗi kernel khác nhau ta sẽ học được những đặc trưng khác nhau của ảnh, nên trong mỗi convolutional layer ta sẽ dùng nhiều kernel để học được nhiều thuộc tính của ảnh. Vì mỗi kernel cho ra output là 1 matrix nên k kernel sẽ cho ra k output matrix. Ta kết hợp k output matrix này lại thành 1 tensor 3 chiều có chiều sâu k.



Hình 8.6: Thực hiện phép tính convolution trên ảnh màu.



Hình 8.7: Convolutional layer đầu tiên

Output của convolutional layer đầu tiên sẽ thành input của convolutional layer tiếp theo.

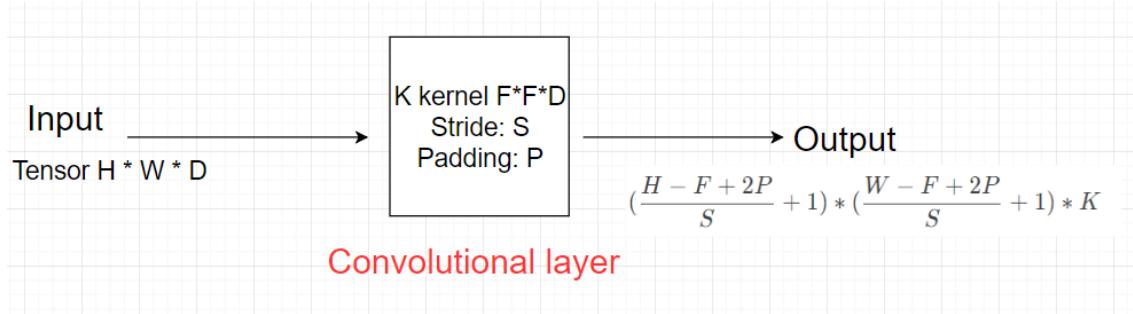
Convolutional layer tổng quát

Giả sử input của 1 convolutional layer tổng quát là tensor kích thước $H * W * D$.

Kernel có kích thước $F * F * D$ (kernel luôn có depth bằng depth của input và F là số lẻ), stride: S , padding: P .

Convolutional layer áp dụng K kernel.

\Rightarrow Output của layer là tensor 3 chiều có kích thước: $(\frac{H-F+2P}{S}+1) * (\frac{W-F+2P}{S}+1) * K$



Lưu ý:

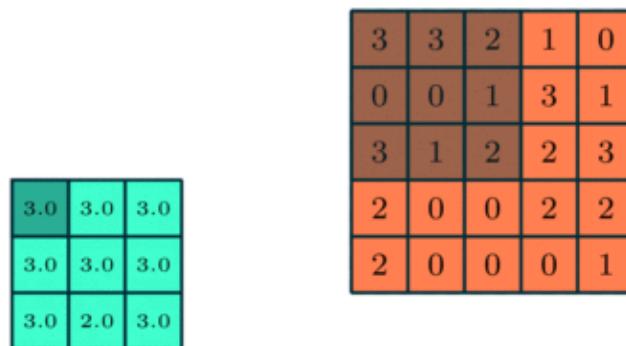
- Output của convolutional layer sẽ qua hàm non-linear activation function trước khi trở thành input của convolutional layer tiếp theo.
- Tổng số parameter của layer: Mỗi kernel có kích thước F^*F^*D và có 1 hệ số bias, nên tổng parameter của 1 kernel là $F^*F^*D + 1$. Mà convolutional layer áp dụng K kernel \Rightarrow Tổng số parameter trong layer này là $K * (F^*F^*D + 1)$.

8.2.2 Pooling layer

Pooling layer thường được dùng giữa các convolutional layer, để giảm kích thước dữ liệu nhưng vẫn giữ được các thuộc tính quan trọng. Việc giảm kích thước dữ liệu giúp giảm các phép tính toán trong model.

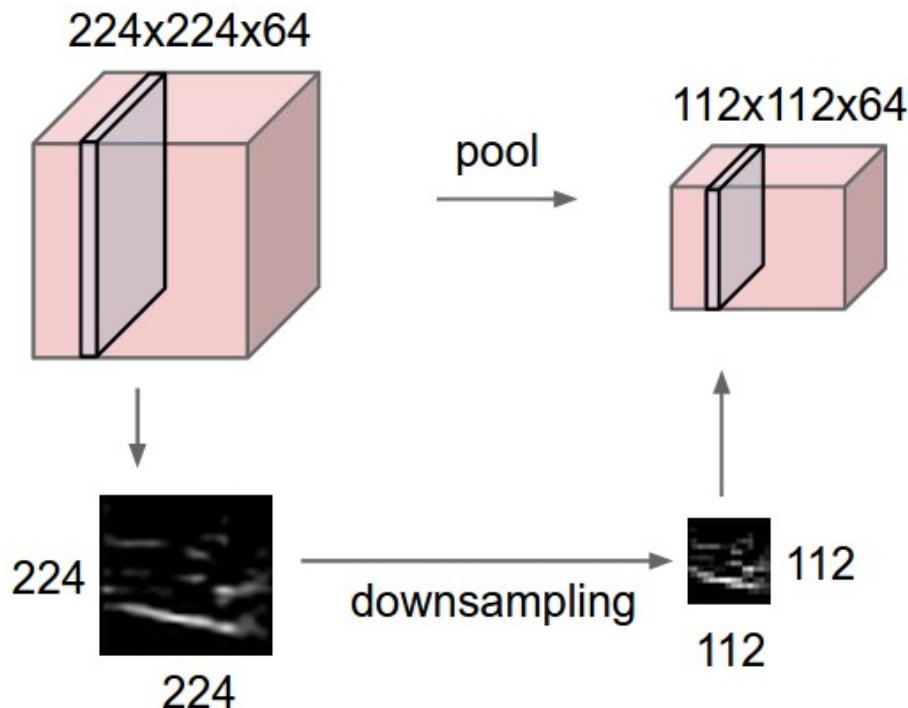
Bên cạnh đó, với phép pooling kích thước ảnh giảm, do đó lớp convolution học được các vùng có kích thước lớn hơn. Ví dụ như ảnh kích thước $224*224$ qua pooling về $112*112$ thì vùng $3*3$ ở ảnh $112*112$ tương ứng với vùng $6*6$ ở ảnh ban đầu. Vì vậy qua các pooling thì kích thước ảnh nhỏ đi và convolutional layer sẽ học được các thuộc tính lớn hơn.

Gọi pooling size kích thước $K*K$. Input của pooling layer có kích thước $H*W*D$, ta tách ra làm D ma trận kích thước $H*W$. Với mỗi ma trận, trên vùng kích thước $K*K$ trên ma trận ta tìm maximum hoặc average của dữ liệu rồi viết vào ma trận kết quả. Quy tắc về stride và padding áp dụng như phép tính convolution trên ảnh.



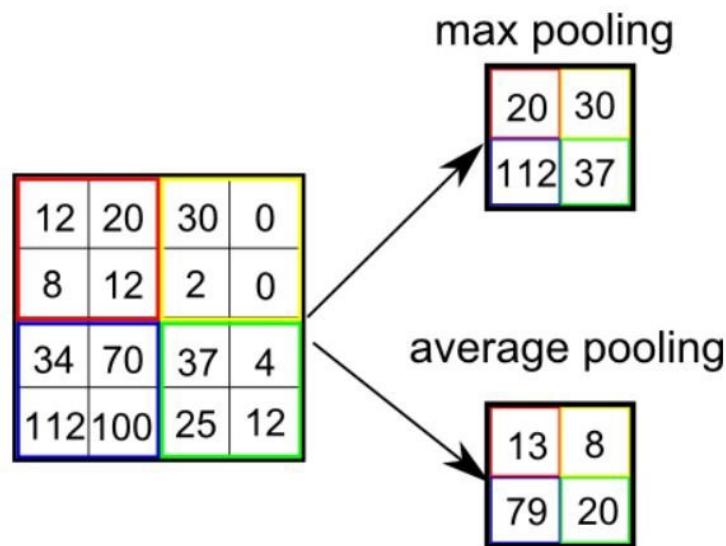
Hình 8.8: max pooling layer với size=(3,3), stride=1, padding=0

Nhưng hầu hết khi dùng pooling layer thì sẽ dùng size=(2,2), stride=2, padding=0. Khi đó output width và height của dữ liệu giảm đi một nửa, depth thì được giữ nguyên .



Hình 8.9: Sau pooling layer (2*2) [4]

Có 2 loại pooling layer phổ biến là: max pooling và average pooling.



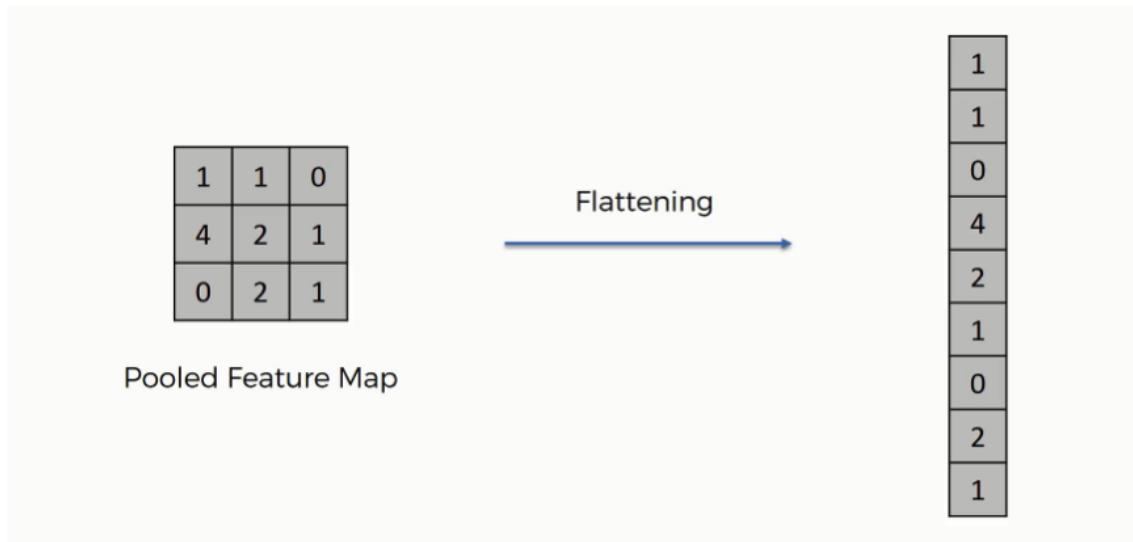
Hình 8.10: Ví dụ về pooling layer

Trong một số model người ta dùng convolutional layer với stride > 1 để giảm kích thước dữ liệu thay cho pooling layer.

8.2.3 Fully connected layer

Sau khi ảnh được truyền qua nhiều convolutional layer và pooling layer thì model đã học được tương đối các đặc điểm của ảnh (ví dụ mắt, mũi, khung mặt,...) thì tensor của output của layer cuối

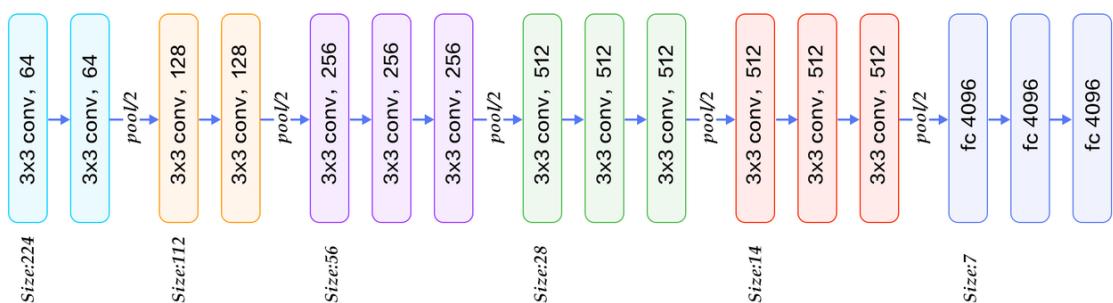
cùng, kích thước $H \times W \times D$, sẽ được chuyển về 1 vector kích thước $(H \times W \times D, 1)$



Sau đó ta dùng các fully connected layer để kết hợp các đặc điểm của ảnh để ra được output của model.

8.3 Mạng VGG 16

VGG16 là mạng convolutional neural network được đề xuất bởi K. Simonyan and A. Zisserman, University of Oxford. Model sau khi train bởi mạng VGG16 đạt độ chính xác 92.7% top-5 test trong dữ liệu ImageNet gồm 14 triệu hình ảnh thuộc 1000 lớp khác nhau. Giờ áp dụng kiến thức ở trên để phân tích mạng VGG 16.



Hình 8.11: Kiến trúc VGG16 conv: convolutional layer, pool: pooling layer, fc: fully connected layer

Phân tích:

- Convolutional layer: kích thước 3×3 , padding=1, stride=1. Tại sao không ghi stride, padding mà vẫn biết? Vì mặc định sẽ là stride=1 và padding để cho output cùng width và height với input.
- Pool/2 : max pooling layer với size 2×2
- 3×3 conv, 64: thì 64 là số kernel áp dụng trong layer đấy, hay depth của output của layer đấy.
- Càng các convolutional layer sau thì kích thước width, height càng giảm nhưng depth càng tăng.

- Sau khá nhiều convolutional layer và pooling layer thì dữ liệu được flatten và cho vào fully connected layer.

Bài sau tôi sẽ giới thiệu về keras và hướng dẫn dùng keras để áp dụng convolutional neural vào các ứng dụng như nhận diện số viết, dự đoán góc di chuyển trong ô tô tự lái.

8.4 Visualizing Convolutional Neural Network

Bên cạnh những thứ rất thú vị mà mô hình học sâu mang lại, thứ mà thực sự bên trong mô hình học sâu vẫn được coi là một chiếc hộp đen. Rõ ràng chúng ta luôn muốn giải mã chiếc hộp đen đó, rằng tại sao mô hình lại phân biệt được các chữ số, hay tại sao tìm ra vị trí của chú chó bull và xa hơn là tại sao mô hình lại có thể chuẩn đoán một tế bào là tế bào ung thư hay không. Càng ứng dụng các mô hình học sâu vào đời sống, việc giải mã chiếc hộp đen càng quan trọng. Nó dẫn đến 2 vấn đề mà ta cần biết:

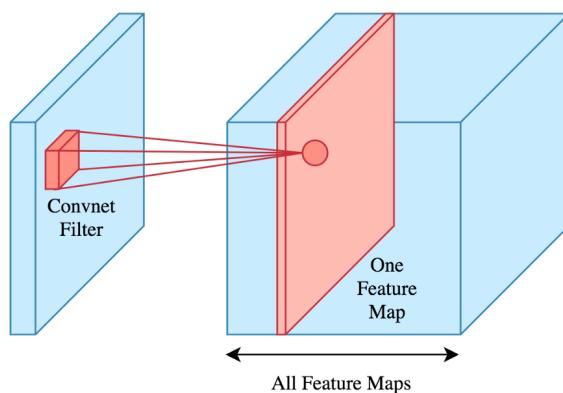
- Để hiểu mô hình học sâu làm gì, từ đó có những giải pháp cải tiến mô hình.
- Nhiều bài toán liên quan đến pháp lý cần lời giải thích hợp lý cho phương hướng giải quyết hơn là chấp nhận chiếc hộp đen

Thật may mắn khi mạng CNN có thể trực quan hóa mô hình làm những gì. Có 3 thành phần quan trọng trong mô hình CNN mà ta có thể visualize để hiểu mô hình CNN thực hiện việc gì:

- Features Map
- Convnet Filters
- Class Output

8.4.1 Visualizing Feature Maps

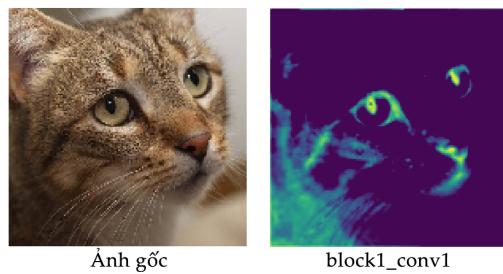
Như đã viết ở phần trên, với một ảnh có số kênh là C , sau khi được đưa qua convolutional layer với K convnet filters, ta được một feature map có K kênh, với mỗi kênh là một feature map được trích xuất từ một convet filter tương ứng.



Hình 8.12: Tensor gồm các Feature Maps

Như đề cập ở trên, đầu ra của convolutional layer là một tensor 3D với chiều rộng, chiều cao và chiều sâu là số kênh. Như vậy mỗi kênh của tensor là một kết quả độc lập từ phép tích chập tương ứng với mỗi filter và có 2 chiều tương ứng. Ta có thể tách từng kênh ra và xuất ra được một hình ảnh đại diện cho kênh đó. Đây chính là cách để visualize feature map.

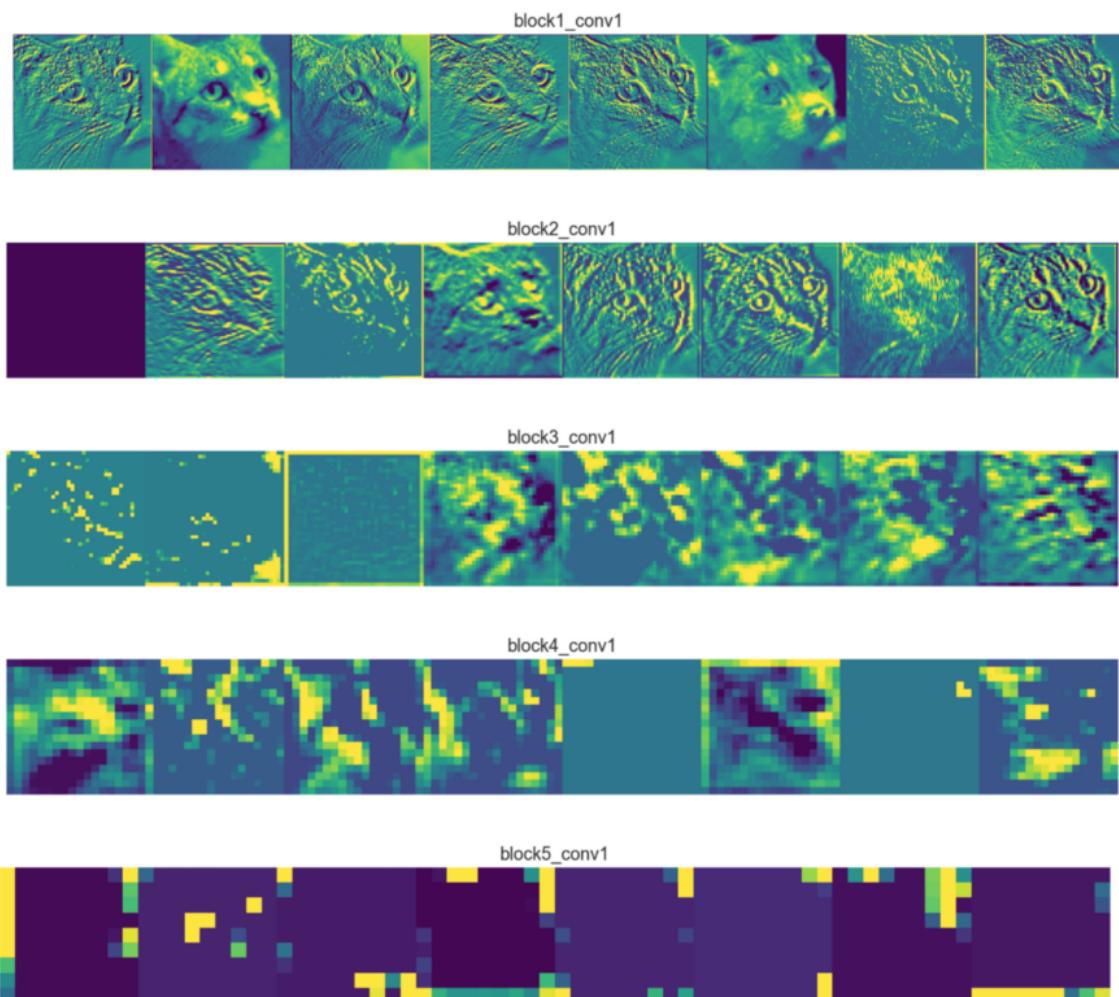
Trong kiến trúc của mạng VGG-16, ta có 5 khối (block) chính, mỗi block có các convolutional layer. Để tiện gọi tên các lớp, quy ước blockX-convY với X và Y là thứ tự tương ứng của block X và conv layer Y. Ví dụ với conv thứ 2 trong khối thứ 4 có tên là block4-conv2.



Hình 8.13: Visualize block1-conv1 của mạng VGG16

Phần có màu sáng hơn được gọi là vùng kích hoạt, nghĩa là vùng đó đã được filter lọc ra thứ mà nó cần tìm. Như hình trên, có vẻ filter này đang muốn trích xuất ra phần mắt và mũi của chú mèo.

Mỗi tensor của conv layer gồm nhiều featurer maps khác nhau, hình bên dưới ta thực hiện visualize 8 feature maps của 5 block khác nhau trong mạng VGG16.



Hình 8.14: Visualize một số feature map từ các block

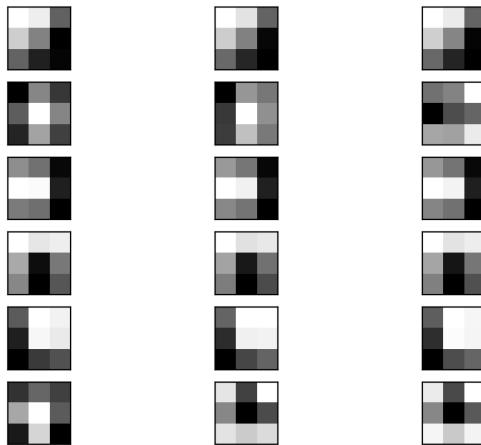
Khá thú vị phải không. Để ý hơn ta có thể đưa ra vài nhận xét như sau:

- Các lớp đầu tiên (block1-conv1) chủ yếu giữ lại hầu hết thông tin của ảnh. Trong mạng CNN, các lớp đầu tiên thường là trích xuất viền cạnh của ảnh.

- Càng đi sâu, các lớp các khó có thể nhận ra là hình gì. Như hình ở block2-conv1, ta vẫn có thể nhận ra đó là con mèo. Tuy nhiên đến block thứ 3 thì rất khó đoán được đó là hình gì và block thứ 4 thì không thể đoán nổi. Lý do là càng về sâu, các feature map biểu diễn cho những hình trừu tượng hơn. Những thông tin càng cụ thể hơn thì ta càng khó hình dung nó là gì. Chẳng hạn như các lớp về sau chỉ giữ lại thông tin của tai, mắt hay mũi của mèo mà thôi.
- Các thông tin càng về sau càng thưa thớt hơn. Điều này hợp lý vì những lớp đầu giữ lại các viền, cạnh của ảnh, thứ mà bức ảnh nào cũng có. Tuy nhiên càng đi sâu, mô hình tìm những thứ phức tạp hơn ví dụ như cái đuôi của con mèo - thứ mà không phải hình nào cũng có. Vậy nên đó là lý do tại sao ta thấy một số hình ở trên bị trống.

8.4.2 Visualizing Convolutional Filters

Cách thứ hai để biểu diễn thứ mà mạng CNN làm là visualize conv filters. Tuy nhiên ta sẽ không trực tiếp visualize conv filters bởi vì như đã biết, các filters thường có kích thước rất nhỏ (3×3 hoặc 5×5). Việc visualize nó không đem lại nhiều ý nghĩa như bạn thấy ở hình dưới đây.

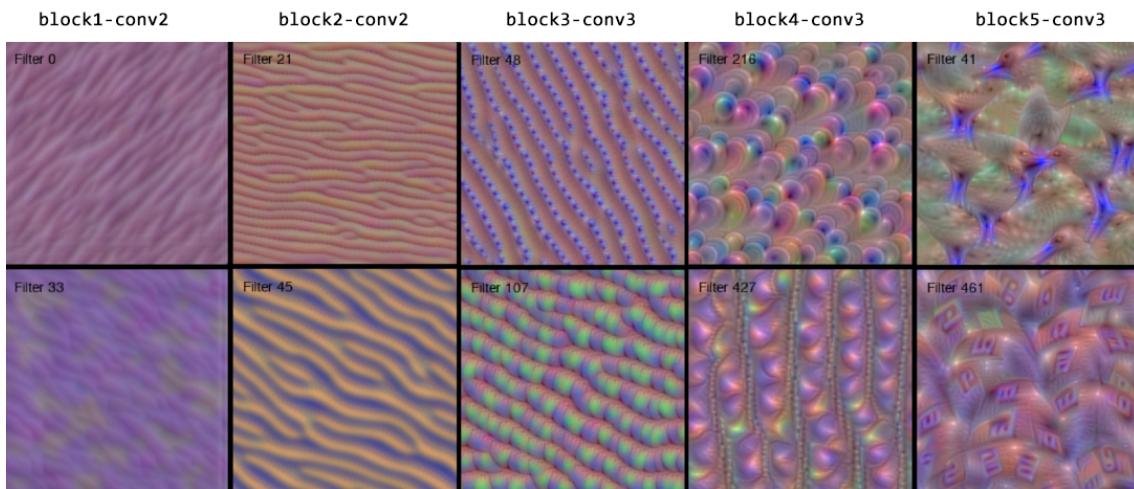


Hình 8.15: Visualize 3×3 filters

Thay vào đó, ta sẽ thử biểu diễn các filters thông qua một hình đầu vào mà khi qua hàm tích chập với filters tương ứng đạt được giá trị kích hoạt trung bình cực đại. Có một số kỹ thuật nâng cao, được đề xuất bởi [Erhan et al. 2009](#), nhưng tựa chung gồm các bước cơ bản như sau:

- Một hình ảnh với giá trị các pixel ngẫu nhiên và một pre-trained model CNN.
- Ta sẽ tìm các giá trị của pixel sao cho hàm loss là để cực đại trung bình giá trị kích hoạt của filter bằng cách thực hiện gradient ascent.
- Lặp lại đến khi đạt giá trị cực đại

Lưu ý rằng, trong phần trước ta visualize đầu ra của ảnh khi đã đi qua filter, phần này là visualize xem filter đó dùng để làm gì bằng cách tìm một hình khiến cho filter đó kích hoạt tối đa các giá trị pixel.



Hình 8.16: Visualize 2 filters với mỗi layer tương ứng

Càng về các layer cuối, hình ảnh càng rõ ràng hơn:

- Các filter đầu tiên chủ yếu dùng để phát hiện những cạnh, viền, màu sắc và những hình dạng đơn giản.
- Các filter về sau mã hoá những hình ảnh phức tạp hơn. Như ta thấy filter 41 ở layer block5-conv3 có vẻ dùng để nhận ra các con chim. Khi mà rất nhiều hình con chim với dáng khác nhau ở đó, nó khiến cho giá trị trung bình hàm kích hoạt gần đến cực đại.

Có thể thấy mô hình CNN cũng giống con người. Khi còn là trẻ sơ sinh, ta tập trung vào những hình đơn giản trước rồi qua quá trình luyện tập những hình ảnh phức tạp dễ dàng được nhận ra bởi não bộ con người.

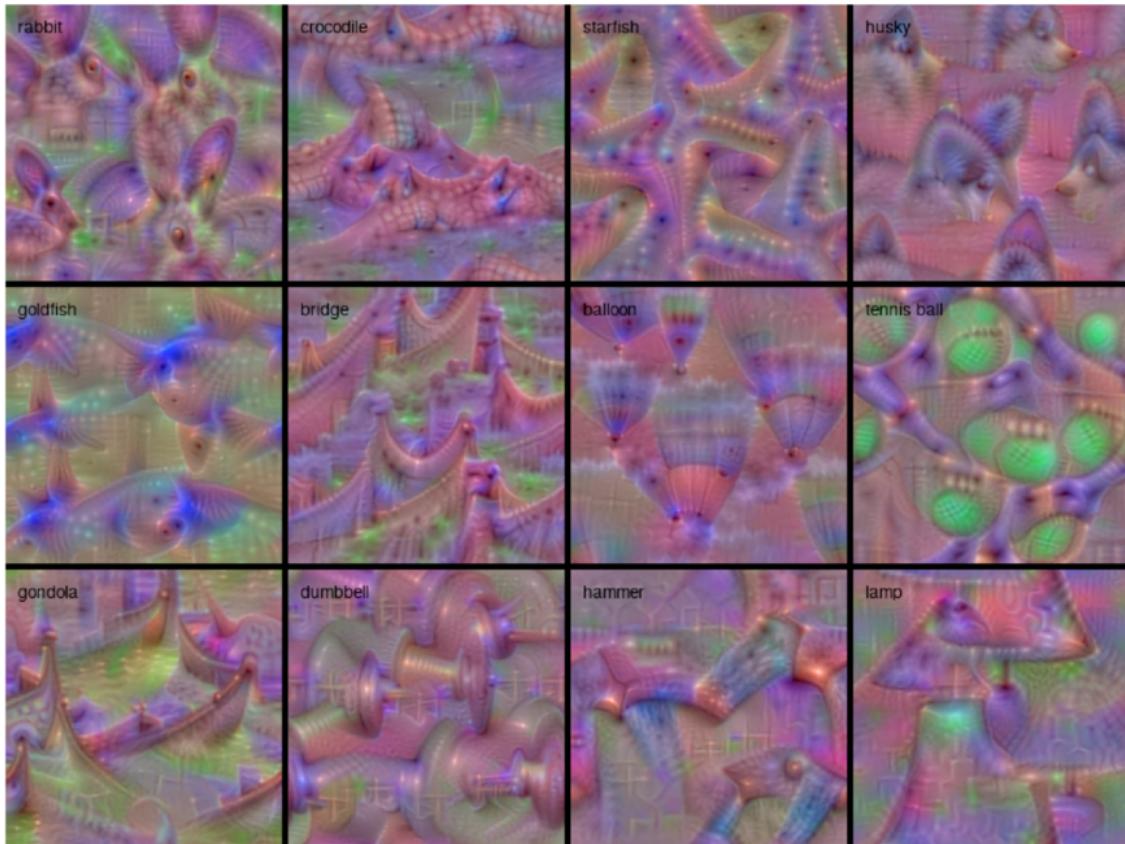
8.4.3 Visualizing Class Outputs

Giống như cách mà ta visualize filter, trong phần này ta sẽ thực hiện visualize ở lớp cuối cùng là lớp softmax. Như cách làm trước, ta tạo ra hình ảnh khiến cho giá trị ở class mong muốn đạt cực đại. Chẳng hạn như yêu cầu mô hình CNN tạo ra hình mà khi đi qua softmax được giá trị ở lớp "cái đèn" đạt cực đại. Ví dụ như trong hình 8.17

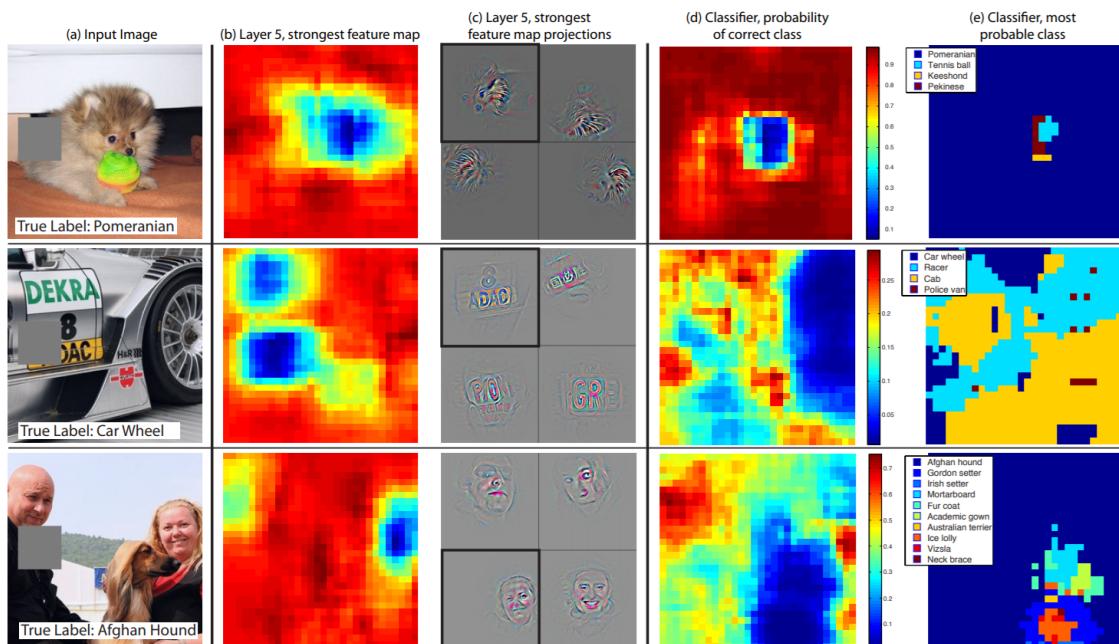
Khá là dễ hiểu khi những hình được lặp lại nhiều lần làm mô hình cảm thấy chắc chắn hơn. Keras có một thư viện riêng dùng cho việc visualize, các bạn có thể tham khảo [tại đây](#)

8.4.4 Visualizing Attention Map

Một cách rất trực quan để visualize một mạng CNN dùng phân loại ảnh là Attention Map. Giả sử ta có một mạng CNN để phân loại ảnh, làm thế nào và tại sao mạng CNN lại đưa ảnh đó vào lớp chó? Liệu có phải mạng CNN chú ý hơn vào những pixel liên quan đến chó? Để biết điều này ta xoá đi một phần của ảnh, sau đó đưa vào mạng CNN phân loại ảnh để xem xác suất rơi vào class nào và bao nhiêu. Khi lặp lại đến hết toàn bộ ảnh, ta có thể vẽ được heatmap với vùng càng nóng thì xác suất mô hình dự đoán đúng càng cao, hay phần bị che đi càng ít quan trọng. Kỹ thuật này được đề xuất bởi [Matthew Zeiler](#)



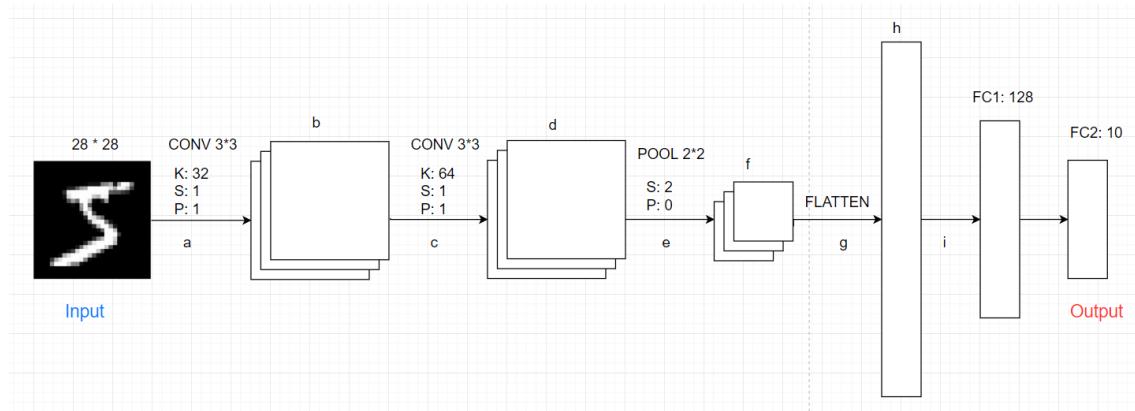
Hình 8.17: Những hình khiết cho mỗi lớp tương ứng đạt giá trị cực đại



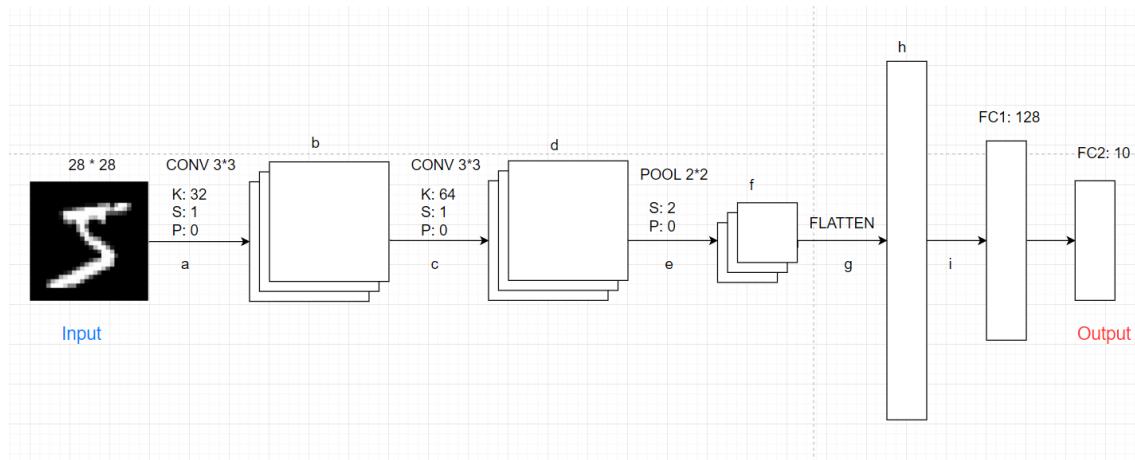
Hình 8.18: Ba ví dụ áp dụng kỹ thuật nêu trên, phần bị che đi dùng một hình vuông màu xám (a) và xem feature map ở layer 5 chú ý đến gì (b & c), cột thứ 4 là attention map, ví dụ khi mặt của con chó Pom bị che thì tỉ lệ rơi vào class chó Pom gần như bằng không và mô hình CNN dự đoán là bóng tennis hoặc loài chó khác (e)

8.5 Bài tập

1. Convolutional layer để làm gì? Lại sao mỗi layer cần nhiều kernel?
2. Hệ số của convolutional layer là gì?
3. Tự tính lại số lượng parameter, output size của convolutional layer với stride và padding trong trường hợp tổng quát.
4. Hình dưới là mô hình nhận diện chữ số MNIST, K: số kernel, S: stride, P: padding. Tính số lượng parameter ở layer và output tương ứng (Tìm a, b, c, d, e, f, g, h, i).



(a)



(b)

5. Tại sao cần flatten trong CNN?
6. Tại sao trong model VGG16, ở layer càng sâu thì width, height giảm nhưng depth lại tăng.
7. Tính backpropagation qua convolutional layer và pooling layer.

9. Giới thiệu keras và bài toán phân loại ảnh

Bài trước đã giới thiệu về convolutional neural network (CNN) cho bài toán với input là ảnh. Bài này sẽ giới thiệu về thư viện keras và ứng dụng keras để xây dựng mô hình CNN cho bài toán phân loại ảnh.

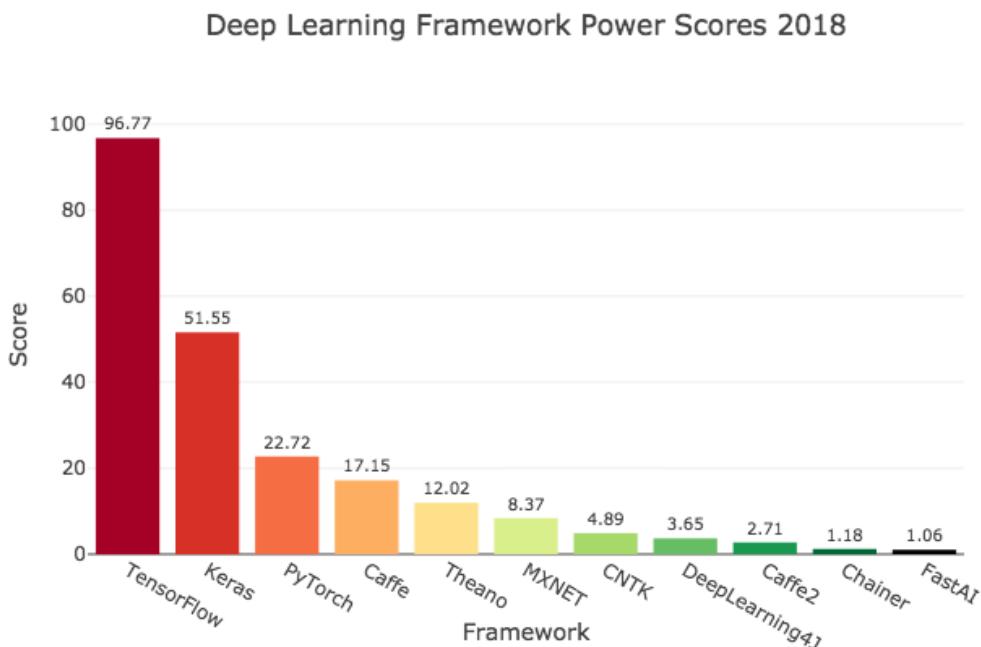
9.1 Giới thiệu về keras

Bản chất của bài toán Deep learning: Bạn có dữ liệu, bạn muốn máy tính học được các mô hình (model) từ dữ liệu, sau đó dùng mô hình đấy để dự đoán được các dữ liệu mới. Các bước cơ bản làm một bài toán deep learning :

1. Xây dựng bài toán
2. Chuẩn bị dữ liệu (dataset)
3. Xây dựng model
4. Định nghĩa loss function
5. Thực hiện backpropagation và áp dụng gradient descent để tìm các parameter gồm weight và bias để tối ưu loss function.
6. Dự đoán dữ liệu mới bằng model với các hệ số tìm được ở trên

Bước xây dựng model thì áp dụng các kiến thức được trình bày trong bài neural network và convolutional neural network ta có thể xây dựng model hoàn chỉnh từ đầu bằng python. Tuy nhiên bước backpropagation trở nên phức tạp hơn rất nhiều. Khó để implement và tối ưu được tốc độ tính toán. Đây là lý do các framework về deep learning ra đời với các đặc điểm:

- Người dùng chỉ cần định nghĩa model và loss function, framework sẽ lo phần backpropagation.
- Việc định nghĩa layer, activation function, loss function đơn giản hơn cho người dùng. Ví dụ để thêm layer trong neural network chỉ cần báo là layer có bao nhiêu node và dùng hàm activation gì.



Hình 9.1: Các deep learning framework phổ biến [5]

Có thể thấy tensorflow là framework phổ biến nhất tuy nhiên tensorflow khá khó sử dụng cho người mới bắt đầu. Nên tôi sẽ giới thiệu về keras: dễ sử dụng, thân thiện với người dùng nhưng đủ tốt để làm các bài toán về deep learning.

Keras là một framework mã nguồn mở cho deep learning được viết bằng Python. Nó có thể chạy trên nền của các deep learning framework khác như: tensorflow, theano, CNTK. Với các API bậc cao, dễ sử dụng, dễ mở rộng, keras giúp người dùng xây dựng các deep learning model một cách đơn giản.

9.2 MNIST Dataset

9.2.1 Xây dựng bài toán

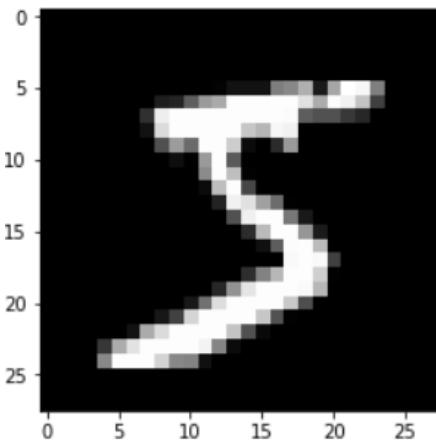
Bạn có ảnh xám kích thước 28*28 của chữ số từ 1 đến 9 và bạn muốn dự đoán số đây là số mấy. Ví dụ:

9.2.2 Chuẩn bị dữ liệu

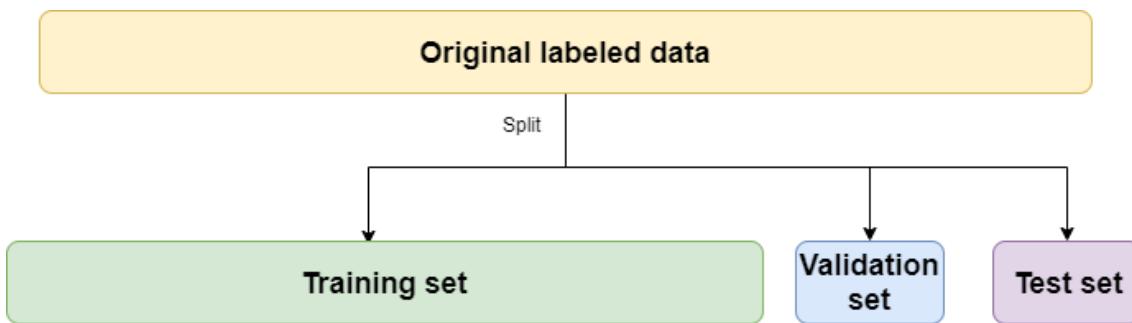
MNIST là bộ cơ sở dữ liệu về chữ số viết tay, bao gồm 2 tập con: training set gồm 60.000 ảnh các chữ số viết tay và test set gồm 10.000 ảnh các chữ số.

Training set, test set là gì? Mục đích cuối cùng của machine learning không phải là predict tốt với dữ liệu mình đang có mà là để predict tốt các dữ liệu mới khi đi vào thực tế, vậy nên mình mong muốn model học được tổng quan hóa dữ liệu (generalization) thay vì chỉ nhớ các dữ liệu trong dataset. Để đánh giá xem model có học không hay chỉ nhớ cũng như khi dùng ngoài thực tế thì performance sẽ thế nào, người ta chia dataset làm 3 tập training set, validation set và test set.

Mình sẽ cho model học trên tập training set và đánh giá model trên tập validation set. Nếu



Hình 9.2: Dữ liệu đầu tiên trong MNIST dataset.



Hình 9.3: Chia các tập dữ liệu

có nhiều hơn 1 mô hình (ví dụ VGG16, VGG19,...) thì mô hình nào cho performance tốt hơn trên tập validation set sẽ được chọn. Và cuối cùng model tốt nhất sẽ được đánh giá trên tập test set làm hiệu suất của model khi dùng thực tế. Nhận thấy là tập test set không được dùng trong cả quá trình training chỉ đến cuối dùng để đánh giá.

Giả sử bạn đang luyện thi đại học và bạn có 10 bộ đề để luyện thi. Nếu bạn học và chưa cả 10 đề một cách chi tiết thì bạn sẽ không thể ước lượng được điểm thi của bạn khoảng bao nhiêu, dẫn đến bạn không chọn được trường phù hợp. Thế là bạn nghĩ ra một giải pháp tốt hơn, trong 10 đề đầy chỉ lấy 8 đề học và chưa chi tiết thôi còn để 2 đề lại coi như là đề thi thật. Như vậy bạn có thể ước lượng điểm thi của mình bằng cách đánh giá điểm ở 2 đề đầy.

Trong ví dụ trên thì:

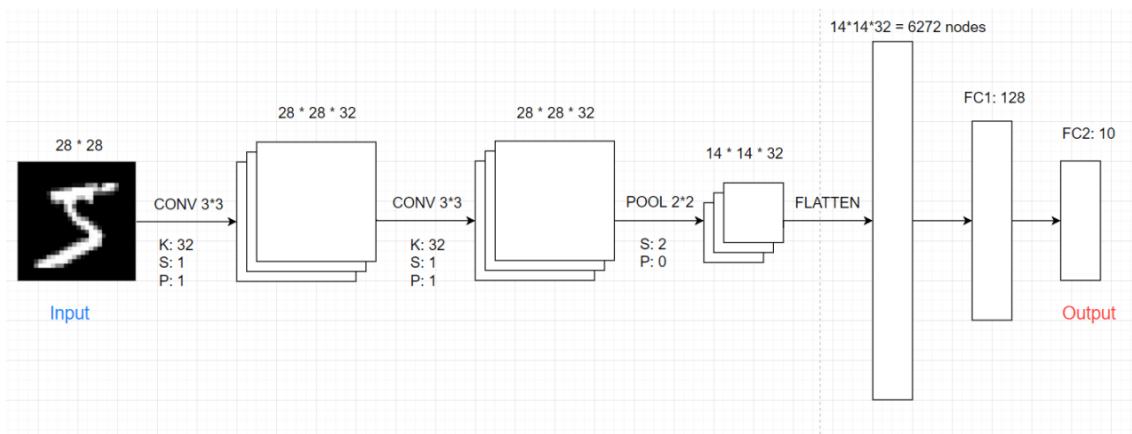
- 8 đề ôn luyện được gọi là training set, có thể hiểu là dữ liệu dùng để dạy cho model học.
- 2 đề để rèn gọi là validation set, là để đánh giá xem model hiện tại có tốt không, thường được dùng để chỉnh các tham số của model.
- đề thi đại học thật là test set, là để đánh giá xem model hoạt động với dữ liệu thực tế có tốt không.

Như vậy MNIST dataset có 60.000 dữ liệu ở training set ở trong MNIST, ta sẽ chia ra 50.000 dữ liệu cho training set và 10.000 dữ liệu cho validation set. Vẫn giữ nguyên 10.000 dữ liệu của test set.

9.2.3 Xây dựng model

Vì input của model là ảnh nên nghĩ ngay đến convolutional neural network (CNN).

Mô hình chung bài toán CNN: Input image -> Convolutional layer (Conv) + Pooling layer (Pool) -> Fully connected layer (FC) -> Output.



Hình 9.4: Model cho bài toán

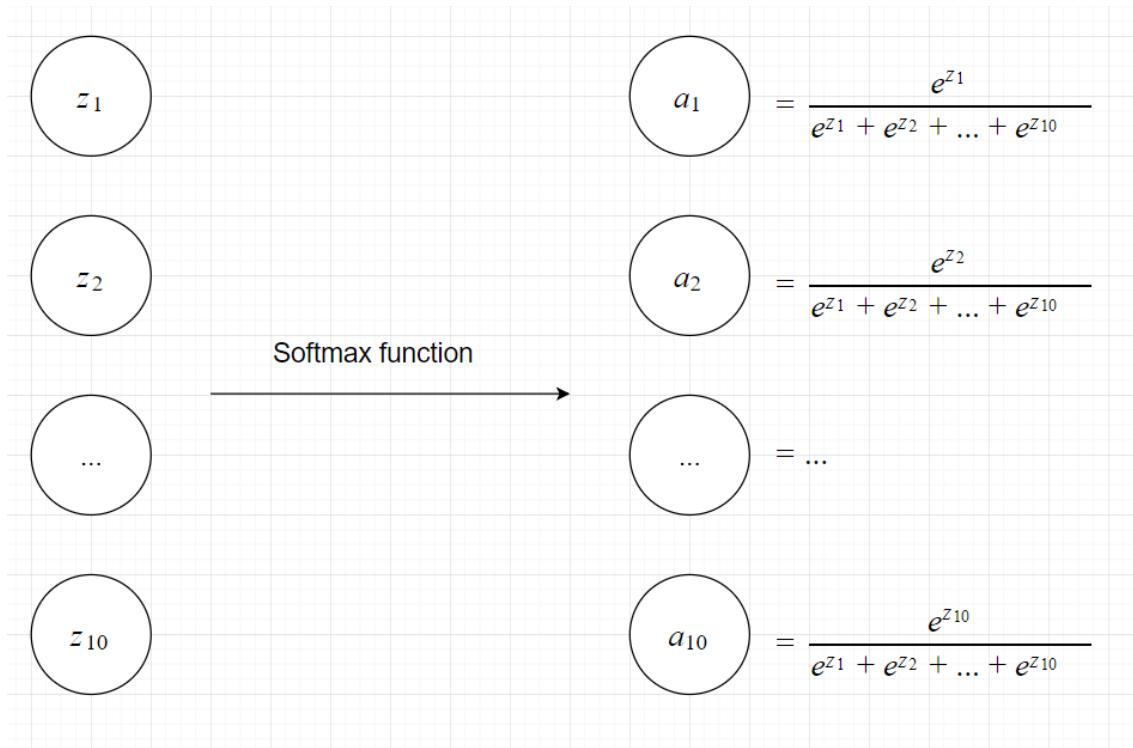
Input của model là ảnh xám kích thước 28*28.

Softmax function

Giống như bài logistic regression, thay vì chỉ muốn kết quả là ảnh là số mấy, ta muốn dự đoán phần trăm của ảnh là số nào. Ví dụ: 90% ảnh là số 5, 1% ảnh là số 1,...

Nhắc lại bài neural network, ở mỗi layer sẽ thực hiện 2 bước: tính tổng linear các node ở layer trước và thực hiện activation function (ví dụ sigmoid function, softmax function). Do sau bước tính tổng linear cho ra các giá trị thực nên cần dùng softmax function dùng để chuyển đổi giá trị thực trong các node ở output layer sang giá trị phần trăm.

Vì mỗi ảnh sẽ thuộc 1 class từ 0 đến 9, nên tất cả sẽ có 10 class. Nên output layer sẽ có 10 node để tương ứng với phần trăm ảnh là số 0,1,...,9. Ví dụ: a_6 là xác suất ảnh là số 5. (Sự khác biệt chỉ số do các số bắt đầu từ 0 trong khi chỉ số của node trong layer bắt đầu từ 1)



Hình 9.5: Softmax function

Tổng quát sau hàm activation: $a_k = \frac{e^{z_k}}{\sum_{i=1}^{10} e^{z_i}}$.

Nhận xét:

- $\sum_{i=1}^{10} a_i = 1$
- $0 < a_i < 1$

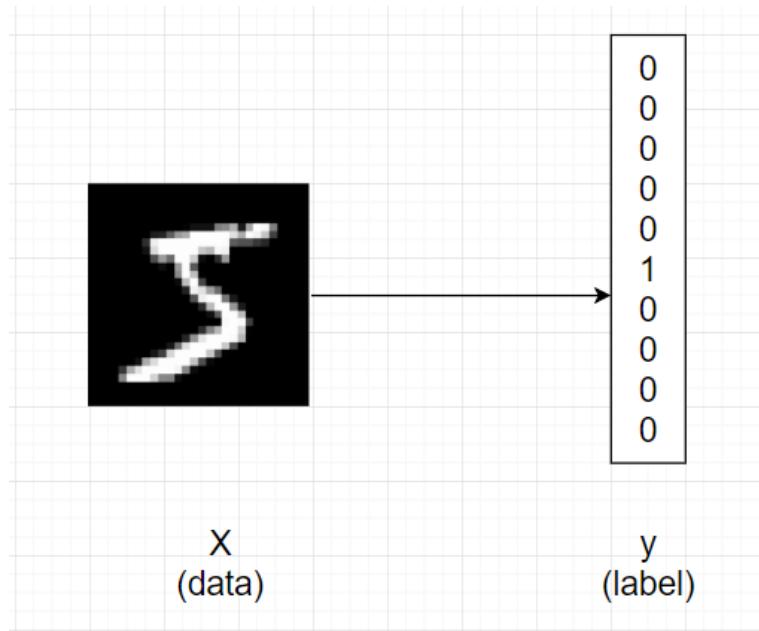
Do đó ta có thể coi a_i là xác suất ảnh là số (i-1).

Với các bài toán classification (phân loại) thì nếu có 2 lớp thì hàm activation ở output layer là hàm sigmoid và hàm loss function là binary_crossentropy, còn nhiều hơn 2 lớp thì hàm activation ở output layer là hàm softmax với loss function là hàm categorical_crossentropy

=> Output layer có 10 nodes và activation là softmax function.

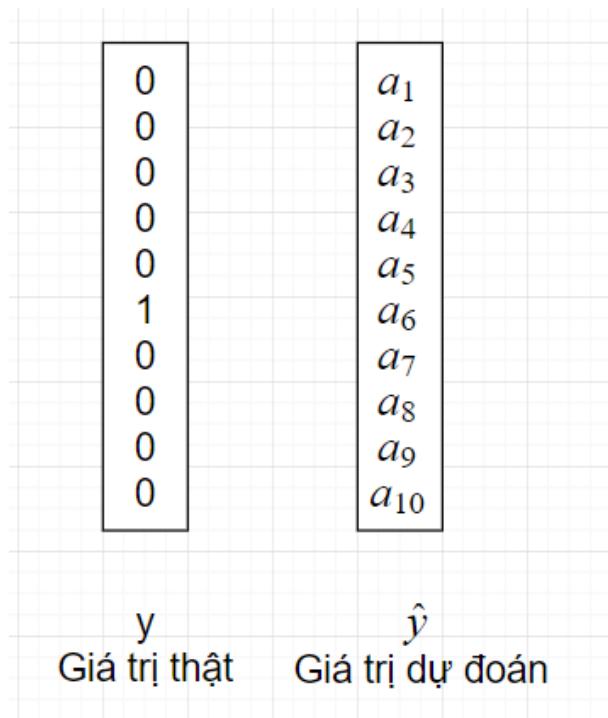
9.2.4 Loss function

Để định nghĩa loss function, trước hết ta dùng one-hot encoding chuyển đổi label của ảnh từ giá trị số sang vector cùng kích thước với output của model. Ví dụ:



Để ý là label của data là số i là vector v kích thước 10×1 với $v_{i+1} = 1$ và các giá trị khác bằng 0. So với quy ước về phần trăm ở trên thì one-hot encoding có ý nghĩa là ta chắc chắn 100% ảnh này là số 5.

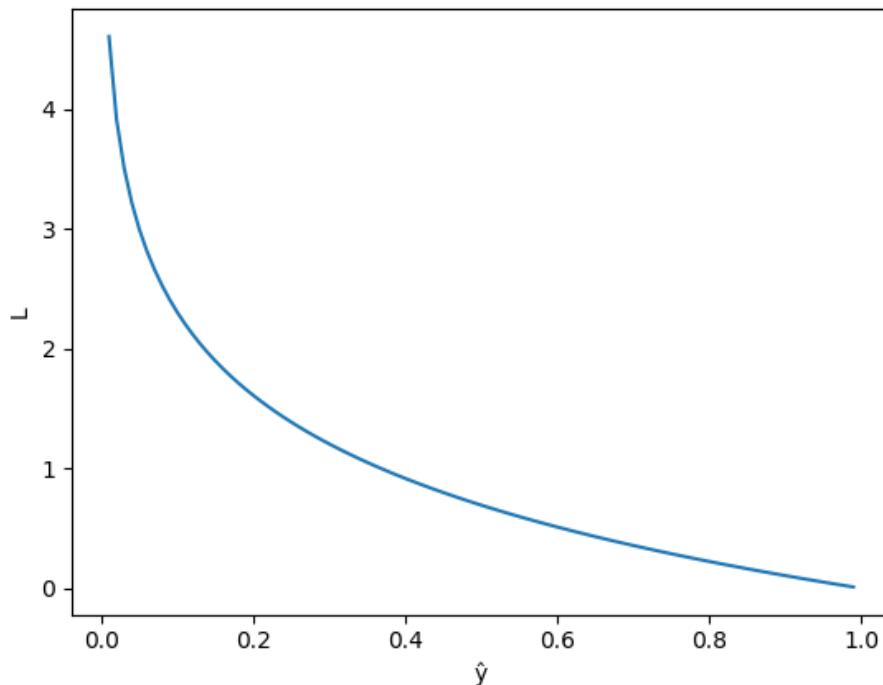
Giờ ta có giá trị thật (label) dạng one-hot encoding giá trị dự đoán ở output layer sau hàm softmax function cùng kích thước 10×1 . Ta cần định nghĩa hàm loss function để đánh giá độ tốt của model.



Mong muốn là a_6 gần 1 còn các giá trị a_i khác gần 0 vì như thế nghĩa là model dự đoán đúng được ảnh đầu vào là ảnh số 5. Ta định nghĩa loss function:

$$L = - \sum_{i=1}^{10} y_i * \log(\hat{y}_i)$$

Thử đánh giá hàm L. Giả sử ảnh là số 5 thì $L = -\log(\hat{y}_6)$.



Nhận xét:

- Hàm L giảm dần từ 0 đến 1
 - Khi model dự đoán \hat{y}_6 gần 1, tức giá trị dự đoán gần với giá trị thật y_6 thì L nhỏ, xấp xỉ 0
 - Khi model dự đoán \hat{y}_6 gần 0, tức giá trị dự đoán ngược lại giá trị thật y_6 thì L rất lớn
- => Hàm L nhỏ khi giá trị model dự đoán gần với giá trị thật và rất lớn khi model dự đoán sai, hay nói cách khác L càng nhỏ thì model dự đoán càng gần với giá trị thật. => Bài toán tìm model trở thành tìm giá trị nhỏ nhất của L.

Hàm loss function định nghĩa như trên trong keras gọi là "categorical_crossentropy"

9.3 Python code

```
@author: DELL
# 1. Thêm các thư viện cần thiết
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist

# 2. Load dữ liệu MNIST
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_val, y_val = X_train[50000:60000, :], y_train[50000:60000]
X_train, y_train = X_train[:50000, :], y_train[:50000]
```

```

print(X_train.shape)

# 3. Reshape lại dữ liệu cho đúng kích thước mà keras yêu cầu
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_val = X_val.reshape(X_val.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)

# 4. One hot encoding label (Y)
Y_train = np_utils.to_categorical(y_train, 10)
Y_val = np_utils.to_categorical(y_val, 10)
Y_test = np_utils.to_categorical(y_test, 10)
print('Dữ liệu y ban đầu ', y_train[0])
print('Dữ liệu y sau one-hot encoding ', Y_train[0])

# 5. Định nghĩa model
model = Sequential()

# Thêm Convolutional layer với 32 kernel, kích thước kernel 3*3
# dùng hàm sigmoid làm activation và chỉ rõ input_shape cho layer đầu tiên
model.add(Conv2D(32, (3, 3), activation='sigmoid', input_shape=(28,28,1)))

# Thêm Convolutional layer
model.add(Conv2D(32, (3, 3), activation='sigmoid'))

# Thêm Max pooling layer
model.add(MaxPooling2D(pool_size=(2,2)))

# Flatten layer chuyển từ tensor sang vector
model.add(Flatten())

# Thêm Fully Connected layer với 128 nodes và dùng hàm sigmoid
model.add(Dense(128, activation='sigmoid'))

# Output layer với 10 node và dùng softmax function để chuyển sang xác suất.
model.add(Dense(10, activation='softmax'))

# 6. Compile model, chỉ rõ hàm loss_function nào được sử dụng, phương thức
# để tối ưu hàm loss function.
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# 7. Thực hiện train model với data
H = model.fit(X_train, Y_train, validation_data=(X_val, Y_val),
               batch_size=32, epochs=10, verbose=1)

# 8. Vẽ đồ thị loss, accuracy của training set và validation set
fig = plt.figure()
numOfEpoch = 10

```

```

plt.plot(np.arange(0, num0fEpoch), H.history['loss'], label='training loss')
plt.plot(np.arange(0, num0fEpoch), H.history['val_loss'], label='validation loss')
plt.plot(np.arange(0, num0fEpoch), H.history['acc'], label='accuracy')
plt.plot(np.arange(0, num0fEpoch), H.history['val_acc'], label='validation accuracy')
plt.title('Accuracy and Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss|Accuracy')
plt.legend()

# 9. Đánh giá model với dữ liệu test set
score = model.evaluate(X_test, Y_test, verbose=0)
print(score)

# 10. Dự đoán ảnh
plt.imshow(X_test[0].reshape(28,28), cmap='gray')

y_predict = model.predict(X_test[0].reshape(1,28,28,1))
print('Giá trị dự đoán: ', np.argmax(y_predict))

```

9.4 Ứng dụng của việc phân loại ảnh

- Chuẩn đoán ảnh X-ray của bệnh nhân có bị ung thư hay không
- Phân loại, nhận diện được các chữ, số viết tay => tự động đọc được biển số xe, văn bản.
- Phân loại được các biển báo giao thông => hỗ trợ cho ô tô tự lái

9.5 Bài tập

1. Tại sao cần dùng softmax activation ở layer cuối cùng?
2. (a) Thiết kế và training model CNN phân loại chó, mèo, nguồn dữ liệu <https://www.kaggle.com/c/dogs-vs-cats/data>
 (b) Thay đổi learning rate, epoch, batch size xem accuracy thay đổi thế nào? Thử giải thích tại sao?
 (c) Thay đổi model bằng cách thêm lần lượt các layer pooling, dropout, batch normalization, activation function và thay đổi kernel size bằng (5x5), (7x7) xem số lượng parameter, accuracy thay đổi thế nào? Thử giải thích tại sao?
3. Xây dựng model CNN cho bài toán phân loại ảnh với dữ liệu CIFAR10 dataset bao gồm 50,000 training set và 10.000 test set ảnh màu kích thước 32x32 cho 10 thể loại khác nhau (máy bay, ô tô, thuyền, chim, chó, mèo, ngựa,...).

```

# Load dữ liệu cifar10
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

```




10. Ứng dụng CNN cho ô tô tự lái

Bài trước đã giới thiệu về thư viện keras và dùng keras để xây dựng mô hình convolutional neural network (CNN) cho bài toán phân loại ảnh. Bài này sẽ dùng mô hình CNN để dự đoán các giá trị thực và áp dụng cho bài toán ô tô tự lái.

10.1 Giới thiệu mô phỏng ô tô tự lái

Ứng dụng mô phỏng ô tô tự lái là phần mềm mã nguồn mở được phát triển bởi Udacity, được viết bằng Unity (công cụ dùng để phát triển game).



Mọi người tải phần mềm ở [đây](#). Kéo xuống dưới và chọn hệ điều hành tương ứng.

[README.md](#)

Welcome to Udacity's Self-Driving Car Simulator

This simulator was built for [Udacity's Self-Driving Car Nanodegree](#), to teach students how to train cars how to navigate road courses using deep learning. See more [project details here](#).

All the assets in this repository require Unity. Please follow the instructions below for the full setup.

Available Game Builds (Precompiled builds of the simulator)

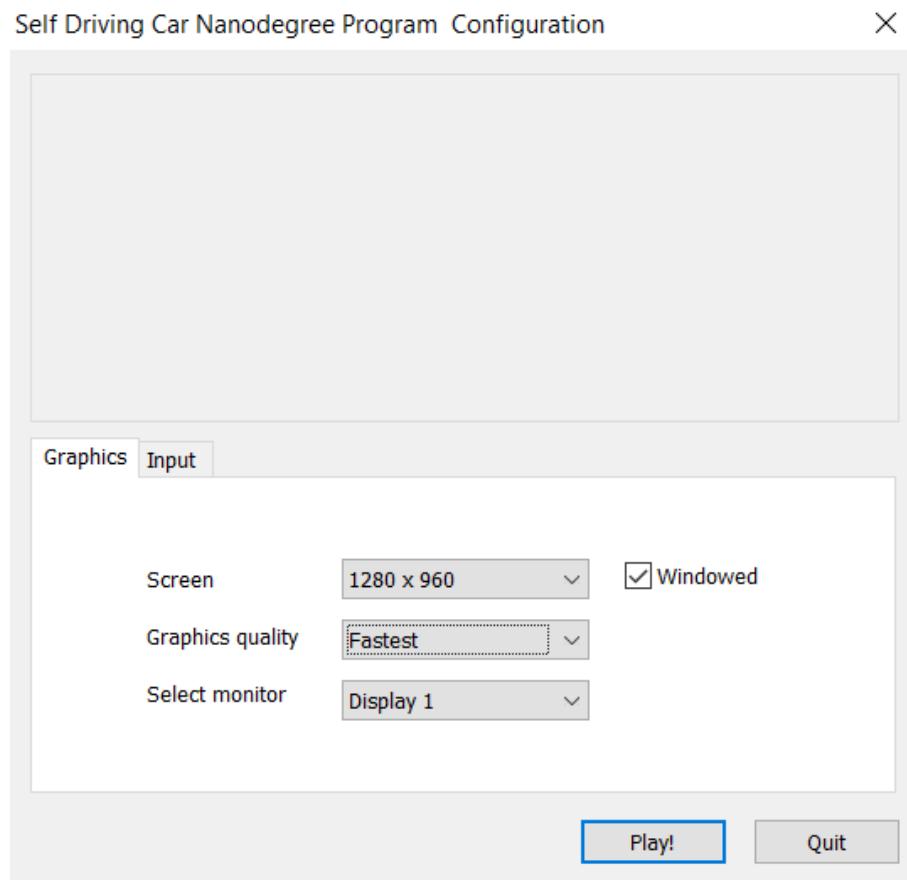
Term 1

Instructions: Download the zip file, extract it and run the executable file.

Version 2, 2/07/17

[Linux](#) [Mac](#) [Windows](#)

Sau khi tải về, giải nén thì bạn có thể chạy được luôn như trên window thì bạn sẽ thấy file .exe



Hình 10.1: Chọn kích thước, cấu hình (máy khỏe thì để nét hơn không thì cứ để Fastest)

Sau đó bạn chọn Play!



Bạn sẽ thấy có 2 chế độ: training mode (sẽ cho dữ liệu về ô tô tự lái để train mô hình) và autonomous mode (sau khi học được mô hình rồi thì đây là phần ô tô tự lái).

Ở ô tô có gắn 3 cameras (trái, giữa, phải)

Ô tô có thể di chuyển sang bên trái (\leftarrow), sang bên phải (\rightarrow), tăng tốc (\uparrow), giảm tốc (\downarrow)

Trong phần training mode, ở mỗi vị trí ô tô di chuyển thì sẽ cho ta các dữ liệu: ảnh ở 3 camera, góc lái của vô lăng, tốc độ xe, độ giảm tốc (throttle) và phanh (brake)

10.2 Bài toán ô tô tự lái

10.2.1 Xây dựng bài toán

Bạn muốn dự đoán góc lái của vô lăng bằng ảnh ở camera trên ô tô. Vì input là ảnh nên nghĩ ngay đến việc dùng CNN.

Dữ liệu training set sẽ được lấy từ training mode của phần mềm mô phỏng.

10.2.2 Chuẩn bị dữ liệu

Sau khi bạn chọn training mode thì hãy làm quen với việc di chuyển của ô tô bằng các phím mũi tên. Đến khi bạn lái mượt rồi thì chọn nút record. Chọn folder bạn muốn lưu dữ liệu và chọn select.

Bạn lái xe khoảng 10 phút sẽ ra khoảng 18000 ảnh (6000 ảnh từ mỗi camera).

Bạn cũng thấy file driving_log.csv để mô tả dữ liệu

đường dẫn ảnh camera giữa	ảnh camera trái	ảnh camera phải	góc lái	độ giảm tốc	phanh	tốc độ
---------------------------	-----------------	-----------------	---------	-------------	-------	--------

10.2.3 Tiền xử lý dữ liệu (Preprocessing)

Ảnh màu từ camera ở ô tô có kích thước 320*160



Để tăng lượng ảnh cho việc training, với các ảnh ở camera giữa ta dùng góc lái như trong file csv. Tuy nhiên với ảnh camera trái ta tăng góc lái lên 0.2 và ảnh camera phải ta giảm góc lái đi 0.2.

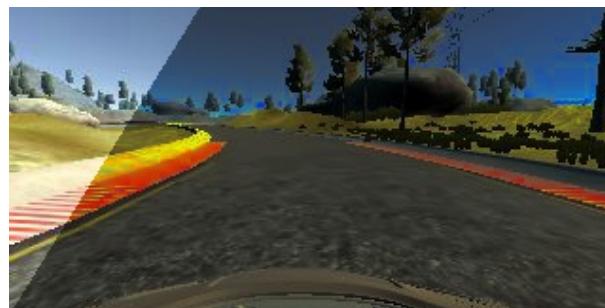
Thêm vào đó ta có thể lật ngược ảnh lại và đổi dấu góc lái của vô lăng.



Tiếp, ta có thể dịch chuyển ảnh hoặc thêm độ sáng tối cho ảnh và giữ nguyên góc lái



Hình 10.2: Thêm sáng



Hình 10.3: Thêm tối



Hình 10.4: Dịch chuyển

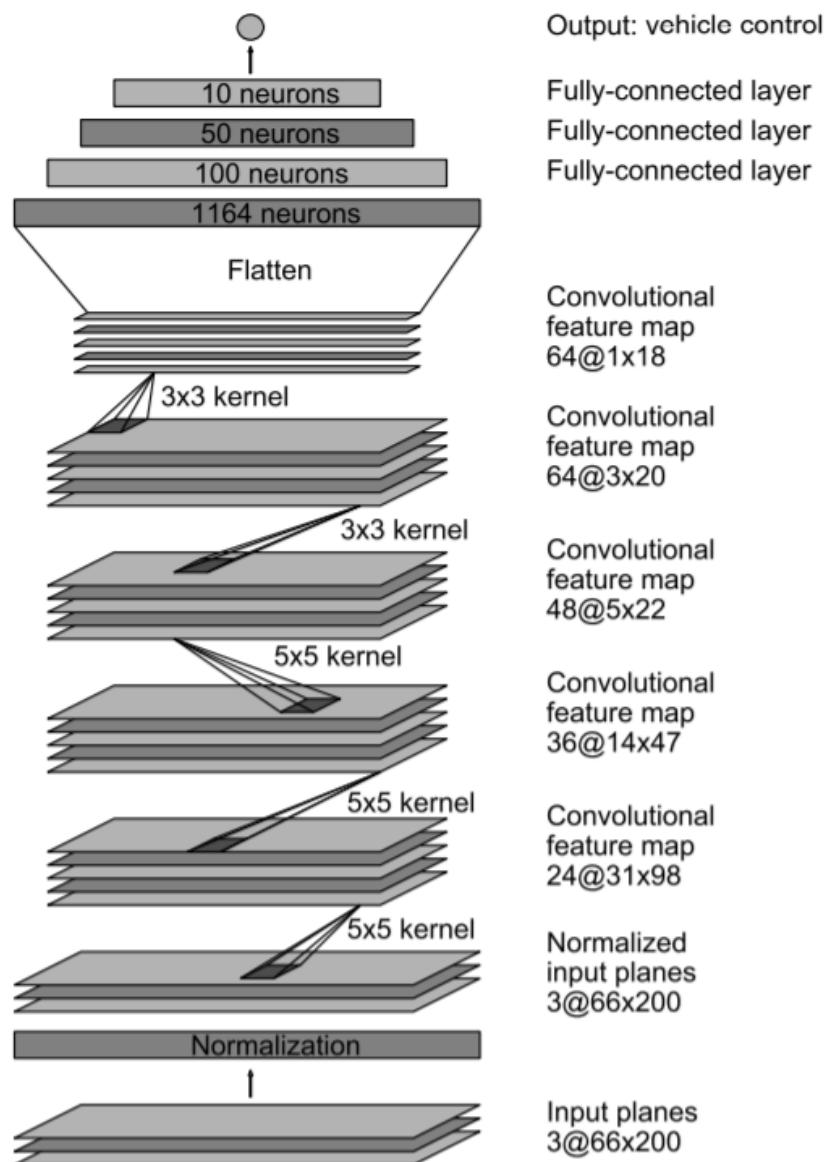
Quá trình thêm ảnh như trên gọi là data augmentation để tránh overfitting, hiểu đơn giản là khi luyện thi thay vì chỉ luyện kĩ 10 đề thì lên mạng tải thêm 100 đề nữa để làm và chữa chi tiết như vậy khi thi đại học thật điểm có thể sẽ tốt hơn vì làm nhiều đề và gặp nhiều dạng bài hơn.

Phần mũi xe ô tô và bầu trời thì không liên quan đến việc dự đoán góc lái vô lăng nên ta sẽ cắt bỏ đi.



Do dùng CNN, các ảnh input cần cùng kích thước nên sẽ reshape ảnh thành kích thước 66*200.

10.2.4 Xây dựng model



Hình 10.5: Model của bài toán [6]

Input layer: Ảnh màu kích thước 66*200

Output layer: 1 node dự đoán góc lái của vô lăng

10.2.5 Loss function

Vì giá trị dự đoán là giá trị thực nên ta sẽ dùng mean square error giống như trong bài 1 khi dự đoán giá nhà.

10.3 Python code

```
# -*- coding: utf-8 -*-

import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from keras.models import Sequential
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint
from keras.layers import Lambda, Conv2D, Dropout, Dense, Flatten
from keras.regularizers import l2

from utils import INPUT_SHAPE, batch_generator

# Thư mục để dữ liệu
data_dir = 'dataset'
# Đọc file driving_log.csv với các cột tương ứng
data_df = pd.read_csv(os.path.join(os.getcwd(), data_dir, 'driving_log.csv'), \
                     names=['center', 'left', 'right', 'steering', 'throttle', 'reverse', 'speed'])

# Lấy đường dẫn đến ảnh ở camera giữa, trái, phải
X = data_df[['center', 'left', 'right']].values
# Lấy góc lái của ô tô
y = data_df['steering'].values

# Vẽ histogram dữ liệu
plt.hist(y)

# Loại bỏ và chỉ lấy 1000 dữ liệu có góc lái ở 0
pos_zero = np.array(np.where(y==0)).reshape(-1, 1)
pos_none_zero = np.array(np.where(y!=0)).reshape(-1, 1)
np.random.shuffle(pos_zero)
pos_zero = pos_zero[:1000]

pos_combined = np.vstack((pos_zero, pos_none_zero))
pos_combined = list(pos_combined)
```

```

y = y[pos_combined].reshape(len(pos_combined))
X = X[pos_combined, :].reshape((len(pos_combined), 3))

# After process
plt.hist(y)

# Chia ra traing set và validation set
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2, \
                                                     random_state=0)

# Xây dựng model
model = Sequential()
model.add(Lambda(x: x/127.5-1.0, input_shape=INPUT_SHAPE))
model.add(Conv2D(24, 5, 5, activation='elu', subsample=(2, 2)))
model.add(Conv2D(36, 5, 5, activation='elu', subsample=(2, 2)))
model.add(Conv2D(48, 5, 5, activation='elu', subsample=(2, 2)))
model.add(Conv2D(64, 3, 3, activation='elu'))
model.add(Conv2D(64, 3, 3, activation='elu'))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(100, activation='elu'))
model.add(Dropout(0.5))
model.add(Dense(50, activation='elu'))
model.add(Dense(10, activation='elu'))
model.add(Dense(1))
model.summary()

nb_epoch = 10
samples_per_epoch = 1000
batch_size = 32
save_best_only = True
learning_rate = 1e-4

# Checkpoint này để nói cho model lưu lại model nếu validation loss thấp nhất
checkpoint = ModelCheckpoint('models/model-{epoch:03d}.h5',
                             monitor='val_loss',
                             verbose=0,
                             save_best_only=save_best_only,
                             mode='auto')

# Dùng mean_squared_error làm loss function
model.compile(loss='mean_squared_error', optimizer=Adam(lr=learning_rate))

# Train model
H = model.fit_generator(batch_generator(data_dir, X_train, y_train, batch_size, True),
                       steps_per_epoch = samples_per_epoch,
                       epochs = nb_epoch,
                       max_q_size=1,
                       validation_data=batch_generator(data_dir, X_valid, y_valid, \

```

```
batch_size, False),
nb_val_samples=len(X_valid),
callbacks=[checkpoint],
verbose=1)
```

10.4 Áp dụng model cho ô tô tự lái

Khi ta bật chế độ autonomous mode trong phần mềm mô phỏng, ta sẽ dùng socket để lấy được dữ liệu từ camera giữa của ô tô sau đó dùng model để dự đoán góc lái. Cuối cùng ta sẽ chuyển lại dữ liệu về góc lái, tốc độ lái cho phần mềm để ô tô di chuyển.

Code cụ thể trong file drive.py trong github nhưng đây là những phần quan trọng nhất

```
# Load model mà ta đã train được từ bước trước
model = load_model('model.h5')

# Góc lái hiện tại của ô tô
steering_angle = float(data["steering_angle"])
# Tốc độ hiện tại của ô tô
speed = float(data["speed"])
# Ảnh từ camera giữa
image = Image.open(BytesIO(base64.b64decode(data["image"])))
try:
    # Tiền xử lý ảnh, cắt, reshape
    image = np.asarray(image)
    image = utils.preprocess(image)
    image = np.array([image])
    print('*'*50)
    steering_angle = float(model.predict(image, batch_size=1))

    # Tốc độ ta để trong khoảng từ 10 đến 25
    global speed_limit
    if speed > speed_limit:
        speed_limit = MIN_SPEED  # giảm tốc độ
    else:
        speed_limit = MAX_SPEED
    throttle = 1.0 - steering_angle**2 - (speed/speed_limit)**2

    print('{} {} {}'.format(steering_angle, throttle, speed))

    # Gửi lại dữ liệu về góc lái, tốc độ cho phần mềm để ô tô tự lái
    send_control(steering_angle, throttle)
```

Đầu tiên các bạn mở phần mềm chọn autonomous mode, ô tô sẽ đứng yên. Sau đó mở command line chạy python drive.py model.h5 và ô tô sẽ bắt đầu tự lái. model.h5 là tên model mà bạn đã lưu ở bước trên.

10.5 Bài tập

- Model được xây dựng chỉ dựa trên góc lái hay mở rộng model để có thể dự đoán cả các thông số khác như góc lái, tốc độ...

V

Deep Learning Tips

11 Transfer learning và data augmentation 161

- 11.1 Transfer learning
- 11.2 Data augmentation
- 11.3 Bài tập

12 Các kỹ thuật cơ bản trong deep learning 177

- 12.1 Vectorization
- 12.2 Mini-batch gradient descent
- 12.3 Bias và variance
- 12.4 Dropout
- 12.5 Activation function
- 12.6 Batch Normalize
- 12.7 Bài tập

11. Transfer learning và data augmentation

Bài này sẽ giới thiệu các kĩ thuật để giải quyết khi bạn không có đủ dữ liệu cho việc training model.

11.1 Transfer learning

Bạn có bài toán cần nhận diện 1000 người nổi tiếng ở Việt Nam, tuy nhiên dữ liệu để train chỉ khoảng 10 ảnh / 1 người. Số lượng dữ liệu là quá ít để train một mô hình CNN hoàn chỉnh.

Bạn tìm trên mạng thấy VGGFace2 dataset có 3.31 triệu ảnh của 9131 người, với trung bình 362.6 ảnh cho mỗi người. Họ làm bài toán tương tự đó là nhận diện ảnh từng người và họ đã train được CNN model với accuracy hơn 99%.

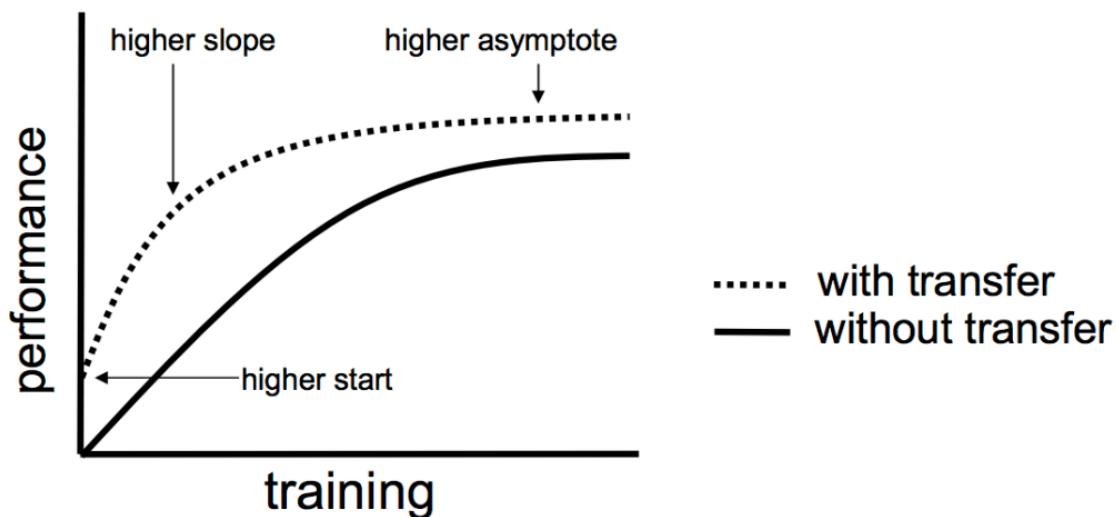
Bạn nhớ ra là trong convolutional neural network, convolutional layer có tác dụng lấy ra các đặc trưng của ảnh, và sau hàng loạt các convolutional layer + pooling layer (ConvNet) thì model sẽ học được các đặc điểm của ảnh, trước khi được cho vào fully connected layer => ConvNet trong VGGFace2 model cũng lấy ra được các đặc điểm của mặt người (tai, mũi, tóc,...) => Ta cũng có thể áp dụng phần ConvNet của VGGFace2 model vào bài toán nhận diện mặt người nổi tiếng ở Việt Nam để lấy ra các đặc điểm của mặt.

Quá trình sử dụng pre-trained model như trên gọi là transfer learning.

Các pre-trained model được sử dụng thường là các bài toán được train với dữ liệu lớn ví dụ ImageNet, dữ liệu chứa 1.2 triệu ảnh với 1000 thể loại khác nhau.

Có 2 loại transfer learning:

- **Feature extractor:** Sau khi lấy ra các đặc điểm của ảnh bằng việc sử dụng ConvNet của pre-trained model, thì ta sẽ dùng linear classifier (linear SVM, softmax classifier,...) để phân loại ảnh. Hiểu đơn giản thì các đặc điểm ảnh (tai, mũi, tóc,...) giờ như input của bài toán linear regression hay logistic regression.
- **Fine tuning:** Sau khi lấy ra các đặc điểm của ảnh bằng việc sử dụng ConvNet của pre-trained model, thì ta sẽ coi đây là input của 1 CNN mới bằng cách thêm các ConvNet và Fully Connected layer. Lý do là ConvNet của VGGFace 2 model có thể lấy ra được các thuộc tính



Hình 11.1: Hiệu quả khi sử dụng transfer learning [2]

của mặt người nói chung nhưng người Việt Nam có những đặc tính khác nên cần thêm 1 số Convnet mới để học thêm các thuộc tính của người Việt Nam.

Bài toán: Ta muốn nhận diện ảnh của 17 loài hoa, mỗi loài hoa có khoảng 80 ảnh.

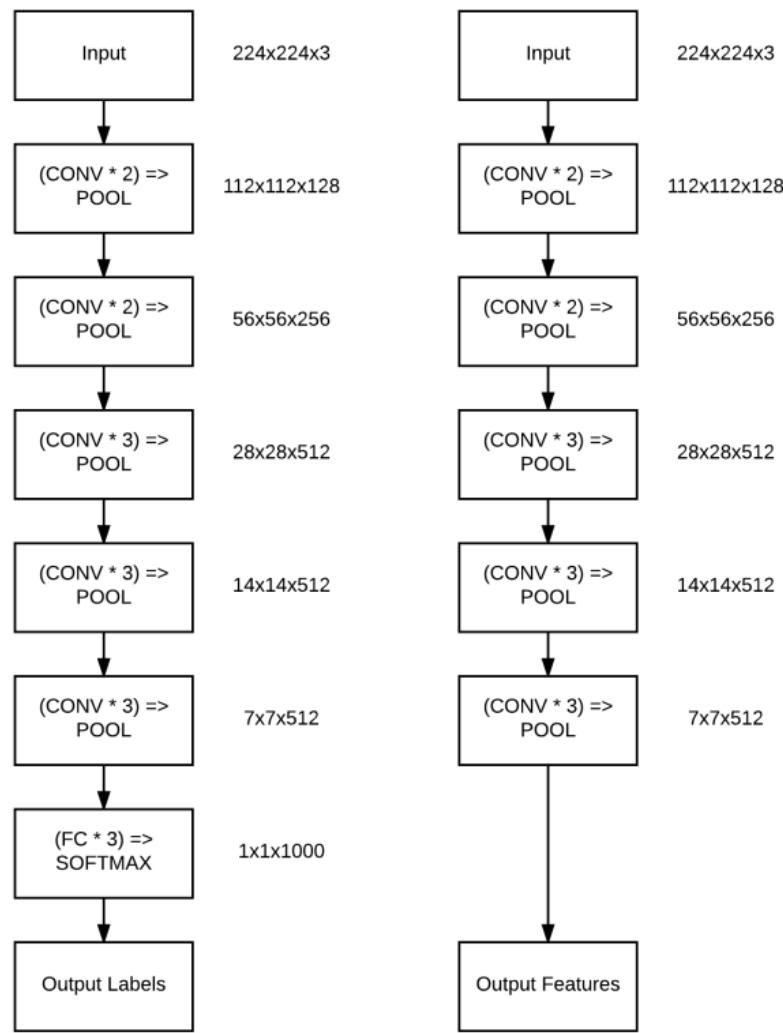
Sử dụng pre-trained model là VGG 16 của ImageNet. Mô hình VGG 16 mọi người có thể xem lại bài CNN. Mô hình VGG16 của ImageNet dataset, phân loại ảnh thuộc 1000 thể loại khác nhau. Nên có thể hiểu là nó đủ tổng quát để tách ra các đặc điểm của bức ảnh, cụ thể ở đây là hoa.

11.1.1 Feature extractor

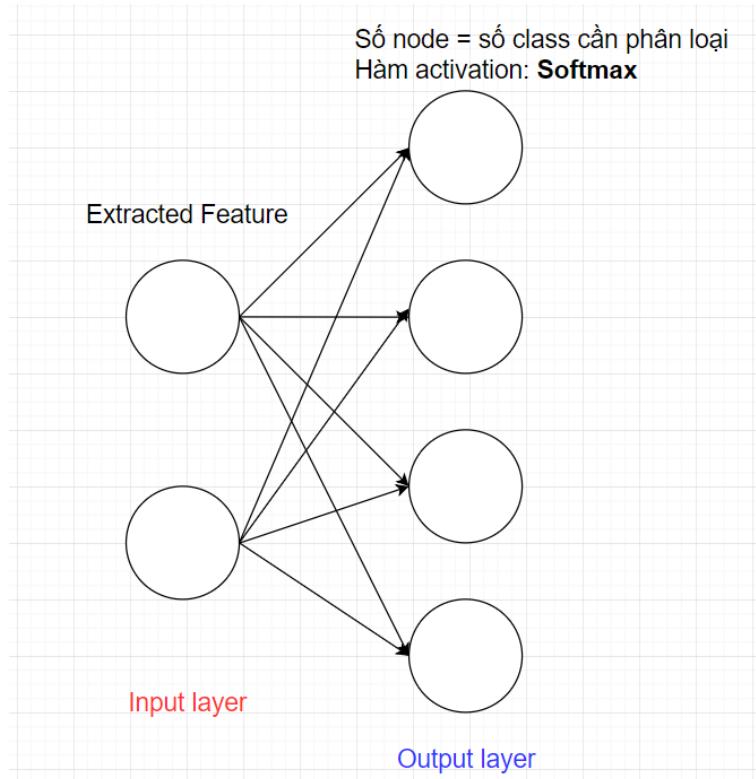
Ta chỉ giữ lại phần ConvNet trong CNN và bỏ đi FCs. Sau đó dùng output của ConvNet còn lại để làm input cho Logistic Regression với nhiều output, như trong hình 11.2.

Mô hình logistic regression với nhiều output có 2 dạng:

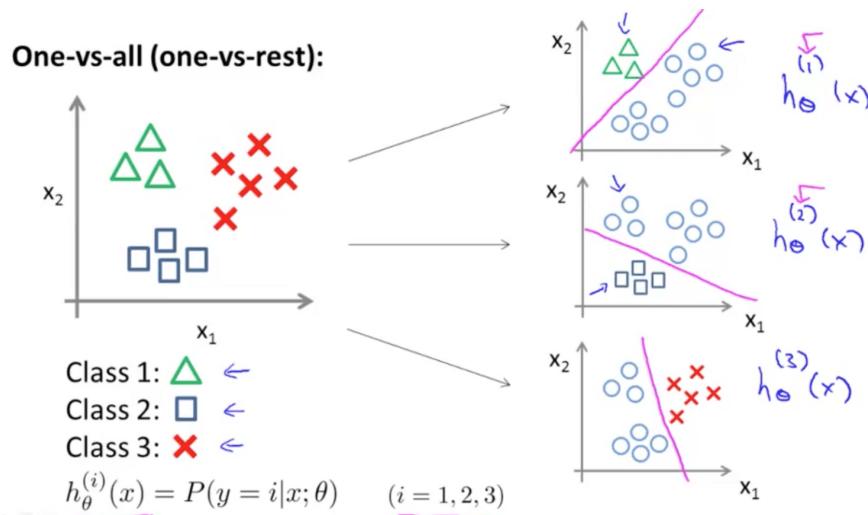
Dạng thứ nhất là một neural network, không có hidden layer, hàm activation ở output layer là softmax function, loss function là hàm categorical-cross entropy, giống như bài phân loại ảnh.



Hình 11.2: Bên trái là mô hình VGG16, bên phải là mô hình VGG16 chỉ bao gồm ConvNet (bỏ Fully Connected layer) [21]



Dạng thứ hai giống như bài logistic regression, tức là model chỉ phân loại 2 class. Mỗi lần ta sẽ phân loại 1 class với tất cả các class còn lại.



Hình 11.3: 1 vs all classification [13]

```
# -*- coding: utf-8 -*-
```

```
# Thêm thư viện
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from imutils import paths
from keras.applications import VGG16
from keras.applications import imagenet_utils
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import numpy as np
import random
import os

# Lấy các đường dẫn đến ảnh.
image_path = list(paths.list_images('dataset/'))

# Đổi vị trí ngẫu nhiên các đường dẫn ảnh
random.shuffle(image_path)

# Đường dẫn ảnh sẽ là dataset/tên_loài_hoa/tên_ảnh ví dụ dataset/Bluebell/image_0241.jpg
labels = [p.split(os.path.sep)[-2] for p in image_path]

# Chuyển tên các loài hoa thành số
le = LabelEncoder()
labels = le.fit_transform(labels)
```

```

# Load model VGG 16 của ImageNet dataset, include_top=False để bỏ phần Fully connected layer
model = VGG16(weights='imagenet', include_top=False)

# Load ảnh và resize về đúng kích thước mà VGG 16 cần là (224,224)
list_image = []
for (j, imagePath) in enumerate(image_path):
    image = load_img(imagePath, target_size=(224, 224))
    image = img_to_array(image)

    image = np.expand_dims(image, 0)
    image = imagenet_utils.preprocess_input(image)

    list_image.append(image)

list_image = np.vstack(list_image)

# Dùng pre-trained model để lấy ra các feature của ảnh
features = model.predict(list_image)

# Giống bước flatten trong CNN, chuyển từ tensor 3 chiều sau ConvNet sang vector 1 chiều
features = features.reshape((features.shape[0], 512*7*7))

# Chia training set, test set tỉ lệ 80-20
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

# Grid search để tìm các parameter tốt nhất cho model. C = 1/lamda, hệ số trong regularizer
# https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
params = {'C' : [0.1, 1.0, 10.0, 100.0]}
#model = GridSearchCV(LogisticRegression(solver='lbfgs', multi_class='multinomial'), params)
model = GridSearchCV(LogisticRegression(), params)
model.fit(X_train, y_train)
print('Best parameter for the model {}'.format(model.best_params_))

# Đánh giá model
preds = model.predict(X_test)
print(classification_report(y_test, preds))

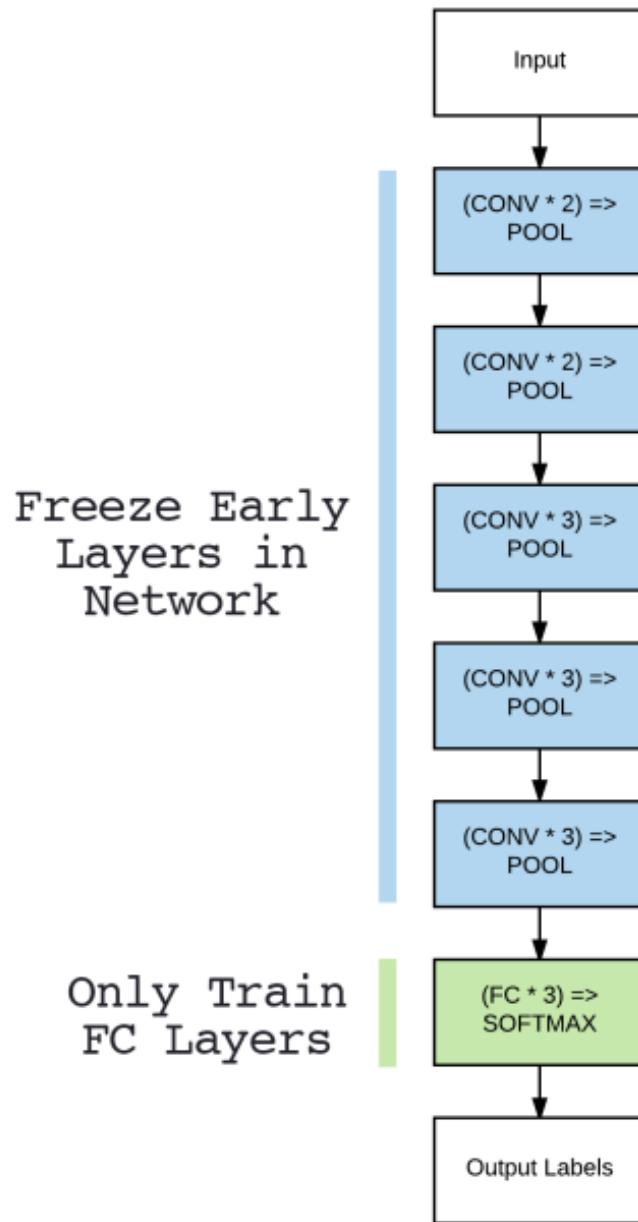
```

11.1.2 Fine tuning

Ta chỉ giữ lại phần ConvNet trong CNN và bỏ đi FCs. Sau đó thêm các Fully Connected layer mới vào output của ConvNet như trong hình 11.4.

Khi train model ta chia làm 2 giai đoạn

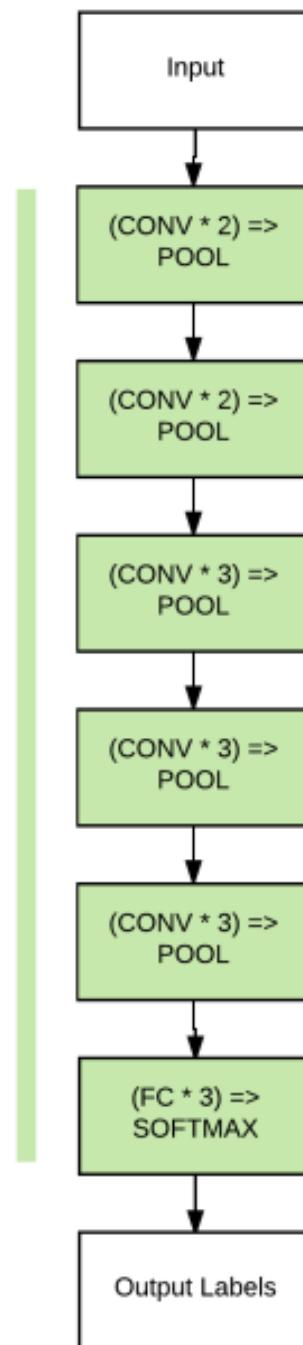
Giai đoạn 1: Vì các Fully Connected layer ta mới thêm vào có các hệ số được khởi tạo ngẫu nhiên tuy nhiên các layer trong ConvNet của pre-trained model đã được train với ImageNet dataset nên ta sẽ không train (đóng băng/freeze) trên các layer trong ConvNet của model VGG16. Sau khoảng 20-30 epoch thì các hệ số ở các layer mới đã được học từ dữ liệu thì ta chuyển sang giai đoạn 2.



Hình 11.5: Freeze các layer của pre-trained model, chỉ train ở các layer mới [21]

Giai đoạn 2: Ta sẽ unfreeze các layer trên ConvNet của pre-trained model và train trên các layer của ConvNet của pre-trained model và các layer mới. Bạn có thể unfreeze tất cả các layer trong ConvNet của VGG16 hoặc chỉ unfreeze một vài layer cuối tùy vào thời gian và GPU bạn có.

Unfreeze Early
Layers & Train
All



Hình 11.6: Freeze các layer của pre-trained model, chỉ train ở các layer mới [21]

```
# -*- coding: utf-8 -*-
```

```
# Thêm thư viện
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from imutils import paths
from keras.applications import VGG16
```

```
from keras.applications import imagenet_utils
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import SGD
from keras.optimizers import RMSprop
from keras.applications import VGG16
from keras.layers import Input
from keras.models import Model
from keras.layers.core import Dense
from keras.layers.core import Dropout
from keras.layers.core import Flatten
import numpy as np
import random
import os

# Lấy các đường dẫn đến ảnh.
image_path = list(paths.list_images('dataset/'))

# Đổi vị trí ngẫu nhiên các đường dẫn ảnh
random.shuffle(image_path)

# Đường dẫn ảnh sẽ là dataset/tên_loài_hoa/tên_ảnh ví dụ dataset/Bluebell/image_0241.jpg
labels = [p.split(os.path.sep)[-2] for p in image_path]

# Chuyển tên các loài hoa thành số
le = LabelEncoder()
labels = le.fit_transform(labels)

# One-hot encoding
lb = LabelBinarizer()
labels = lb.fit_transform(labels)

# Load ảnh và resize về đúng kích thước mà VGG 16 cần là (224,224)
list_image = []
for (j, imagePath) in enumerate(image_path):
    image = load_img(imagePath, target_size=(224, 224))
    image = img_to_array(image)

    image = np.expand_dims(image, 0)
    image = imagenet_utils.preprocess_input(image)

    list_image.append(image)

list_image = np.vstack(list_image)
```

```
# Load model VGG 16 của ImageNet dataset, include_top=False để bỏ phần Fully connected layer
baseModel = VGG16(weights='imagenet', include_top=False, \
                   input_tensor=Input(shape=(224, 224, 3)))

# Xây thêm các layer
# Lấy output của ConvNet trong VGG16
fcHead = baseModel.output

# Flatten trước khi dùng FCs
fcHead = Flatten(name='flatten')(fcHead)

# Thêm FC
fcHead = Dense(256, activation='relu')(fcHead)
fcHead = Dropout(0.5)(fcHead)

# Output layer với softmax activation
fcHead = Dense(17, activation='softmax')(fcHead)

# Xây dựng model bằng việc nối ConvNet của VGG16 và fcHead
model = Model(inputs=baseModel.input, outputs=fcHead)

# Chia training set, test set tỉ lệ 80-20
X_train, X_test, y_train, y_test = train_test_split(list_image, labels, \
                                                    test_size=0.2, random_state=42)

# augmentation cho training data
aug_train = ImageDataGenerator(rescale=1./255, rotation_range=30, \
                               width_shift_range=0.1, height_shift_range=0.1, \
                               shear_range=0.2, zoom_range=0.2, horizontal_flip=True, \
                               fill_mode='nearest')
# augmentation cho test
aug_test = ImageDataGenerator(rescale=1./255)

# freeze VGG model
for layer in baseModel.layers:
    layer.trainable = False

opt = RMSprop(0.001)
model.compile(opt, 'categorical_crossentropy', ['accuracy'])
numOfEpoch = 25
H = model.fit_generator(aug_train.flow(X_train, y_train, batch_size=32),
                        steps_per_epoch=len(X_train)//32,
                        validation_data=(aug_test.flow(X_test, y_test, batch_size=32)),
                        validation_steps=len(X_test)//32,
                        epochs=numOfEpoch)

# unfreeze some last CNN layer:
for layer in baseModel.layers[15:]:
    layer.trainable = True
```

```

numOfEpoch = 35
opt = SGD(0.001)
model.compile(opt, 'categorical_crossentropy', ['accuracy'])
H = model.fit_generator(aug_train.flow(X_train, y_train, batch_size=32),
                        steps_per_epoch=len(X_train)//32,
                        validation_data=(aug_test.flow(X_test, y_test, batch_size=32)),
                        validation_steps=len(X_test)//32,
                        epochs=numOfEpoch)

```

Accuracy của fine-tuning tốt hơn so với feature extractor tuy nhiên thời gian train của fine-tuning cũng lâu hơn rất nhiều. Giải thích đơn giản thì feature extractor chỉ lấy ra đặc điểm chung chung từ pre-trained model của ImageNet dataset cho các loài hoa, nên không được chính xác lắm. Tuy nhiên ở phần fine-tuning ta thêm các layer mới, cũng như train lại 1 số layer ở trong ConvNet của VGG16 nên model giờ học được các thuộc tính, đặc điểm của các loài hoa nên độ chính xác tốt hơn.

11.1.3 Khi nào nên dùng transfer learning

Có 2 yếu tố quan trọng nhất để dùng transfer learning đó là kích thước của dữ liệu bạn có và sự tương đồng của dữ liệu giữa mô hình bạn cần train và pre-trained model.

- Dữ liệu bạn có nhỏ và tương tự với dữ liệu ở pre-trained model. Vì dữ liệu nhỏ nên nếu dùng fine-tuning thì model sẽ bị overfitting. Hơn nữa là dữ liệu tương tự nhau nên là ConvNet của pre-trained model cũng lấy ra các đặc điểm ở dữ liệu của chúng ta. Do đó nên dùng feature extractor.
- Dữ liệu bạn có lớn và tương tự với dữ liệu ở pre-trained model. Giờ có nhiều dữ liệu ta không sợ overfitting do đó nên dùng fine-tuning.
- Dữ liệu bạn có nhỏ nhưng khác với dữ liệu ở pre-trained model. Vì dữ liệu nhỏ nên ta nên dùng feature extractor để tránh overfitting. Tuy nhiên do dữ liệu ta có và dữ liệu ở pre-trained model khác nhau, nên không nên dùng feature extractor với toàn bộ ConvNet của pre-trained model mà chỉ dùng các layer đầu. Lý do là vì các layer ở phía trước sẽ học các đặc điểm chung chung hơn (cạnh, góc,...), còn các layer phía sau trong ConvNet sẽ học các đặc điểm cụ thể hơn trong dataset (ví dụ mắt, mũi,...).
- Dữ liệu bạn có lớn và khác với dữ liệu ở pre-trained model. Ta có thể train model từ đầu, tuy nhiên sẽ tốt hơn nếu ta khởi tạo các giá trị weight của model với giá trị của pre-trained model và sau đó train bình thường.

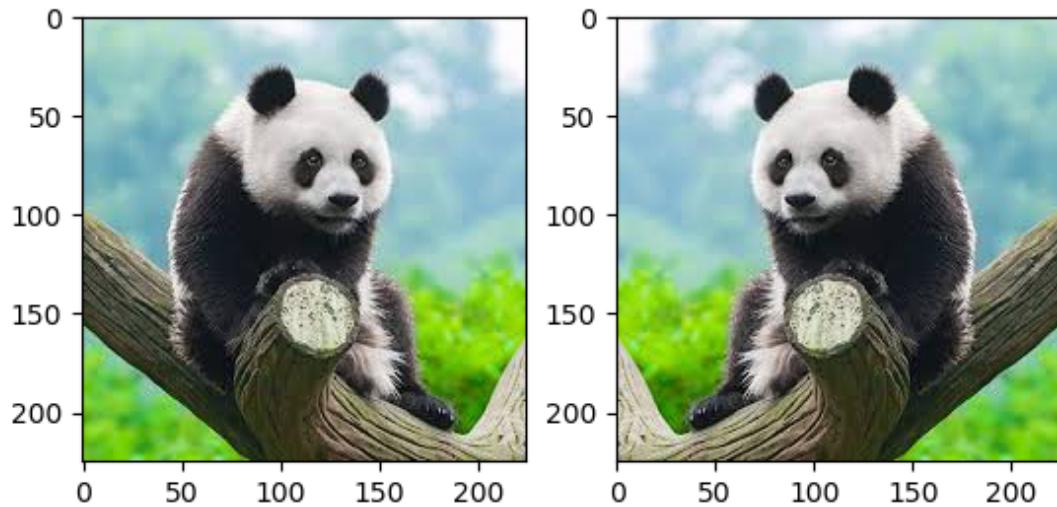
Lưu ý

- Vì pre-trained model đã được train với kích thước ảnh cố định, nên khi dùng pre-trained model ta cần resize lại ảnh có kích thước bằng kích thước mà ConvNet của pre-trained model yêu cầu.
- Hệ số learning rate của ConvNet của pre-trained model nên được đặt với giá trị nhỏ vì nó đã được học ở pre-trained model nên ít cần cập nhật hơn so với các layer mới thêm.

11.2 Data augmentation

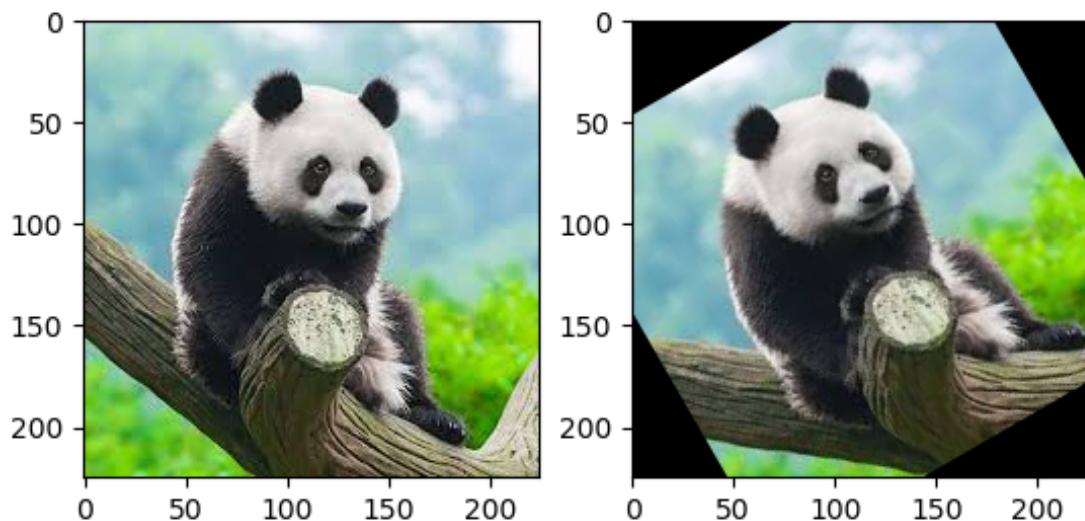
Ngoài transfer learning, có một kỹ thuật nữa giải quyết vấn đề có ít dữ liệu cho việc training model, đó là data augmentation. Augmentation là kỹ thuật tạo ra dữ liệu training từ dữ liệu mà ta đang có. Cùng xem một số kỹ thuật augmentation phổ biến với ảnh nhé.

Flip: Lật ngược ảnh theo chiều dọc hoặc chiều ngang



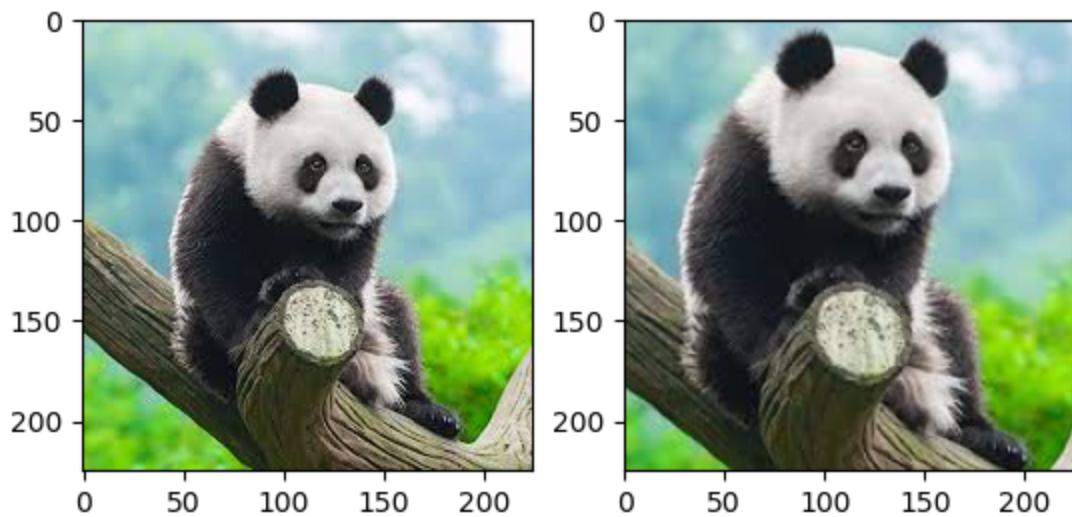
Hình 11.7: lật ngược ảnh theo chiều dọc

Rotation: Quay ảnh theo nhiều góc khác nhau



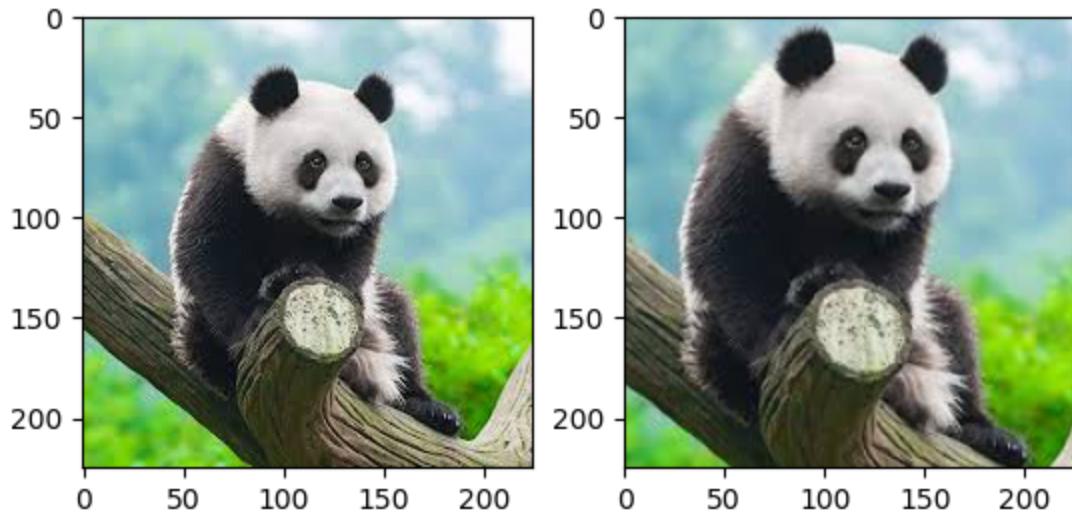
Hình 11.8: Rotate ảnh 30 độ

Scale: Phóng to hoặc thu nhỏ ảnh



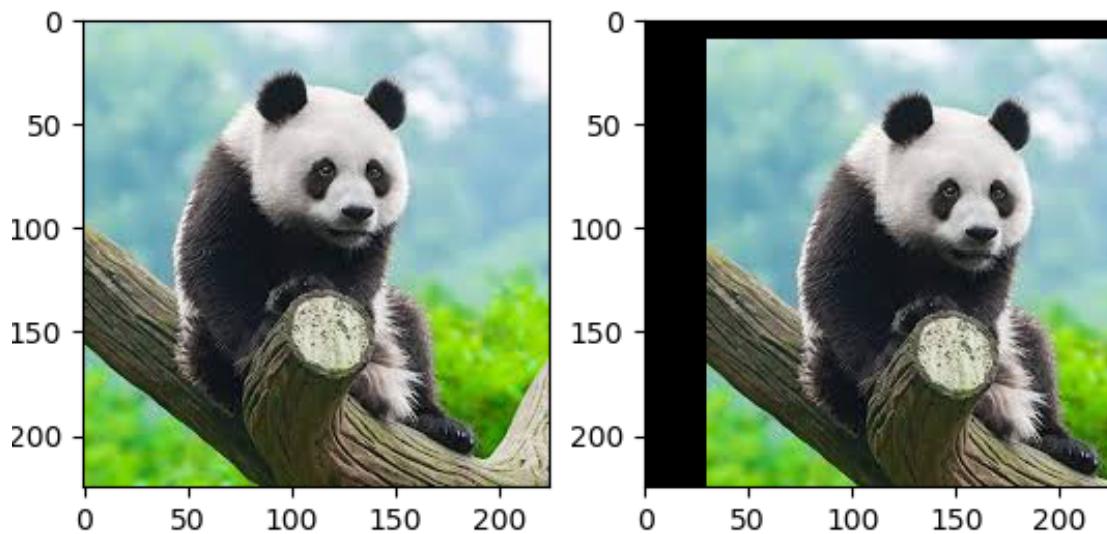
Hình 11.9: Scale ảnh

Crop: Cắt một vùng ảnh sau đó resize vùng ảnh đấy về kích thước ảnh ban đầu



Hình 11.10: Crop ảnh

Translation: dịch chuyển ảnh theo chiều x, y.

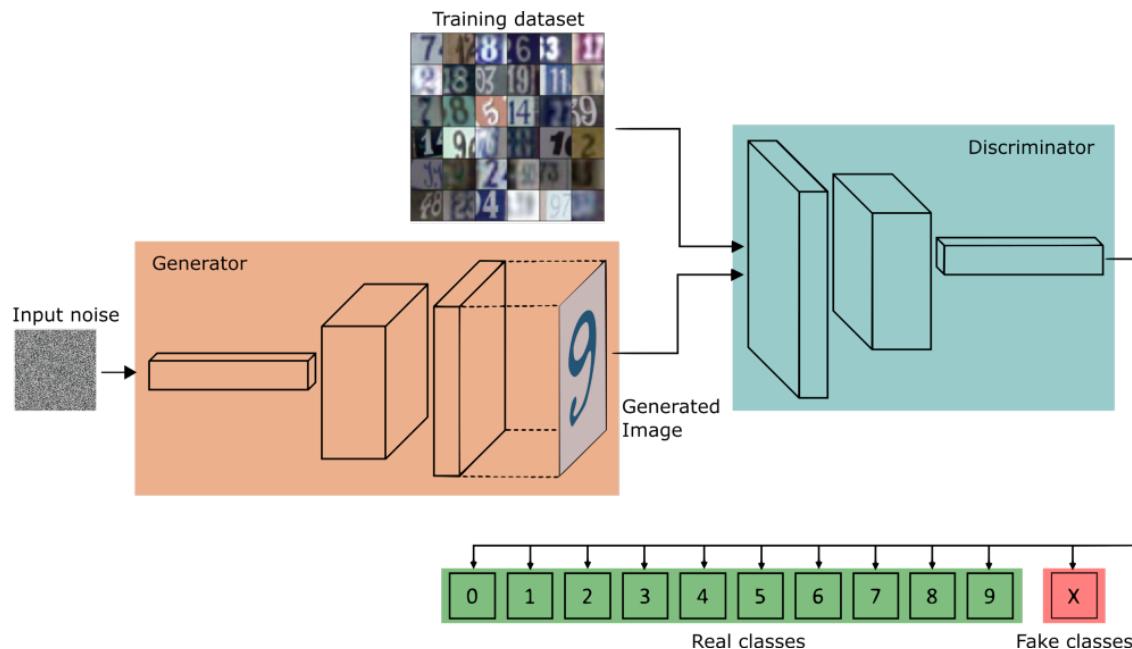


Hình 11.11: Translate ảnh 30px theo phương x, 10px theo phương y

Tuy nhiên khi rotate hoặc translation thì ảnh bị những khoảng đen mà thường ảnh thực tế không có các khoảng đen đây nên có một số cách để xử lý như: lấy giá trị từ cạnh của ảnh mới để cho các pixel bị đen, gán các giá trị đen bằng giá trị của ảnh đối xứng qua cạnh,...

Nên áp dụng kiểu augmentation nào thì tùy thuộc vào ngữ nghĩa của ảnh trong bài toán bạn đang giải quyết.

*** Ngoài ra người ta còn dùng GAN cho semi-supervised learning



Hình 11.12: Ứng dụng GAN cho semi-supervised learning [24]

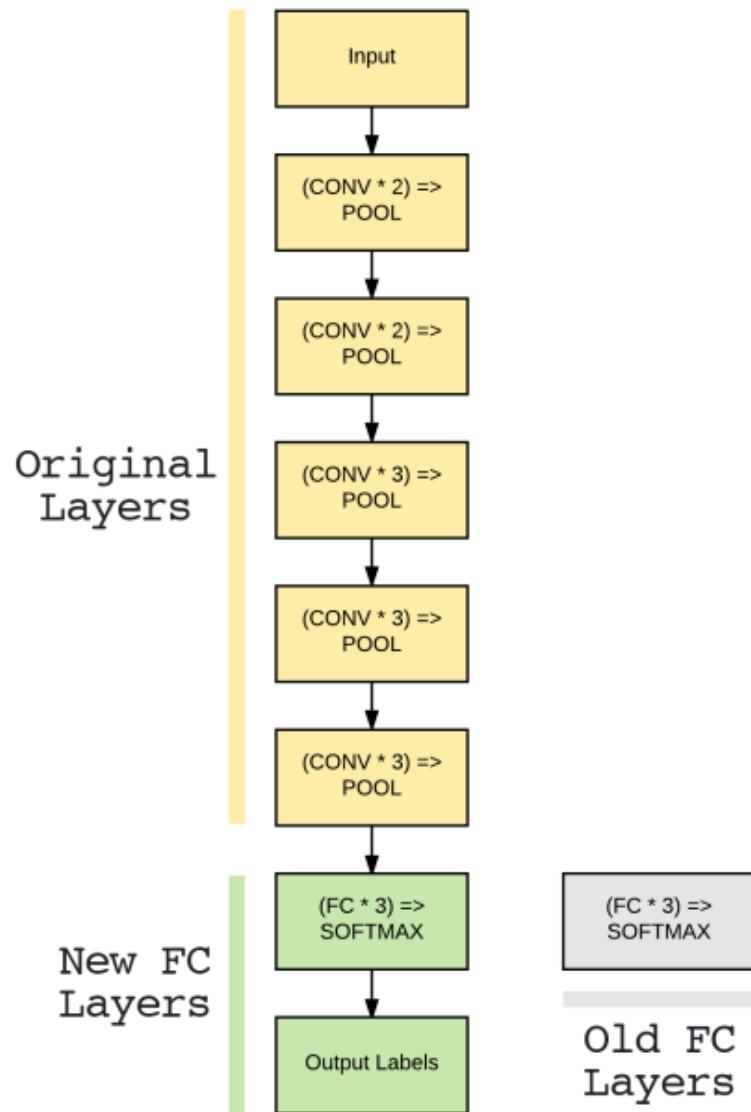
Ý tưởng là generator giống như một nguồn cung cấp ảnh mới, discriminator giờ có 2 việc:

- nhận diện ảnh fake và ảnh thật
- phân loại ảnh

Tài liệu cho ai muốn đọc thêm ở [đây](#).

11.3 Bài tập

1. Có mấy kiểu transer learning? Khi nào nên dùng kiểu transfer learning nào?
2. Data augmentation là gì? Tác dụng của augmentation?
3. Áp dụng kĩ thuật transfer learning và augmentation cho các bài toán phân loại ảnh trong những bài trước xem hiệu quả có tốt hơn không?



Hình 11.4: Ta bỏ các Fully Connected layer ở model VGG16 đi và thêm vào các Fully Connected layer mới [21]

12. Các kỹ thuật cơ bản trong deep learning

Bài này tôi tổng hợp một số kĩ thuật để train model tốt hơn, cũng như một số khái niệm bạn sẽ gặp khi làm việc với deep learning. Đầu tiên tôi nói về tầm quan trọng của vectorization, tiếp tôi sẽ nói về kĩ thuật mini-batch gradient descent với lượng dữ liệu lớn. Sau đó tôi nói về bias, variance, cách đánh giá bias-variance trong model và một số biện pháp để giải quyết vấn đề high bias, high variance. Tôi có giải thích về kĩ thuật dropout để tránh overfitting. Phần cuối tôi nói về vanishing và exploding gradient, cũng như giới thiệu một số hàm activation function phổ biến và cách chọn hàm activation.

12.1 Vectorization

Ngay từ bài đầu tiên về linear regression, tôi đã giới thiệu về việc biểu diễn và tính toán dưới dạng vector. Việc biểu diễn bài toán dưới dạng vector như vậy gọi là vectorization. Nó không chỉ giúp code gọn lại mà còn tăng tốc độ tính toán một cách đáng kể khi thực hiện các phép tính trên vector, ma trận so với for-loop.

```
# -*- coding: utf-8 -*-

import numpy as np
import time

a = np.random.rand(10000000)
b = np.random.rand(10000000)

start = time.time()
c = np.dot(a, b.T)
end = time.time()

print("Vectorization time : ", end-start)
```

```

start = time.time()
c = 0
for i in range(len(a)):
    c += a[i] * b[i]
end = time.time()

print("For-loop time : ", end-start)

```

Bạn sẽ thấy nếu dùng for-loop mất hơn 4s để nhân 2 vector trong khi dùng thư viện numpy để tính chỉ mất 0.01s. Tại sao lại như thế nhỉ?

Giả sử bạn có 10 tấn hàng cần vận chuyển từ A đến B và bạn có một xe tải với khả năng chở được 5 tấn mỗi lần. Vậy nếu chất đầy hàng mỗi lần chở thì chỉ cần 2 lần chạy là chuyển hết số hàng. Tuy nhiên nếu mỗi lần bạn chỉ chất 1 tấn hàng lên xe để chở đi thì xe cần đi tới 10 lần để chuyển hết số hàng.

Tương tự như vậy, khi nhân 2 vector ở trên bạn cần thực hiện 10000000 phép tính nhân 2 số. Giả sử máy tính có thể tính được tối đa 1000 phép tính nhân một lúc. Việc bạn dùng for-loop giống như mỗi thời điểm bạn chỉ yêu cầu máy tính thực hiện một phép nhân, nên để nhân 2 vector sẽ cần 10000000 đơn vị thời gian. Tuy nhiên thư viện numpy sẽ tối ưu việc tính toán bằng cách yêu cầu máy tính thực hiện 1000 phép tính một lúc, tức là chỉ cần $\frac{10000000}{1000} = 10000$ đơn vị thời gian để hoàn thành phép nhân 2 vector. Vậy nên là việc vectorization thông thường sẽ tính toán nhanh hơn.

12.2 Mini-batch gradient descent

12.2.1 Mini-batch gradient descent là gì

Ở trong thuật toán gradient descent, tại bước thứ hai khi ta tính đạo hàm của loss function với các biến. Trong bài linear regression, ta dùng tất cả các dữ liệu trong dataset để tính đạo hàm rồi cập nhật bước 2:

$$\begin{aligned}
 X &= \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots & \dots \\ 1 & x_n \end{bmatrix}, Y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} \\
 \frac{dJ}{dw_0} &= \text{sum}(\hat{Y} - Y), \frac{dJ}{dw_1} = \text{sum}(X[:, 1] \otimes (\hat{Y} - Y)) \\
 w_0 &= w_0 - \text{learning_rate} * \frac{dJ}{dw_0}, w_1 = w_1 - \text{learning_rate} * \frac{dJ}{dw_1}
 \end{aligned}$$

Thuật toán gradient descent chạy tốt nhưng số lượng dữ liệu trong training set chỉ là 30. Tuy nhiên nếu dữ liệu có kích thước lớn như ảnh và số lượng lớn hơn ví dụ 5000 thì việc tính đạo hàm với loss function với toàn bộ dữ liệu sẽ rất tốn thời gian. Vì mini-batch gradient descent ra đời để giải quyết vấn đề này.

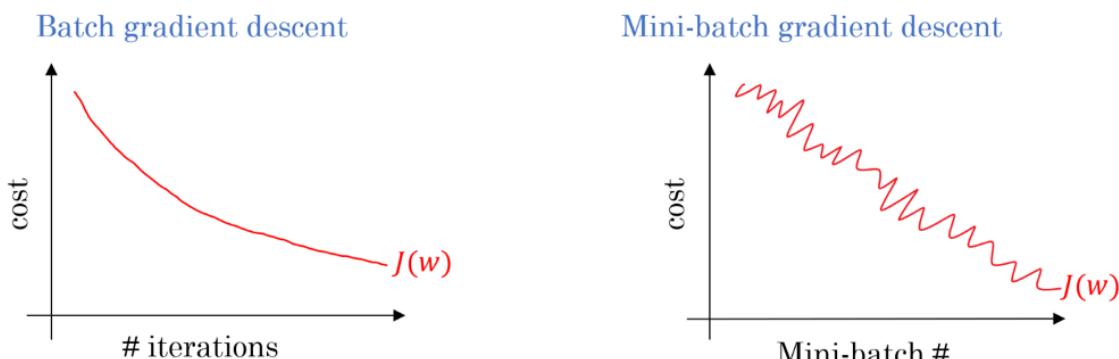
Dựa vào số lượng dữ liệu cho mỗi lần thực hiện bước 2 trong gradient descent là người ta chia ra làm 3 loại:

- Batch gradient descent: Dùng tất cả dữ liệu trong training set cho mỗi lần thực hiện bước tính đạo hàm.

- Mini-batch gradient descent: Dùng một phần dữ liệu trong training set cho mỗi lần thực hiện bước tính đạo hàm.
- Stochastic gradient descent: Chỉ dùng một dữ liệu trong training set cho mỗi lần thực hiện bước tính đạo hàm.

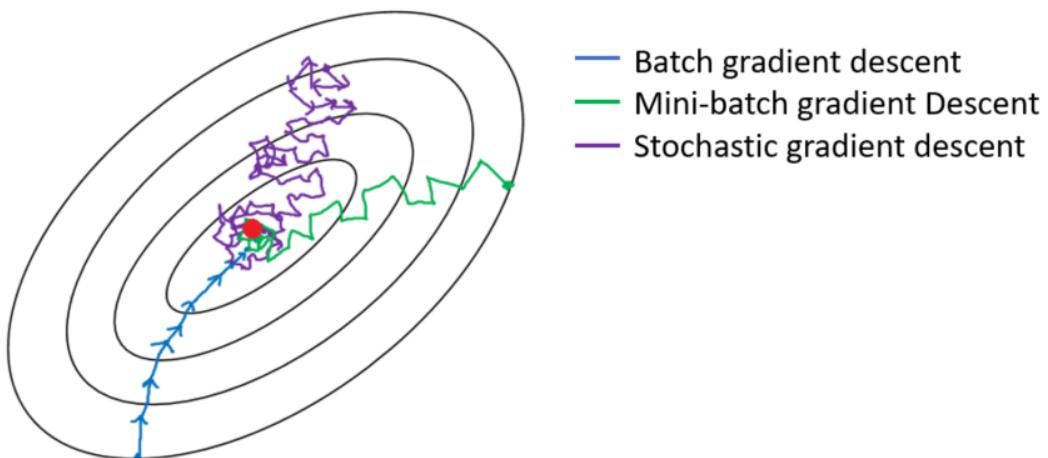
Ví dụ điểm thi đại học trung bình của một trường trung học phổ thông là 24 điểm. Batch gradient descent giống như tính điểm trung bình tất cả học sinh thi đại học năm nay của trường, con số sẽ tương đối gần 24, ví dụ 24,5. Mini-batch sẽ chọn ngẫu nhiên một số học sinh, ví dụ 32 học sinh để tính điểm trung bình thì điểm trung bình sẽ xa 24 hơn ví dụ 22. Tuy nhiên, Stochastic giống như chỉ chọn một học sinh làm điểm trung bình, thì lúc này điểm trung bình có thể khác xa con số 24 rất nhiều, ví dụ 18 hoặc 29. Việc điểm trung bình ở mini-batch hay stochastic khác so với điểm trung bình toàn trường gọi là nhiễu trong dữ liệu. Hay giải thích đơn giản là chỉ lấy 1 hoặc một phần dữ liệu thì không thể mô tả hết được tất cả dữ liệu.

Do đó hàm loss-function với hệ số learning_rate phù hợp thì batch gradient descent theo epoch sẽ giảm đều đặn. Vì có nhiều trong dữ liệu nên mini-batch thì vẫn giảm nhưng có dao động và stochastic có giảm nhưng dao động cực kì lớn.



Hình 12.1: So sánh loss function khi dùng batch và mini-batch [13]

Hình dưới là biểu diễn biệc cập nhật hệ số trong gradient descent, điểm đỏ là giá trị nhỏ nhất ta cần tìm, các điểm ở ngoài cùng là giá trị khởi tạo của hệ số trong gradient descent. Ta có thể thấy vì không có nhiễu nên batch gradient descent thì hệ số cập nhật trực tiếp theo 1 đường thẳng. Mini-batch thì mất nhiều thời gian hơn và còn đi chệch hướng tuy nhiên thì vẫn đến được điểm đỏ. Còn stochastic thì đi khá lòng vòng để đến được điểm đỏ và vì dữ liệu quá nhiều nên có thể thuật toán gradient descent chỉ quanh điểm đỏ mà không đến được điểm đỏ (minimum point).



Hình 12.2: Cập nhật loss function đến minimum point của các thuật toán [13]

Batch gradient descent thường được dùng khi số lượng dữ liệu trong training set nhỏ hơn 2000. Với lượng dữ liệu lớn thì mini-batch gradient descent được sử dụng. Nó có thể giải quyết được vấn đề lượng dữ liệu quá lớn như trong batch gradient descent, hơn nữa đỡ nhiều và có thể dùng vectorization so với stochastic gradient descent nên thường được sử dụng trong deep learning.

12.2.2 Các thông số trong mini-batch gradient descent

Ví dụ code trong bài 7, trong bài toán phân loại chữ số cho dữ liệu MNIST

```
# 7. Thực hiện train model với data
H = model.fit(X_train, Y_train, validation_data=(X_val, Y_val),
               batch_size=32, epochs=10, verbose=1)
```

X_train, Y_train là dữ liệu và label cho training set. Tương tự X_val, Y_val là dữ liệu cho validation set.

batch_size: Là size trong mini-batch gradient descent, nghĩa là dùng bao nhiêu dữ liệu cho mỗi lần tính và cập nhật hệ số.

steps_per_epoch: Là bao nhiêu lần thực hiện bước 2 trong gradient descent trong mỗi epoch. Mặc định sẽ là số lượng dữ liệu chia cho batch_size. Hiểu đơn giản là mỗi epoch sẽ dùng hết các dữ liệu để tính gradient descent.

epochs: số lượng epoch thực hiện trong quá trình training.

Vậy thực sự số lần thực hiện bước 2 trong gradient descent trong quá trình training là: epochs * steps_per_epoch.

Lời khuyên:

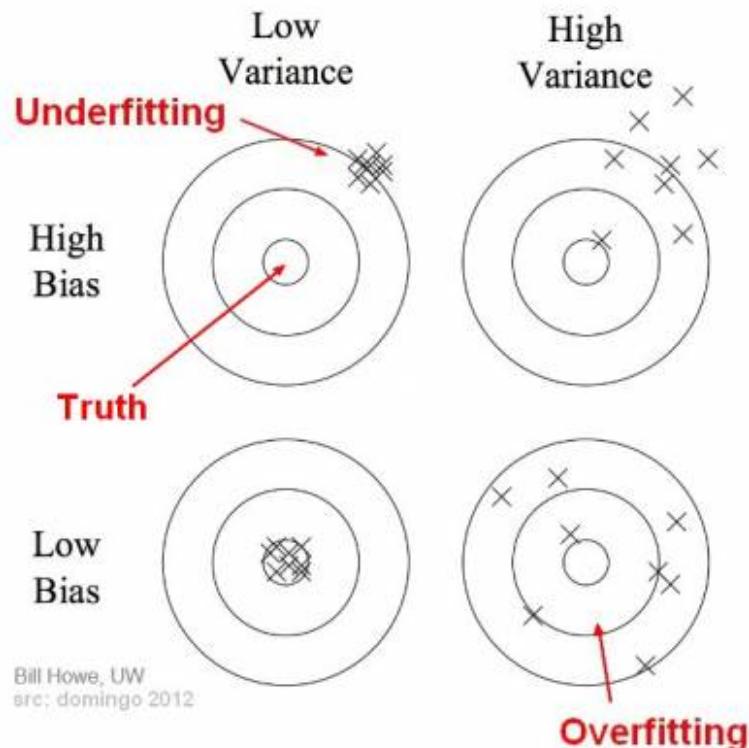
- Batch_size nên được chọn là số mũ của 2 ví dụ 16, 32, 64, 128 để CPU/GPU tính toán tốt hơn. Giá trị mặc định là 32.
- Nên vẽ đồ thị loss/epoch để chọn batch_size phù hợp.

12.3 Bias và variance

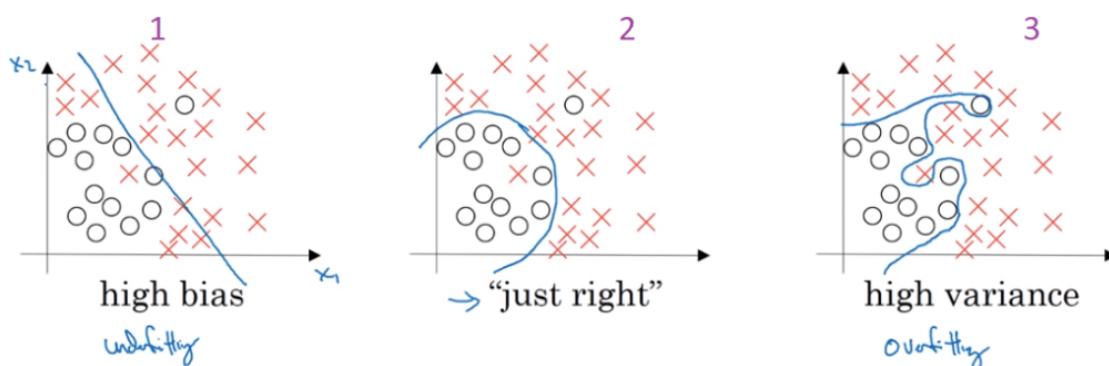
12.3.1 Bias, variance là gì

Bias: nghĩa là độ lệch, biểu thị sự chênh lệch giữa giá trị trung bình mà mô hình dự đoán và giá trị thực tế của dữ liệu.

Variance: nghĩa là phương sai, biểu thị độ phân tán của các giá trị mà mô hình dự đoán so với giá trị thực tế.



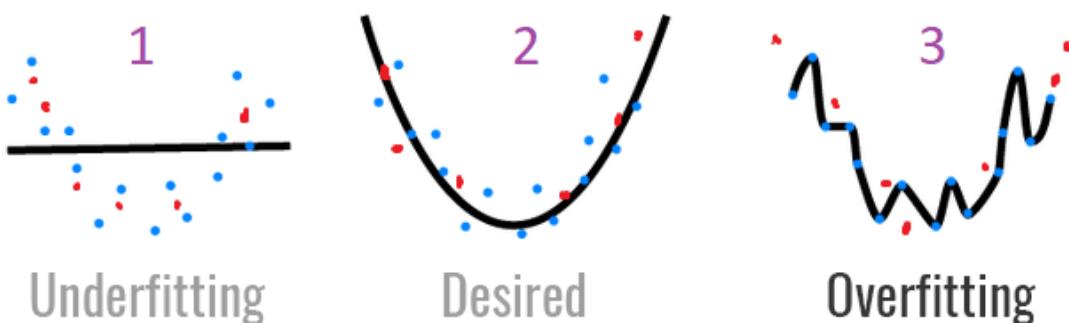
Giá trị thật dữ liệu (ground truth) ở giữa tâm các đường tròn. Các dấu X là các giá trị dự đoán. Ta thấy nếu high bias thì giá trị dự đoán rất xa tâm. Tuy nhiên nếu high variance thì các giá trị dự đoán phân tán rộng dẫn đến việc ra giá trị thực tế. => Ta mong muốn low bias và low variance.



Đây là bài toán logistic regression, cần tìm đường phân chia dữ liệu.

- Ở hình 1, đường phân chia có khá nhiều điểm bị lỗi => sự chênh lệch giữa mô hình dự đoán và giá trị thực tế của dữ liệu cao => **high bias**, hay còn được gọi là **underfitting**, ý hiểu là mô hình hiện tại đơn giản hơn và chưa mô tả được mô hình của dữ liệu thực tế.
- Ở hình 2, đường phân chia vẫn có lỗi nhưng ở mức chấp nhận được và nó có thể mô tả dữ liệu => **low bias, low variance**.
- Ở hình 3, đường phân chia có thể dự đoán đúng tất cả các điểm trong training set nhưng vì nó không tổng quát hóa mô hình dữ liệu thực sự nên khi áp dụng dự đoán vào validation set thì sẽ có rất nhiều lỗi => **high variance** hay còn được gọi là **overfitting**, ý hiểu là mô hình hiện tại thực hiện tốt với dữ liệu trong training set nhưng dự đoán không tốt với validation set.

Thực ra khái niệm high bias và high variance khá trìu tượng và nhiều lúc dùng nhầm lẫn giữa thống kê và machine learning. Nên khái niệm hay được dùng hơn là underfitting và overfitting.



Hình 12.3: Các điểm màu xanh là training set, điểm màu đỏ là validation set

Ví dụ khi luyện thi đại học, nếu bạn chỉ luyện khoảng 1-2 đề trước khi thi thì bạn sẽ bị underfitting vì bạn chưa hiểu hết cấu trúc, nội dung của đề thi. Tuy nhiên nếu bạn chỉ luyện kĩ 50 đề thầy cô giáo bạn soạn và đưa cho thì khả năng bạn sẽ bị overfitting với các đề mà thầy cô giáo các bạn soạn mà khi thi đại học có thể điểm số của các bạn vẫn tệ.

12.3.2 Bias, variance tradeoff

Nếu model quá đơn giản thì ta sẽ bị high bias và low variance. Tuy nhiên nếu model quá phức tạp thì sẽ bị high variance và low bias. Đây là bias, variance tradeoff. Do đó để train được model tốt ta cần cân bằng giữa bias và variance.

12.3.3 Đánh giá bias and variance

Có 2 thông số thường được sử dụng để đánh giá bias and variance của mô hình là training set error và validation set error. Ví dụ error (1-accuracy) trong logistic regression.

Train set error	1%	15%	15%	0.5%
Val set error	11%	16%	30%	1%
	High variance	High bias	High bias High variance	Low bias Low variance

Ta mong muốn model là low bias và low variance. Cùng xem một số cách để giải quyết vấn đề high bias hoặc high variance nhé.

Giải quyết high bias (underfitting): Ta cần tăng độ phức tạp của model

- Tăng số lượng hidden layer và số node trong mỗi hidden layer.
- Dùng nhiều epoch hơn để train model.

Giải quyết high variance (overfitting):

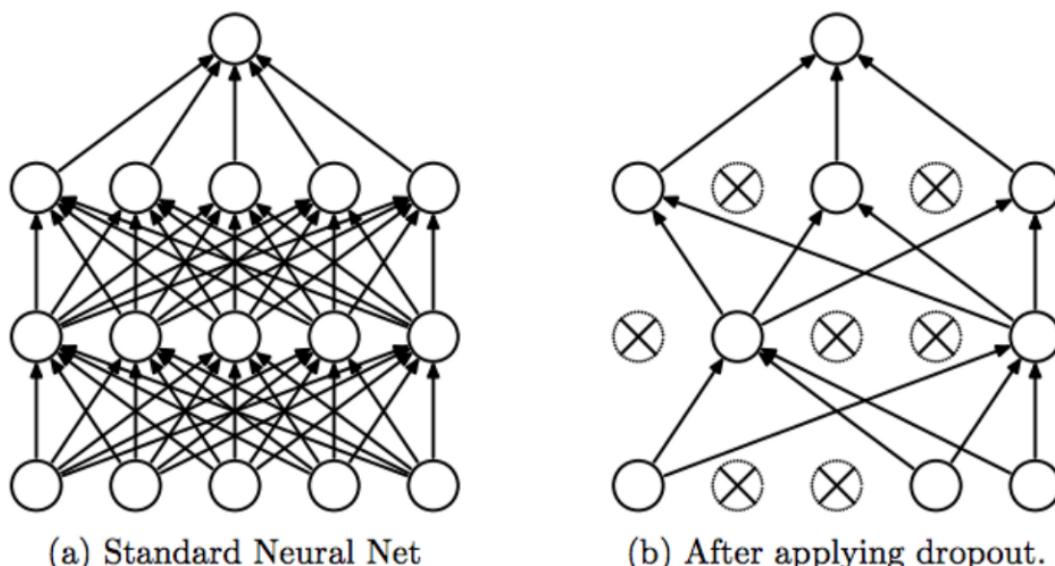
- Thu thập thêm dữ liệu hoặc dùng data augmentation

- Dùng regularization như: L1, L2, dropout

12.4 Dropout

12.4.1 Dropout là gì

Dropout với hệ số p nghĩa là trong quá trình train model, với mỗi lần thực hiện cập nhật hệ số trong gradient descent ta ngẫu nhiên loại bỏ $p\%$ số lượng node trong layer đấy, hay nói cách khác là giữ lại $(1-p\%)$ node. Mỗi layer có thể có các hệ số dropout p khác nhau.



Hình 12.4: So sánh model dropout và neural network thông thường [26]

Ví dụ mô hình neural network 1-2-1: 1 input layer, 2 hidden layer và 1 output layer. Ví dụ như hidden layer 1, ta dùng dropout với $p = 0.6$, nên chỉ giữ lại 2 trên 5 node cho mỗi lần cập nhật.

12.4.2 Dropout hạn chế việc overfitting

Overfitting là mô hình đang dùng quá phức tạp so với mô hình thật của dữ liệu. Khi ta dùng dropout như hình trên thì rõ ràng mô hình bên phải đơn giản hơn => tránh overfitting.

Thêm vào đó, vì mỗi bước khi train model thì ngẫu nhiên $(1-p\%)$ các node bị loại bỏ nên model không thể phụ thuộc vào bất kì node nào của layer trước mà thay vào đó có xu hướng trải đều weight, giống như trong L2 regularization => tránh được overfitting.

12.4.3 Lời khuyên khi dùng dropout

- Hệ số p nên ở khoảng $[0.2, 0.5]$. Nếu p quá nhỏ thì không có tác dụng chống overfitting, tuy nhiên nếu p quá lớn thì gần như loại bỏ layer đấy và có thể dẫn đến underfitting.
- Nên dùng model lớn, phức tạp hơn vì ta có dropout chống overfitting.
- Dropout chỉ nên dùng cho Fully Connected layer, ít khi được dùng cho ConvNet layer
- Hệ số p ở các layer nên tỉ lệ với số lượng node trong FC layer đó.

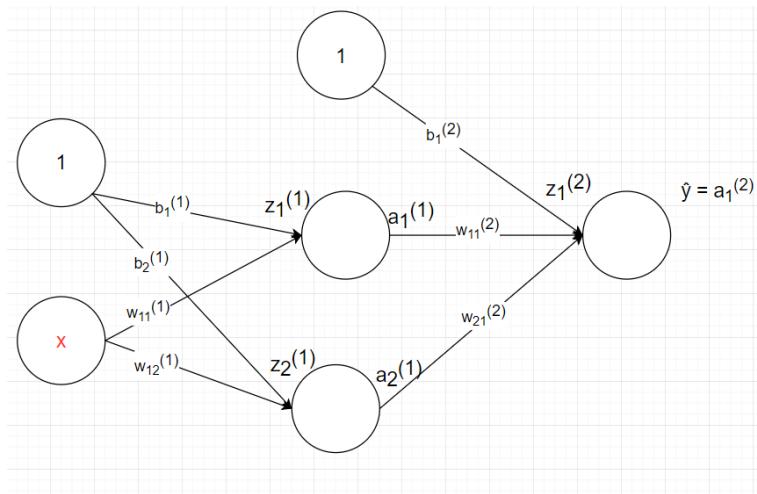
12.5 Activation function

12.5.1 Non-linear activation function

Hàm activation function được dùng sau bước tính tổng linear trong neural network hoặc sau convolutional layer trong CNN. Và hàm activation là non-linear function.

Linear function là gì? Theo wiki, "a linear function from the real numbers to the real numbers is a function whose graph is a line in the plane", tóm lại linear function là một đường thẳng dạng $y = a*x + b$. Vậy sẽ ra sao nếu hàm activation trong neural network là một linear function?

Giả sử hàm activation dạng $y = f(x) = 2*x + 3$ và neural network như sau:



Hình 12.5: Mô hình neural network, 1-2-1.

$$z_1^{(1)} = b_1^{(1)} + x * w_{11}^{(1)} \Rightarrow a_1^{(1)} = f(z_1^{(1)}) = 2 * z_1^{(1)} + 3 = 2 * (b_1^{(1)} + x * w_{11}^{(1)}) + 3$$

Tương tự

$$a_2^{(1)} = 2 * (b_2^{(1)} + x * w_{12}^{(1)}) + 3$$

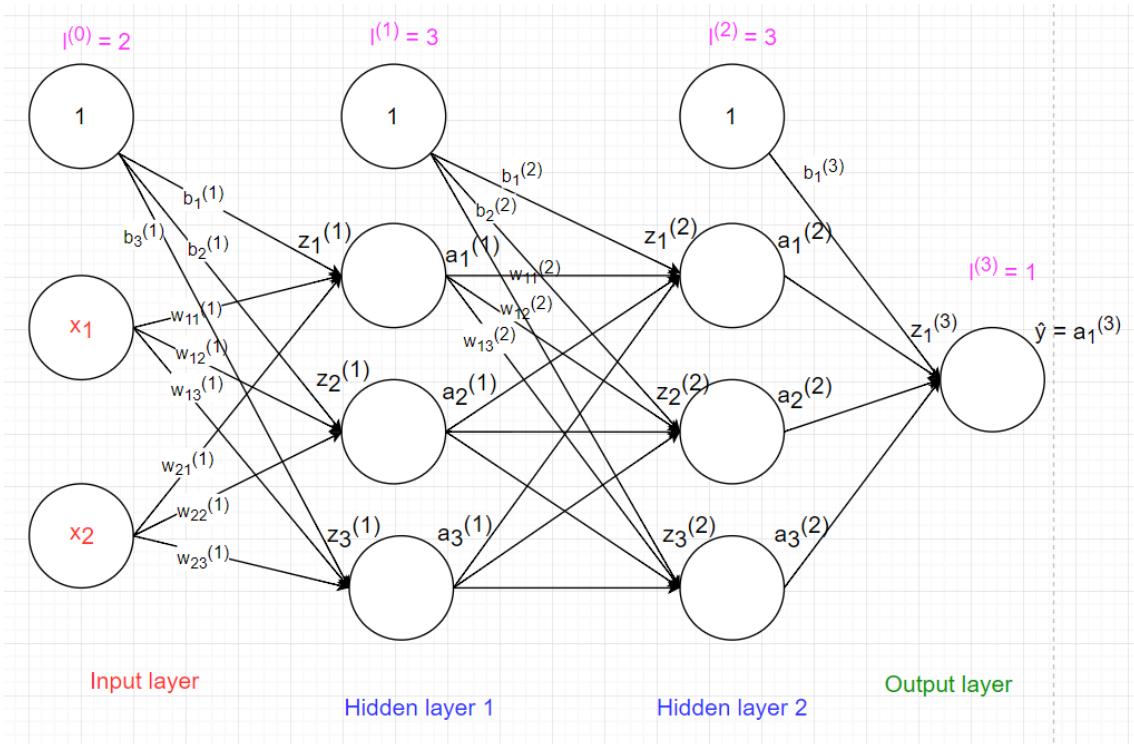
Do đó

$$\begin{aligned} \hat{y} = a_1^{(2)} &= f(z_1^{(2)}) = 2 * z_1^{(2)} + 3 = 2 * (b_1^{(2)} + a_1^{(1)} * w_{11}^{(2)} + a_2^{(1)} * w_{21}^{(2)}) + 3 = 2 * (b_1^{(2)} + (2 * (b_1^{(1)} + x * w_{11}^{(1)}) + 3) * w_{11}^{(2)} + (2 * (b_2^{(1)} + x * w_{12}^{(1)}) + 3) * w_{21}^{(2)}) + 3 = x * (4 * w_{11}^{(1)} * w_{11}^{(2)} + 4 * w_{12}^{(1)} * w_{21}^{(2)}) + (2 * (b_1^{(2)} + (2 * b_1^{(1)} + 3) * w_{11}^{(2)} + (2 * b_2^{(1)} + 3) * w_{21}^{(2)}) + 3) \end{aligned}$$

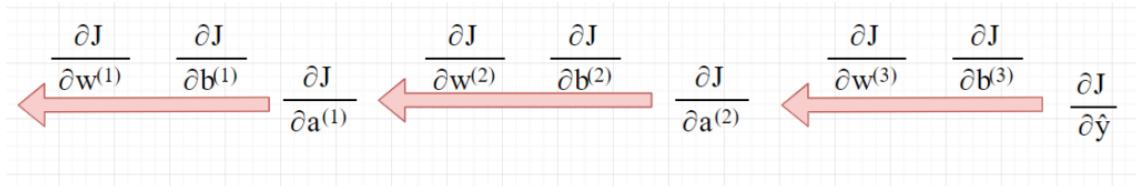
Tóm lại $\hat{y} = x * a + b$ hay nói cách khác mô hình neural network chỉ là mô hình linear regression đơn giản => **Hàm activation function phải là non-linear function**.

12.5.2 Vanishing và exploding gradient

Backpropagation là thuật toán được dùng để tính đạo hàm các hệ số trong neural network với loss function để rồi áp dụng gradient descent để tìm các hệ số.



Hình 12.6: Mô hình neural network 2-3-3-1



Hình 12.7: Quá trình backpropagation

Ta có:

$$\frac{\partial J}{\partial A(\hat{2})} = \left(\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial \hat{Y}}{\partial Z^{(3)}} \right) * (W^{(3)})^T \quad \text{và} \quad \frac{\partial J}{\partial A(\hat{1})} = \left(\frac{\partial J}{\partial A^{(2)}} \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}} \right) * (W^{(2)})^T$$

$$\text{Do đó } \frac{\partial J}{\partial A(\hat{1})} = \left(\left(\frac{\partial J}{\partial \hat{Y}} \otimes \frac{\partial \hat{Y}}{\partial Z^{(3)}} \right) * (W^{(3)})^T \right) \otimes \frac{\partial A^{(2)}}{\partial Z^{(2)}} * (W^{(2)})^T$$

$$\text{Ta tạm kí hiệu đạo hàm của biến qua hàm activation } \frac{\partial A^{(i)}}{\partial Z^{(i)}} = D^{(i)}$$

$$\text{Có thể tạm hiểu là: } \frac{\partial J}{\partial A(\hat{1})} = \frac{\partial J}{\partial \hat{Y}} * D^{(3)} * D^{(2)} * W^{(3)} * W^{(2)}$$

$$\text{Nếu neural network có } n \text{ layer thì } \frac{\partial J}{\partial A(\hat{l})} = \frac{\partial J}{\partial \hat{Y}} * \prod_{i=l+1}^n D^{(i)} * \prod_{i=l+1}^n W^{(i)}. \quad (1)$$

Nhận xét:

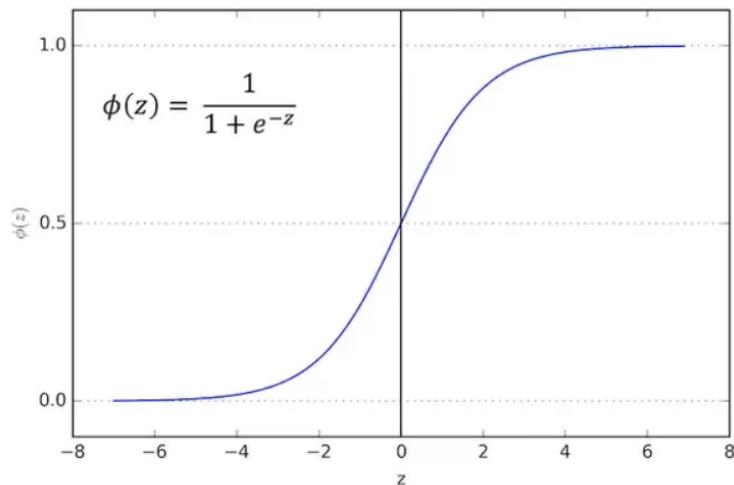
- Nếu các hệ số W và D đều nhỏ hơn 1 thì khi tính gradient ở các layer đầu ta sẽ phải nhân tích của rất nhiều số nhỏ hơn 1 nên giá trị sẽ tiến dần về 0 và bước cập nhật hệ số trong gradient descent trở nên vô nghĩa và các hệ số neural network sẽ không học được nữa. => **Vanishing gradient**

- Nếu các hệ số W và D đều lớn hơn 1 thì khi tính gradient ở các layer đầu ta sẽ phải nhân tích của rất nhiều số lớn hơn 1 nên giá trị sẽ tiến dần về vô cùng và bước cập nhật hệ số trong gradient descent trở nên không chính xác và các hệ số neural network sẽ không học được nữa. => **Exploding gradient**

Cách giải quyết vanishing/exploding gradient là lựa chọn các giá trị khởi tạo cho hệ số phù hợp và chọn activation function phù hợp.

12.5.3 Một số activation thông dụng

Sigmoid activation function

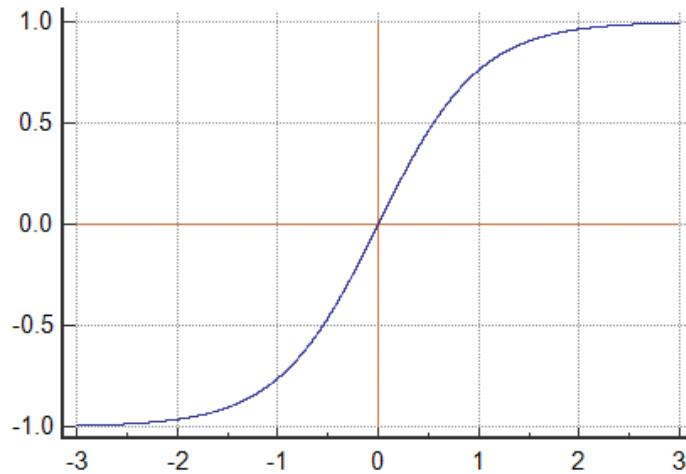


Hình 12.8: Hàm sigmoid

Đạo hàm hàm sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}} \Rightarrow \frac{d(\sigma(x))}{dx} = \sigma(x) * (1 - \sigma(x)), \text{ do } \sigma(x) > 0 \Rightarrow \sigma(x) * (1 - \sigma(x)) \leq \frac{1}{4}$$

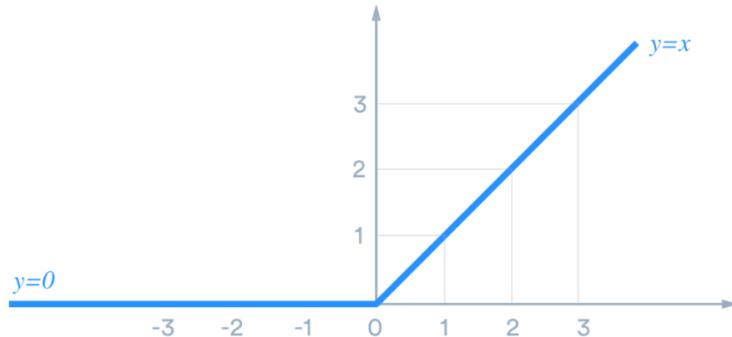
Ví dụ $(\frac{1}{4})^{20} = 9 * 10^{-13}$ nên nếu bạn nhìn vào công thức (1) ở trên thì ở những layer đầu tiên sẽ bị **vanishing gradient**.

Tanh activation function

Hình 12.9: Hàm tanh

Hàm tanh: $g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, giá trị $g(x)$ trong đoạn (-1,1)

Đạo hàm hàm tanh: $\frac{d(g(x))}{dx} = 1 - g(x)^2 \leq 1$. Do đó khi dùng tanh activation function sẽ bị vanishing gradient.

ReLU activation function

Hình 12.10: Hàm tanh

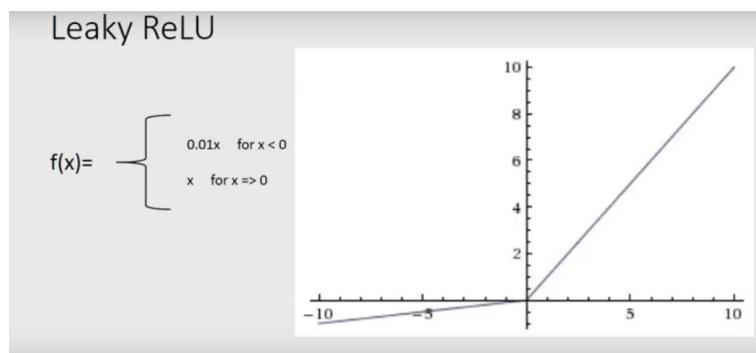
Hàm relu (rectified linear unit): $y = \max(0, x)$

Nhận xét:

- Hàm ReLU activation đơn giản để tính => thời gian train model nhanh hơn.
- Đạo hàm là 1 với $x \geq 0$ nên không bị vanishing gradient.

Tuy nhiên với các node có giá trị nhỏ hơn 0, qua ReLU activation sẽ thành 0, hiện tượng đấy gọi là "Dying ReLU". Nếu các node bị chuyển thành 0 thì sẽ không có ý nghĩa với bước linear activation ở lớp tiếp theo và các hệ số tương ứng từ node đấy cũng không được cập nhật với gradient descent. => Leaky ReLU ra đời.

Leaky ReLU



Hình 12.11: Hàm Leaky ReLU

Hàm Leaky ReLU có các điểm tốt của hàm ReLU và giải quyết được vấn đề Dying ReLU bằng cách xét một độ dốc nhỏ cho các giá trị âm thay vì để giá trị là 0.

Lời khuyên: Mặc định nên dùng ReLU làm hàm activation. Không nên dùng hàm sigmoid.

12.6 Batch Normalize

Một trong những giả định chính trong được đưa ra trong quá trình huấn luyện một mô hình học máy đó là phân phối của dữ liệu được giữ nguyên trong suốt quá trình training. Đối với các mô hình tuyến tính, đơn giản là ánh xạ input với output thích hợp, điều kiện này luôn được thỏa mãn. Tuy nhiên, trong trường hợp Neural Network với các lớp được xếp chồng lên nhau, ảnh hưởng của các hàm activation non-linear, điều kiện trên không còn đúng nữa.

Trong kiến trúc neural network, đầu vào của mỗi lớp phụ thuộc nhiều vào tham số của toàn bộ các lớp trước đó. Hậu quả là trong quá trình backprop, các trọng số của một lớp được cập nhật dẫn đến những thay đổi về mặt dữ liệu sau khi đi qua lớp đó, những thay đổi này bị khuyếch đại khi mạng trở nên sâu hơn và cuối cùng làm phân phối của bản đồ đặc trưng (feature map) thay đổi, đây được gọi là hiện tượng covariance shifting. Khi huấn luyện, các lớp luôn phải điều chỉnh trọng số để đáp ứng những thay đổi về phân phối dữ liệu nhận được từ các lớp trước, điều này làm chậm quá trình hội tụ của mô hình.

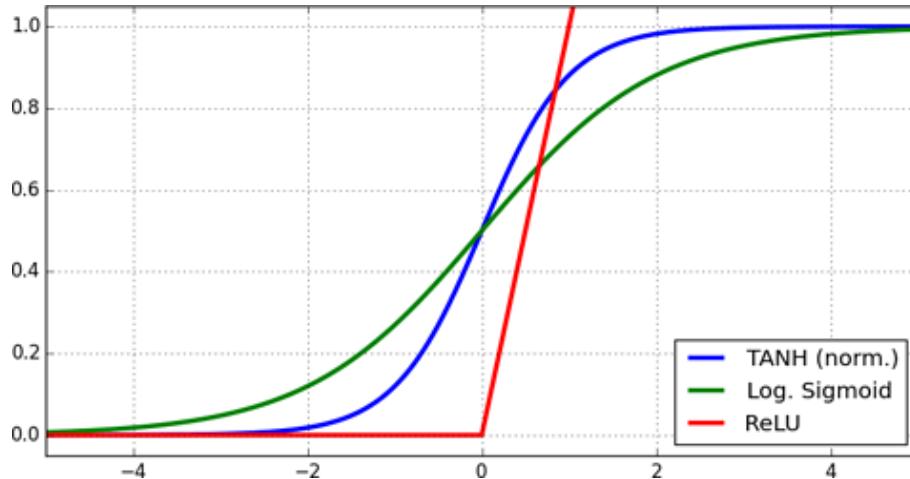
12.6.1 Phân tích nguyên nhân

Vấn đề 1 : Khi dữ liệu chứa nhiều thành phần lớn hơn nhỏ hơn 0 và không phân bố quanh giá trị trung bình 0 (Non zero mean), kết hợp với việc phương sai lớn (high variance) làm cho dữ liệu chứa nhiều thành phần rất lớn hoặc rất nhỏ. Trong quá trình cập nhật trọng số bằng gradient descent, giá trị của dữ liệu ảnh hưởng trực tiếp lên giá trị đạo hàm (gradient), do đó làm giá trị gradient trở nên quá lớn hoặc quá nhỏ, như chúng ta đã biết điều này không hề tốt chút nào. Hiện tượng trên xuất hiện khá phổ biến, phụ thuộc nhiều vào việc khởi tạo trọng số, và có xu hướng nghiêm trọng hơn khi mạng ngày càng sâu.
=> Cần một bước normalize các thành phần dữ liệu về cùng mean và chuẩn hóa variance.

Vấn đề 2 : Các hàm activation non-linear như sigmoid, relu, tanh,... đều có ngưỡng hay vùng bão hòa. Khi lan truyền thẳng, nếu dữ liệu có các thành phần quá lớn hoặc quá nhỏ, sau khi đi qua các hàm activation, các thành phần này sẽ rơi vào vùng bão hòa và có đầu ra giống nhau. Điều này dẫn đến luồng dữ liệu sau đó trở nên giống nhau khi lan truyền trong mạng (covariance

shifting), lúc này các lớp còn lại trong mạng không còn phân biệt được các đặc trưng khác nhau. Ngoài ra, đạo hàm tại ngưỡng của các hàm activation bằng 0, điều này cũng khiến mô hình bị vanishing gradient.

=> Cần một bước normalize dữ liệu trước khi đi qua hàm activation.



Hình 12.12: Đồ thị các hàm activation

12.6.2 Batch Normalization ra đời

Batch normalization thực hiện việc chuẩn hóa (normalizing) và zero centering (mean subtracting) dữ liệu trước khi đưa qua hàm activation (giá trị trung bình (mean) sẽ được đưa về 0 và phương sai (variance) sẽ được đưa về 1). Để thực hiện 2 công việc trên, batch normalization tính toán phương sai và độ lệch chuẩn của dữ liệu dựa trên các batches, rồi sử dụng 2 tham số γ và β tinh chỉnh đầu ra.

Batch normalization:

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2$$

$$\hat{x}^{(i)} = (x^{(i)} - \mu_B) / (\sqrt{\sigma_B^2 + \xi})$$

$$z_{(i)} = \gamma \hat{x} + \beta$$

Trong đó:

μ_B là giá trị trung bình của batch B

σ_B^2 là phương sai của batch B

$\hat{x}^{(i)}$ là giá trị của mẫu thứ i trong batch B sau khi được normalize và zero centering

$z_{(i)}$ là đầu ra của mẫu thứ i trong batch B

γ là scaling parameter của lớp

β là shifting parameter của lớp

ξ là smoothing parameter, tránh xảy ra việc chia cho 0, giá trị rất nhỏ

Chú ý: γ và β là 2 tham số được học trong quá trình training.

12.6.3 Hiệu quả của batch normalization

- Batch normalization đưa dữ liệu về zero mean và chuẩn hóa variance trước khi đưa qua activation function nhờ đó giải quyết các vấn đề vanishing gradient hay exploding gradient.
- Batch normalization cho phép learning rate lớn trong quá trình huấn luyện.
- Batch-Norm giảm thiểu sự ảnh hưởng của quá trình khởi tạo trọng số ban đầu.
- Batch-Norm chuẩn hóa dữ liệu đầu ra của các layer giúp model trong quá trình huấn luyện không bị phụ thuộc vào một thành phần trọng số nhất định. Do đó, Batch-norm còn được sử dụng như một regularizer giúp giảm overfitting

12.7 Bài tập

1. Dùng tất cả kiến thức của bài này để cải thiện accuracy và loss function của các model trong bài phân loại ảnh và ô tô tự lái.

Computer Vision Task

VII

13	Object detection với Faster R-CNN ... 193
13.1	Bài toán object detection
13.2	Faster R-CNN
13.3	Ứng dụng object detection
13.4	Bài tập
14	Bài toán phát hiện biển số xe máy Việt Nam 205
14.1	Lời mở đầu
14.2	Chuẩn bị dữ liệu
14.3	Huấn luyện mô hình
14.4	Dự đoán
15	Image segmentation với U-Net 217
15.1	Bài toán image segmentation
15.2	Mạng U-Net với bài toán semantic segmentation
15.3	Bài tập

13. Object detection với Faster R-CNN

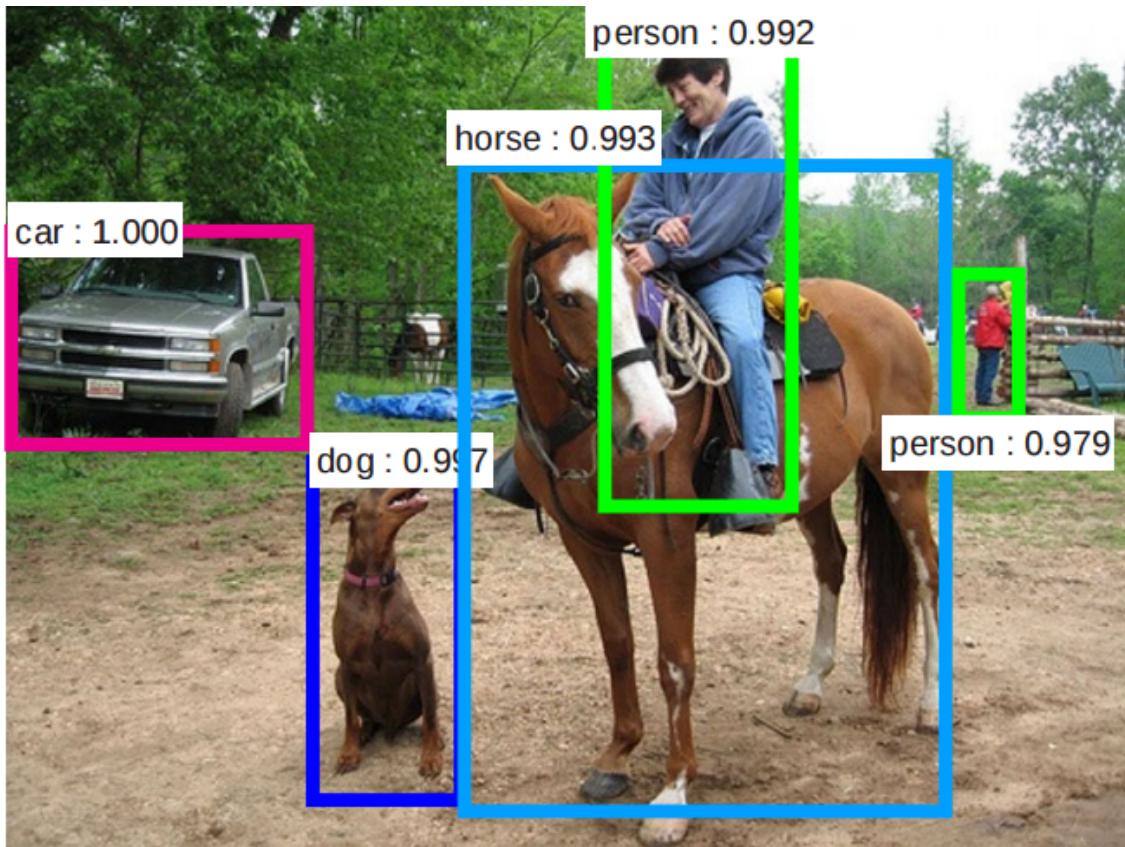
13.1 Bài toán object detection

Trong bài 7, sách đã giới thiệu về ứng dụng mô hình CNN cho bài toán phân loại ảnh, tuy nhiên các ảnh input của bài toán phân loại chỉ bao gồm 01 đối tượng cụ thể như chữ số hay 01 loài hoa.



Hình 13.1: Ví dụ ảnh trong bài toán phân loại ảnh

Tuy nhiên là ảnh trong cuộc sống bình thường thì không chỉ có 01 đối tượng mà thường chứa nhiều các đối tượng khác. Từ đó nảy sinh vấn đề cần tìm vị trí của từng đối tượng trong ảnh. Đó là bài toán object detection.



Hình 13.2: Ví dụ output của object detection [22]

Bài toán object detection có input là ảnh màu và output là vị trí của các đối tượng trong ảnh. Ta thấy nó bao gồm 2 bài toán nhỏ:

- Xác định các bounding box (hình chữ nhật) quanh đối tượng.
- Với mỗi bounding box thì cần phân loại xem đây là đối tượng gì (chó, ngựa, ô tô,...) với bao nhiêu phần trăm chắc chắn.

Việc lựa chọn có bao nhiêu loại đối tượng thì phụ thuộc vào bài toán mà ta đang giải quyết.

Bạn tự hỏi liệu mô hình CNN có giải quyết được bài toán object detection không? Vấn đề chính là vì không biết trước có bao nhiêu đối tượng trong ảnh, nên không thiết kế được output layer hiệu quả => mô hình CNN truyền thống không giải quyết được => R-CNN (regional convolutional neural network) ra đời.

13.2 Faster R-CNN

13.2.1 R-CNN (Region with CNN feature)

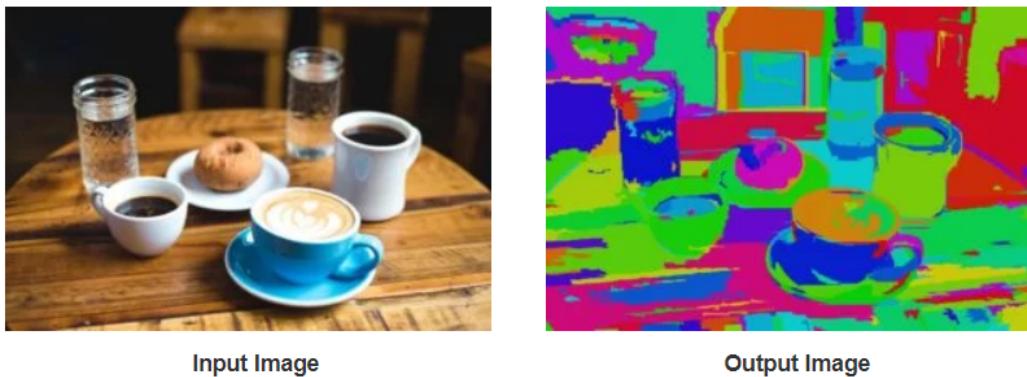
Ý tưởng thuật toán R-CNN khá đơn giản

- Bước 1: Dùng Selective Search algorithm để lấy ra khoảng 2000 bounding box trong input mà có khả năng chứa đối tượng.
- Bước 2: Với mỗi bounding box ta xác định xem nó là đối tượng nào (người, ô tô, xe đạp,...)

Selective search algorithm

Input của thuật toán là ảnh màu, output là khoảng 2000 region proposal (bounding box) mà có khả năng chứa các đối tượng.

Đầu tiên ảnh được segment qua thuật toán Graph Based Image Segmentation, vì thuật toán dựa vào lý thuyết đồ thị và không áp dụng deep learning nên sách không giải thích chi tiết, bạn có thể đọc theo link ở dưới để tìm hiểu thêm.



Hình 13.3: Output sau khi thực hiện graph based image segmentation [23]

Nhận xét: Ta không thể dùng mỗi màu trong output để làm 1 region proposal được vì:

- Mỗi đối tượng trong ảnh có thể chứa nhiều hơn 1 màu.
- Các đối tượng bị che mất một phần như cái đĩa dưới cái chén không thể xác định được.
=> Cần nhóm các vùng màu với nhau để làm region proposal.

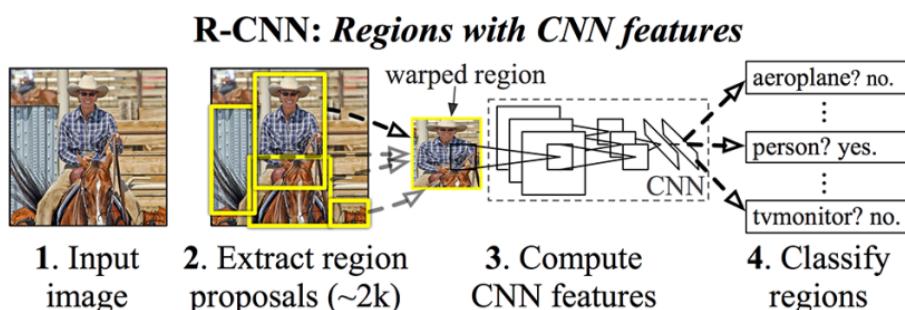
Tiếp theo, các vùng màu được nhóm với nhau dựa trên độ tương đồng về màu sắc, hướng gradient, kích thước,...

Cuối cùng các region proposal được xác định dựa trên các nhóm vùng màu.

Tài liệu cho bạn nào muốn đọc thêm ở

Phân loại region proposal

Bài toán trở thành phân loại ảnh cho các region proposal. Do thuật toán selective search cho tới 2000 region proposal nên có rất nhiều region proposal không chứa đối tượng nào. Vậy nên ta cần thêm 1 lớp background (không chứa đối tượng nào). Ví dụ như hình dưới ta có 4 region proposal, ta sẽ phân loại mỗi bounding box là người, ngựa hay background.



Hình 13.4: Các bước trong RCNN [8]

Sau đó các region proposal được resize lại về cùng kích thước và thực hiện transfer learning với feature extractor, sau đó các extracted feature được cho vào thuật toán SVM để phân loại ảnh.

Bên cạnh đó thì extracted feature cũng được dùng để dự đoán 4 offset values cho mỗi cạnh. Ví dụ như khi region proposal chứa người nhưng chỉ có phần thân và nửa mặt, nửa mặt còn lại không có trong region proposal đó thì offset value có thể giúp mở rộng region proposal để lấy được toàn bộ người.

Vấn đề với R-CNN

Hồi mới xuất hiện thì thuật toán hoạt động khá tốt cho với các thuật toán về computer vision trước đó nhờ vào CNN, tuy nhiên nó vẫn có khá nhiều hạn chế:

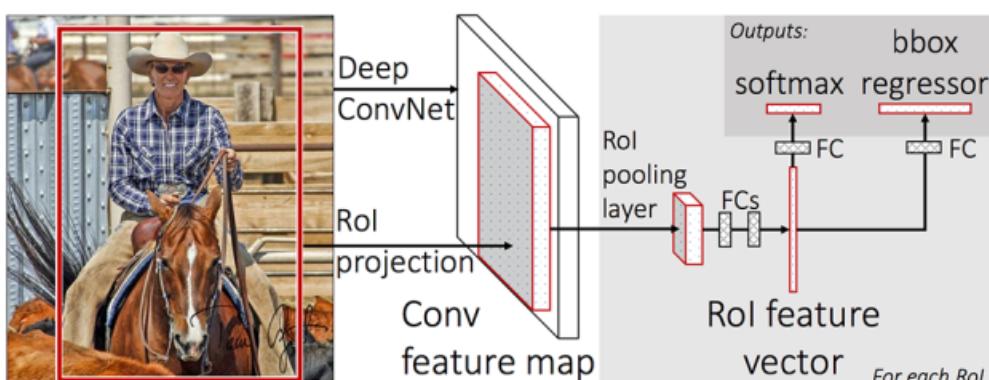
- Vì với mỗi ảnh ta cần phân loại các class cho 2000 region proposal nên thời gian train rất lâu.
- Không thể áp dụng cho real-time vì mỗi ảnh trong test set mất tới 47s để xử lý.

13.2.2 Fast R-CNN

Khoảng 1.5 năm sau đó, Fast R-CNN được giới thiệu bởi cùng tác giả của R-CNN, nó giải quyết được một số hạn chế của R-CNN để cải thiện tốc độ.

Tương tự như R-CNN thì Fast R-CNN vẫn dùng selective search để lấy ra các region proposal. Tuy nhiên là nó không tách 2000 region proposal ra khỏi ảnh và thực hiện bài toán image classification cho mỗi ảnh. Fast R-CNN cho cả bức ảnh vào ConvNet (một vài convolutional layer + max pooling layer) để tạo ra convolutional feature map.

Sau đó các vùng region proposal được lấy ra tương ứng từ convolutional feature map. Tiếp đó được Flatten và thêm 2 Fully connected layer (FCs) để dự đoán lớp của region proposal và giá trị offset values của bounding box.



Hình 13.5: Các bước trong Fast RCNN [7]

Tuy nhiên là kích thước của các region proposal khác nhau nên khi Flatten sẽ ra các vector có kích thước khác nhau nên không thể áp dụng neural network được. Thử nhìn lại xem ở trên R-CNN đã xử lý như thế nào? Nó đã resize các region proposal về cùng kích thước trước khi dùng transfer learning. Tuy nhiên ở feature map ta không thể resize được, nên ta phải có cách gì đấy để chuyển các region proposal trong feature map về cùng kích thước => Region of Interest (ROI) pooling ra đời.

Region of Interest (ROI) pooling

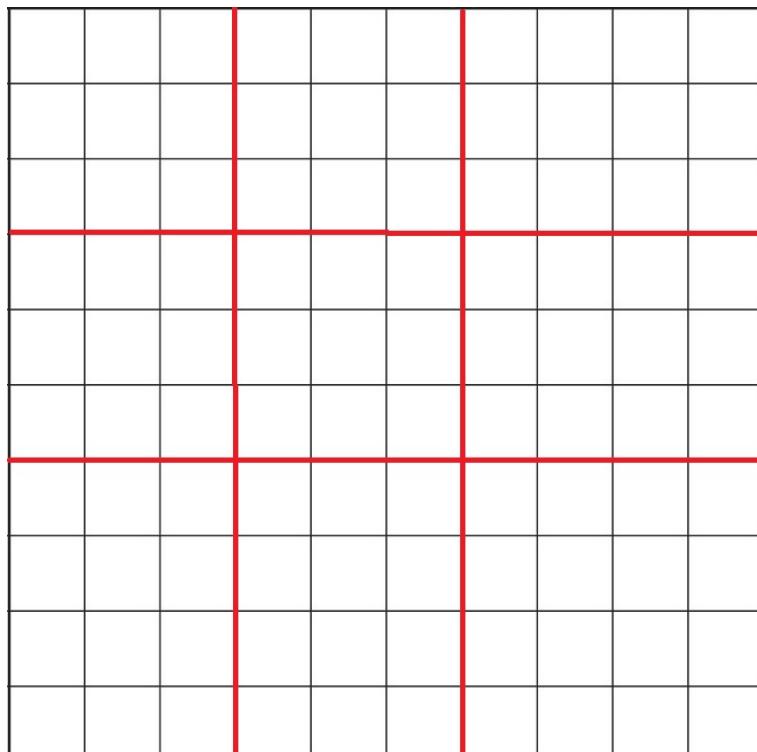
ROI pooling là một dạng của pooling layer. Điểm khác so với max pooling hay average pooling là bất kể kích thước của tensor input, ROI pooling luôn cho ra output có kích thước cố định được định nghĩa trước.

Ta kí hiệu a/b là phần nguyên của a khi chia cho b và $a\%b$ là phần dư của a khi chia cho b . Ví dụ: $10/3 = 3$ và $10\%3 = 1$.

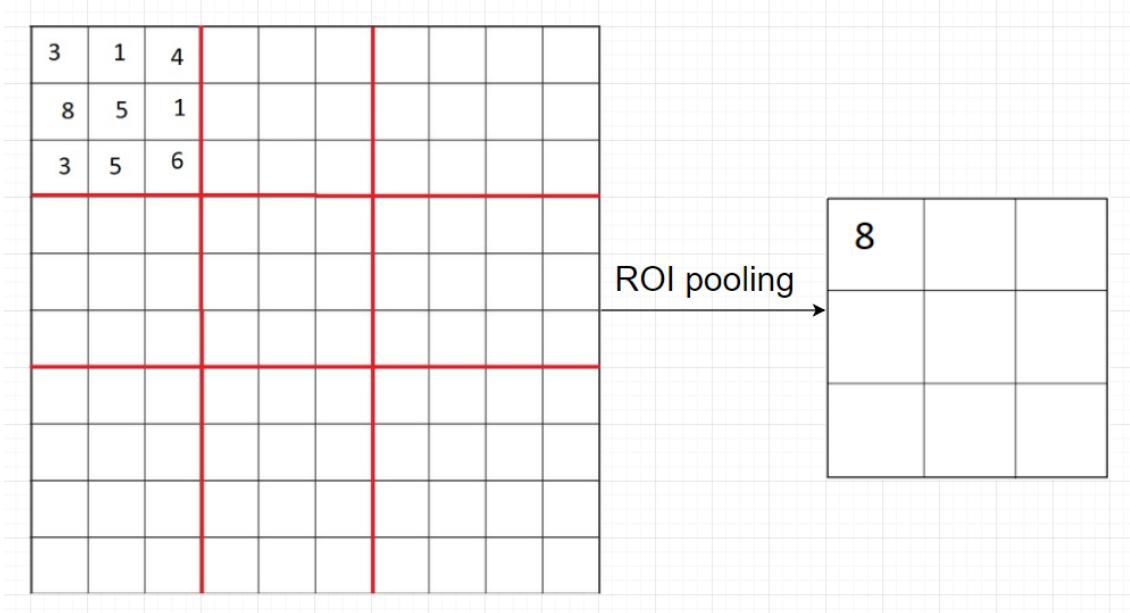
Gọi input của ROI pooling kích thước $m*n$ và output có kích thước $h*k$ (thông thường h, k nhỏ ví dụ $7*7$).

- Ta chia chiều rộng thành h phần, ($h-1$) phần có kích thước m/h , phần cuối có kích thước $m/h + m\%h$.
- Tương tự ta chia chiều dài thành k phần, ($k-1$) phần có kích thước n/k , phần cuối có kích thước $n/k + n\%k$.

Ví dụ $m=n=10$, $h=k=3$, do $m/h = 3$ và $m\%h = 1$, nên ta sẽ chia chiều rộng thành 3 phần, 2 phần có kích thước 3, và 1 phần có kích thước 4.



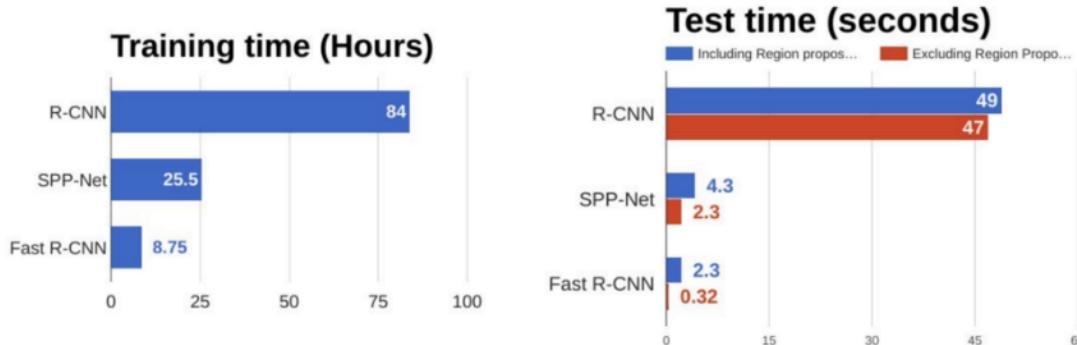
Sau đó với mỗi khối được tạo ra bằng các đường đỏ và cạnh, ta thực hiện max pooling lấy ra 1 giá trị.



Hình 13.6: Thực hiện ROI pooling

Ta có thể thấy là kích thước sau khi thực hiện ROI pooling về đúng $h*k$ như ta mong muốn.

Đánh giá Fast R-CNN



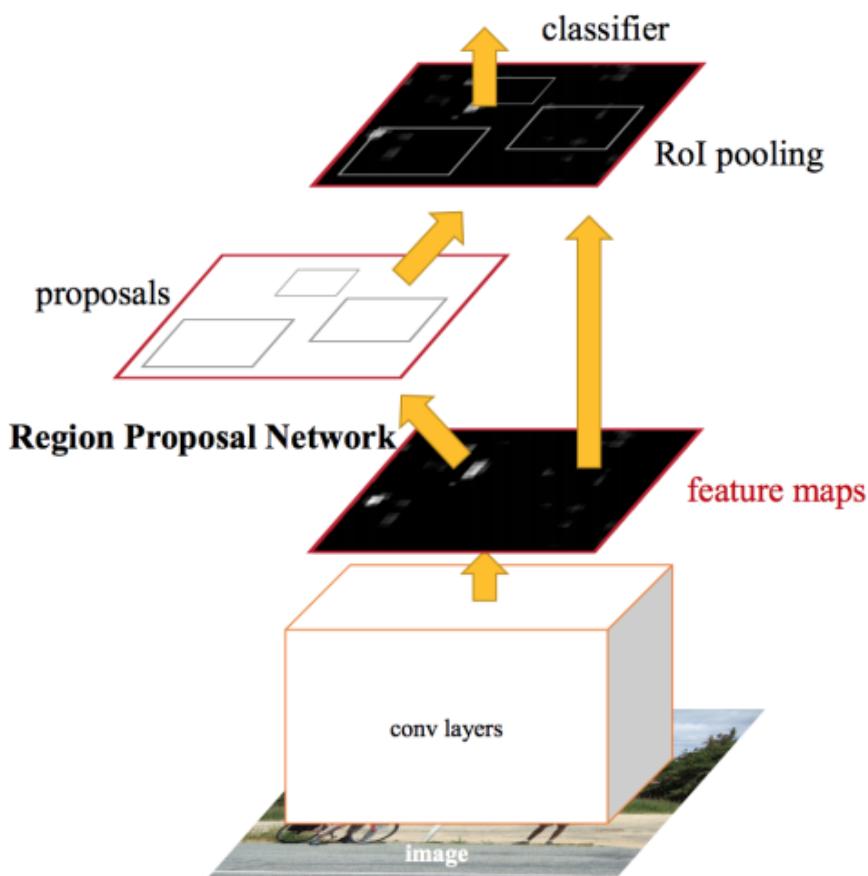
Hình 13.7: So sánh thời train train và test giữa R-CNN và Fast R-CNN [17]

Fast R-CNN khác với R-CNN là nó thực hiện feature map với cả ảnh sau đó với lấy các region proposal ra từ feature map, còn R-CNN thực hiện tách các region proposal ra rồi mới thực hiện CNN trên từng region proposal. Do đó Fast R-CNN nhanh hơn đáng kể nhờ tối ưu việc tính toán bằng Vectorization.

Tuy nhiên nhìn hình trên ở phần test time với mục Fast R-CNN thì thời gian tính region proposal rất lâu và làm chậm thuật toán => Cần thay thế thuật toán selective search. Giờ người ta nghĩ đến việc dùng deep learning để tạo ra region proposal => Faster R-CNN ra đời.

13.2.3 Faster R-CNN

Faster R-CNN không dùng thuật toán selective search để lấy ra các region proposal, mà nó thêm một mạng CNN mới gọi là Region Proposal Network (RPN) để tìm các region proposal.

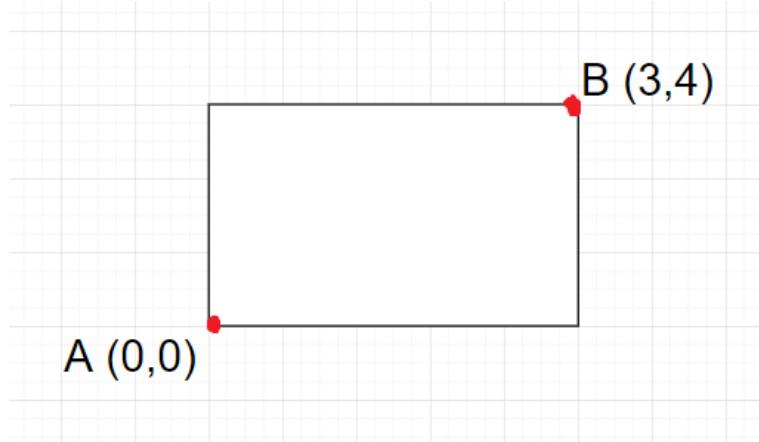


Hình 13.8: Kiến trúc mới Faster R-CNN [18]

Đầu tiên cả bức ảnh được cho qua pre-trained model để lấy feature map. Sau đó feature map được dùng cho Region Proposal Network để lấy được các region proposal. Sau khi lấy được vị trí các region proposal thì thực hiện tương tự Fast R-CNN.

Region Proposal Network (RPN)

Input của RPN là feature map và output là các region proposal. Ta thấy các region proposal là hình chữ nhật.

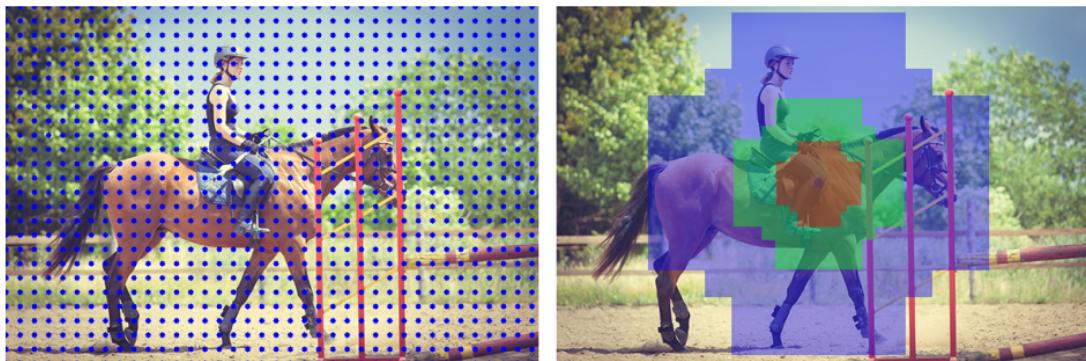


Mà một hình chữ nhật được xác định bằng 2 điểm ở 2 góc, ví dụ $A(x_{min}, y_{min})$ và $B(x_{max}, y_{max})$. Nhận xét:

- Khi RPN dự đoán ta phải ràng buộc $x_{min} < x_{max}$ và $y_{min} < y_{max}$.
 - Hơn nữa các giá trị x,y khi dự đoán có thể ra ngoài khôi bức ảnh
- => Cần một kĩ thuật mới để biểu diễn region proposal => Anchor ra đời.

Ý tưởng là thay vì dự đoán 2 góc ta sẽ dự đoán điểm trung tâm (x_{center} , y_{center}) và width, height của hình chữ nhật. Như vậy mỗi anchor được xác định bằng 4 tham số (x_{center} , y_{center} , width, height).

Vì không sử dụng Selective search nên RPN ban đầu cần xác định các anchor box có thể là region proposal, sau đó qua RPN thì chỉ output những anchor box chắc chắn chứa đối tượng.



Hình 13.9: Ví dụ về anchor [21]

Ảnh bên trái kích thước $400 * 600$ pixel, tác tâm của anchor box màu xanh, cách nhau 16 pixel => có khoảng $(400*600)/(16*16) = 938$ tâm. Do các object trong ảnh có thể có kích thước và tỉ lệ khác nhau nên với mỗi tâm ta định nghĩa 9 anchors với kích thước 64×64 , 128×128 , 256×256 , mỗi kích thước có 3 tỉ lệ tương ứng: $1 : 1$, $1 : 2$ và $2 : 1$.

Giống như hình bên phải với tâm ở giữa 3 kích thước ứng với màu da cam, xanh lam, xanh lục và với mỗi kích thước có 3 tỉ lệ.

=> Số lượng anchor box giờ là $938 * 9 = 8442$ anchors. Tuy nhiên sau RPN ta chỉ giữ lại khoảng

1000 anchors box để thực hiện như trong Fast R-CNN.

Việc của RPN là lấy ra các region proposal giống như selective search thôi chứ không phải là phân loại ảnh.

Mô hình RPN khá đơn giản, feature map được cho qua Conv layer 3*3, 512 kernels. Sau đó với mỗi anchor lấy được ở trên, RPN thực hiện 2 bước:

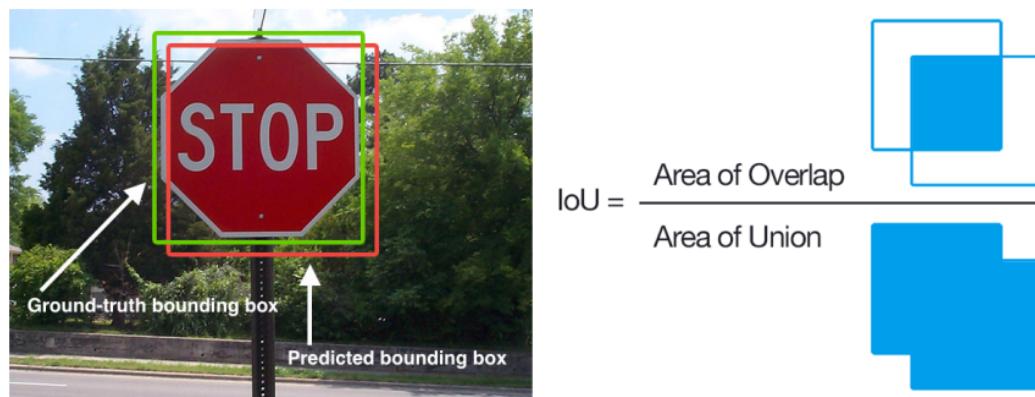
1. Dự đoán xem anchor đây là foreground (chứa object) hay background (không chứa object)
2. Dự đoán 4 offset value cho x_center, y_center, width, height cho các anchor.

Nhận xét: có rất nhiều anchor bị chồng lên nhau nên non-maxima suppression được dùng để loại bỏ các anchor chồng lên nhau.

Sau cùng dựa vào phần trăm dự đoán background RPN sẽ lấy N anchor (N có thể 2000, 1000, thậm chí 100 vẫn chạy tốt) để làm region proposal.

Intersection over Union (IoU)

IoU được sử dụng trong bài toán object detection, để đánh giá xem bounding box dự đoán đối tượng khớp với ground truth thật của đối tượng.



Hình 13.10: Ví dụ về IoU [11]

Ví dụ về hệ số IoU, nhận xét:

- Chỉ số IoU trong khoảng [0,1]
- IoU càng gần 1 thì bounding box dự đoán càng gần ground truth



Hình 13.11: Chỉ số IoU [11]

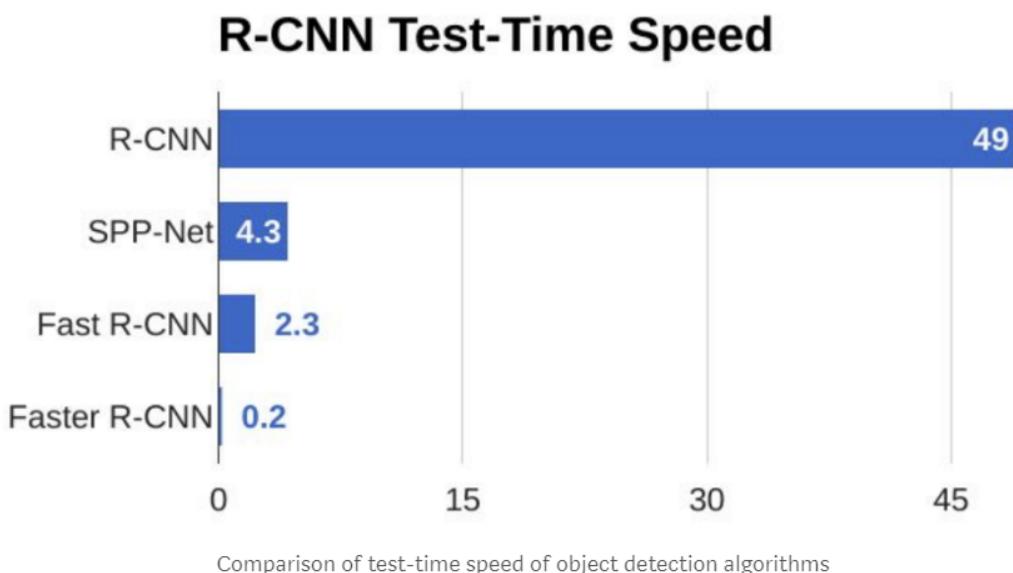
Ngoài ra thì hệ số IoU cũng được dùng để đánh giá độ khớp của 2 hình chữ nhật.

Non-maxima suppression

Ở trong Region Proposal Network đầu tiên ta có khoảng 9000 anchor box (tập Input) tuy nhiên ta chỉ muốn giữ lại 100 anchor (tập Ouput) làm region proposal. Ta sẽ làm như sau:

- Chọn ra anchor box (A) có xác suất là foreground lớn nhất trong tập Input.
- Thêm A vào tập Ouput.
- Loại bỏ A và các anchor box trong tập Input mà có hệ số IoU với A lớn hơn 0.5 ra khỏi tập Input.
- Kiểm tra nếu tập Input rỗng hoặc tập Output đủ 100 anchor thì dừng lại, nếu không quay lại bước 1.

Kết quả của Faster R-CNN



Nhìn ở hình trên ta thấy Faster R-CNN nhanh hơn hẳn các dòng R-CNN trước đó, vì vậy có thể

dùng cho real-time object detection

13.3 Ứng dụng object detection

- Tự động điểm danh: Xác định được vị trí mặt của các học sinh và phân loại các học sinh trong lớp.
- Hỗ trợ ô tô tự lái: Xác định được vị trí và phân loại được các phương tiện giao thông, người đi bộ.
- Dự đoán hành vi: Xác định được vị trí và phân loại người => track được người => dùng RNN để dự đoán hành vi.

13.4 Bài tập

1. Tìm code implement Faster RCNN trên github, download pre-trained model về và chạy thử với ảnh mới.
2. Customize dataset tùy ý và dùng pre-trained model ở trên để train Faster RCNN với dữ liệu mới.

14. Bài toán phát hiện biển số xe máy Việt Nam

14.1 Lời mở đầu

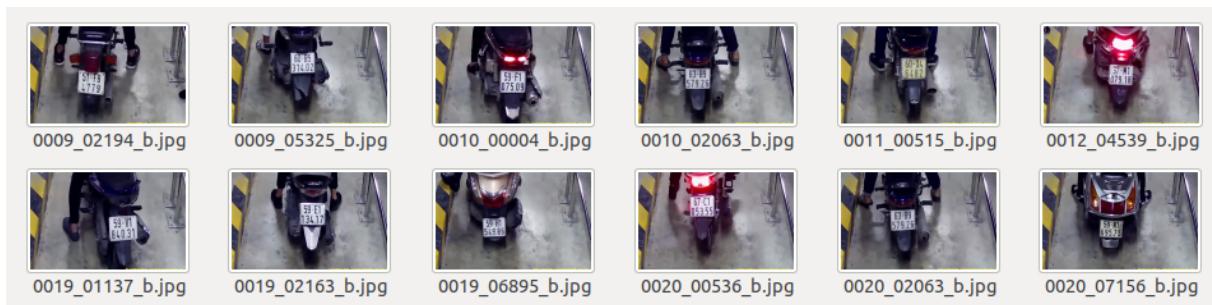
Bài toán nhận diện biển số xe Việt Nam là một bài toán không còn mới, đã được phát triển dựa trên các phương pháp xử lý ảnh truyền thống và cả những kỹ thuật mới sử dụng Deep Learning. Trong bài toán này tôi chỉ phát triển bài toán phát hiện biển số (một phần trong bài toán nhận diện biển số) dựa trên thuật toán YOLO-Tinyv4 với mục đích:

- Hướng dẫn chuẩn bị dữ liệu cho bài toán Object Detection.
- Hướng dẫn huấn luyện YOLO-TinyV4 dùng darknet trên Google Colab.

14.2 Chuẩn bị dữ liệu

14.2.1 Đánh giá bộ dữ liệu

Trong bài viết tôi sử dụng bộ dữ liệu biển số xe máy Việt Nam chứa 1750 ảnh, bạn đọc có thể tải tại [đây](#).



Hình 14.1: Ảnh biển số trong bộ dữ liệu

Ảnh biển số xe được trong bộ dữ liệu được chụp từ một camera tại vị trí kiểm soát xe ra vào trong hầm. Do vậy:

- Kích thước các biển số xe không có sự đa dạng, do khoảng cách từ camera đến biển số xe xấp xỉ gần bằng nhau giữa các ảnh.
- Ảnh có độ sáng thấp và gần giống nhau do ảnh được chụp trong hầm chung cư.

=> Cần làm đa dạng bộ dữ liệu.

14.2.2 Các phương pháp tăng sự đa dạng của bộ dữ liệu

Đa dạng kích thước của biển số

Đa dạng kích thước bằng 2 cách:

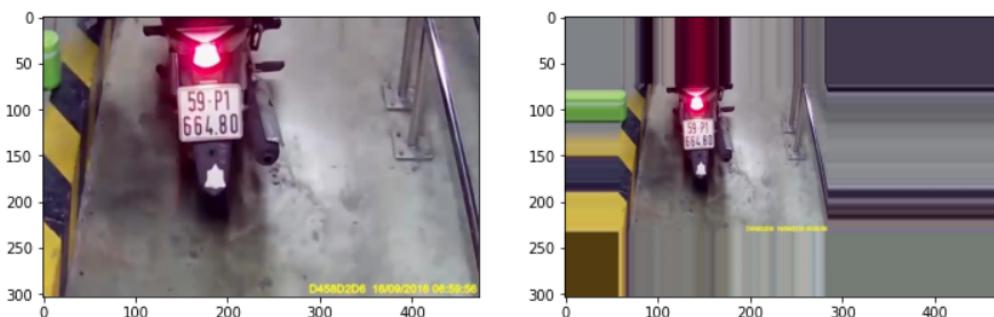
- Cách 1: Thu nhỏ kích thước biển bằng cách thêm biên kích thước ngẫu nhiên vào ảnh gốc, sau đó resize ảnh bằng kích thước ảnh ban đầu.
- Cách 2: Crop ảnh chứa biển số với kích thước ngẫu nhiên, sau đó resize ảnh bằng kích thước ảnh ban đầu.

```
# Cách 1
def add_boder(image_path, output_path, low, high):
    """
    low: kích thước biên thấp nhất (pixel)
    hight: kích thước biên lớn nhất (pixel)
    """
    # random các kích thước biên trong khoảng (low, high)
    top = random.randint(low, high)
    bottom = random.randint(low, high)
    left = random.randint(low, high)
    right = random.randint(low, high)

    image = cv2.imread(image_path)
    original_width, original_height = image.shape[1], image.shape[0]

    # sử dụng hàm của opencv để thêm biên
    image = cv2.copyMakeBorder(image, top, bottom, left, right, cv2.BORDER_REPLICATE)

    # sau đó resize ảnh bằng kích thước ban đầu của ảnh
    image = cv2.resize(image, (original_width, original_height))
    cv2.imwrite(output_path, image)
```



Hình 14.2: Ảnh thu được (bên phải) sau khi chạy hàm trên

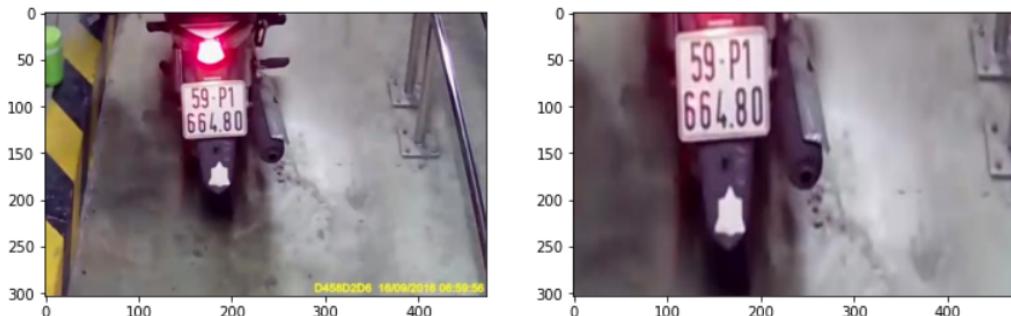
```
# Cách2
def random_crop(image_path, out_path):
    image = cv2.imread(image_path)

    original_width, original_height = image.shape[1], image.shape[0]
    x_center, y_center = original_height//2, original_width//2

    x_left = random.randint(0, x_center//2)
    x_right = random.randint(original_width-x_center//2, original_width)

    y_top = random.randint(0, y_center//2)
    y_bottom = random.randint(original_height-y_center//2, original_width)

    # crop ra vùng ảnh với kích thước ngẫu nhiên
    cropped_image = image[y_top:y_bottom, x_left:x_right]
    # resize ảnh bằng kích thước ảnh ban đầu
    cropped_image = cv2.resize(cropped_image, (original_width, original_height))
    cv2.imwrite(out_path, cropped_image)
```



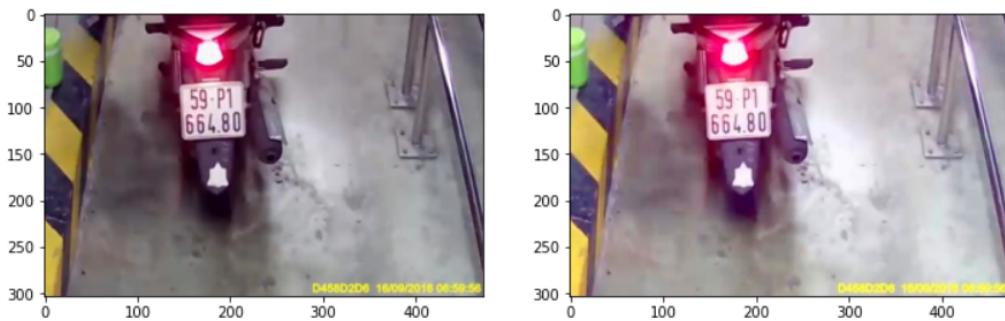
Hình 14.3: Ảnh thu được (bên phải) sau khi chạy hàm trên

Thay đổi độ sáng của ảnh

```
def change_brightness(image_path, output_path, value):
    """
    value: độ sáng thay đổi
    """
    img=cv2.imread(image_path)
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    v = cv2.add(v, value)
    v[v > 255] = 255
    v[v < 0] = 0

    final_hsv = cv2.merge((h, s, v))
    img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)

    cv2.imwrite(output_path, img)
```

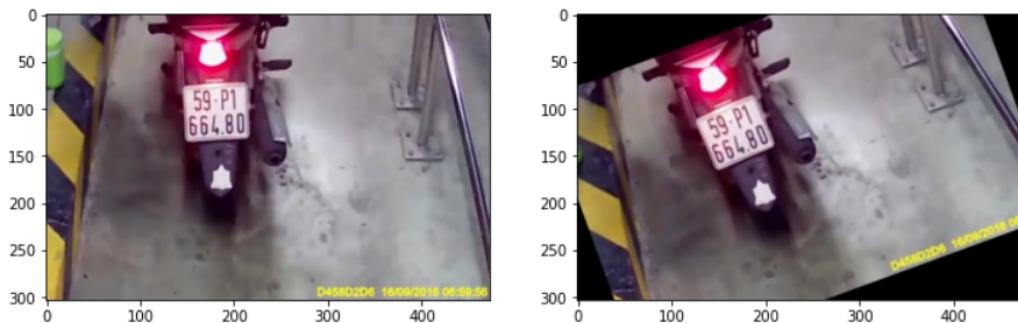


Hình 14.4: Độ sáng thay đổi (bên phải)

Xoay ảnh

```
import imutils
def rotate_image(image_path, range_angle, output_path):
    """
    range_angle: Khoảng góc quay
    """
    image = cv2.imread(image_path)
    #lựa chọn ngẫu nhiên góc quay
    angle = random.randint(-range_angle, range_angle)

    img_rot = imutils.rotate(image, angle)
    cv2.imwrite(output_path, img_rot)
```



Hình 14.5: Ảnh được xoay (bên phải)

14.2.3 Gán nhãn dữ liệu

Tool gán nhãn ở đây tôi dùng là **labelImg**, bạn đọc có thể tải và đọc hướng dẫn sử dụng tại [đây](#).



Hình 14.6: Xác định vùng biển chứa biển số

LabelImg hỗ trợ gán nhãn trên cả 2 định dạng PASCAL VOC và YOLO với phần mở rộng file annotation tương ứng là .xml và .txt.

Trong bài toán sử dụng mô hình YOLO, tôi lưu file annotation dưới dạng .txt.

```
0 0.384534 0.346535 0.201271 0.250825
```

Hình 14.7: Nội dung trong một file annotation

Mỗi dòng trong một file annotation bao gồm: <object-class> <x> <y> <width> <height>.

Trong đó: <x> <y> <width> <height> tương ứng là tọa độ trung tâm và kích thước của đối tượng. Các giá trị này đã được chuẩn hóa lại, do vậy giá trị luôn nằm trong đoạn [0,1]. **object-class** là chỉ số đánh dấu các classes.

Lưu ý: Với bài toán có nhiều nhãn, nhiều người cùng gán nhãn thì cần thống nhất với nhau trước về thứ tự nhãn. Nguyên nhân do trong file annotation chỉ lưu chỉ số (0,1,3,4,...) của nhãn chứ không lưu tên nhãn.

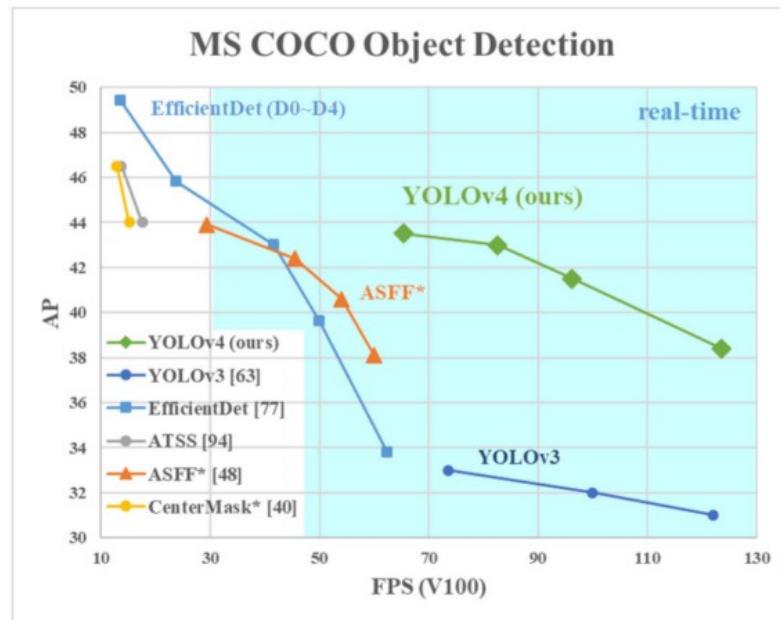
Sau khi gán nhãn xong các bạn để file annotation và ảnh tương ứng **vào cùng một thư mục**.

14.3 Huấn luyện mô hình

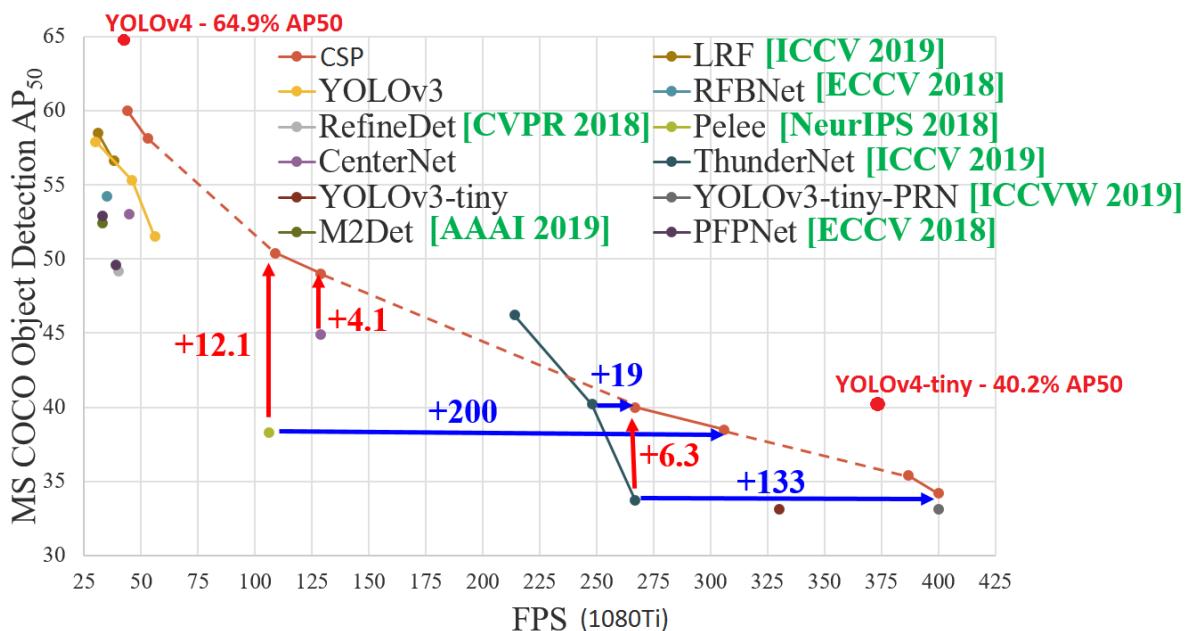
14.3.1 Giới thiệu về YOLO-Tinyv4 và darknet

YOLO-Tinyv4

YOLOv4 là thuật toán Object Detection, mới được công bố trong thời gian gần đây với sự cải thiện về kết quả đáng kể so với YOLOv3.

Hình 14.8: Sự cải thiện của YOLOv4 ([source](#))

YOLOv4 cho kết quả real-time khi chạy trên các nền tảng GPU cao cấp. Với mục đích trade-off giữa độ chính xác và tốc độ để có thể chạy trên các nền tảng CPU và GPU thấp hơn thì YOLO-Tinyv4 được ra đời.

Hình 14.9: YOLOv4 với YOLO-Tinyv4 ([source](#))

YOLOv4-tiny released: 40.2% AP50, 371 FPS (GTX 1080 Ti) / 330 FPS (RTX 2070)

- 1770 FPS - on GPU RTX 2080Ti - (416x416, fp16, batch=4) tkDNN/TensorRT [ceccocats/tkDNN#59](#) (comment)
- 1353 FPS - on GPU RTX 2080Ti - (416x416, fp16, batch=4) OpenCV 4.4.0 (including: transferring CPU->GPU and GPU->CPU) (excluding: nms, pre/post-processing) [#6067](#) (comment)
- 39 FPS - 25ms latency - on Jetson Nano - (416x416, fp16, batch=1) tkDNN/TensorRT [ceccocats/tkDNN#59](#) (comment)
- 290 FPS - 3.5ms latency - on Jetson AGX - (416x416, fp16, batch=1) tkDNN/TensorRT [ceccocats/tkDNN#59](#) (comment)
- 42 FPS - on CPU Core i7 7700HQ (4 Cores / 8 Logical Cores) - (416x416, fp16, batch=1) OpenCV 4.4.0 (compiled with OpenVINO backend) [#6067](#) (comment)
- 20 FPS on CPU ARM Kirin 990 - Smartphone Huawei P40 [#6091](#) (comment) - Tencent/NCNN library <https://github.com/Tencent/ncnn>
- 120 FPS on nVidia Jetson AGX Xavier - MAX_N - Darknet framework
- 371 FPS on GPU GTX 1080 Ti - Darknet framework

Hình 14.10: YOLO-Tinyv4 trên các nền tảng ([source](#))

Darknet

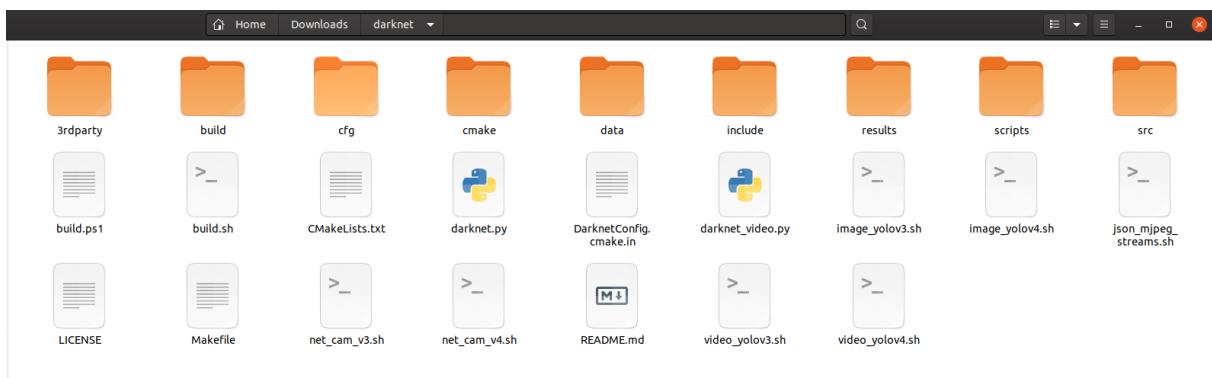
Darknet là một framework open source chuyên về Object Detection được viết bằng ngôn ngữ C và CUDA. Darknet dùng để huấn luyện các mô hình YOLO một cách nhanh chóng, dễ sử dụng.

14.3.2 Cấu hình darknet

Việc cấu hình trực tiếp trên Google Colab tương đối khó khăn với những bạn chưa quen làm việc với linux, bên cạnh đó vấn đề delay kết nối giữa Google Colab với drive trong một số trường hợp. Vì vậy tôi sẽ hướng dẫn các bạn cấu hình trên máy cá nhân, xong nén thành file .zip, sau đó đẩy lên Google Drive.

Việc cấu hình trải qua các bước:

Bước 1: Clone [darknet](#) về máy.



Hình 14.11: Nội dung trong thư mục vừa clone về.

Bước 2: Sửa file Makefile trong thư mục vừa clone về.

```

GPU=1
CUDNN=1
CUDNN_HALF=0
OPENCV=1
AVX=0
OPENMP=0
LIBSO=0
ZED_CAMERA=0
ZED_CAMERA_V2_8=0

```

Hình 14.12: Sửa một vài dòng đầu trong file.

Trong trường hợp huấn luyện mô hình trên GPU có TensorCores thì bạn đọc có thể sửa "**CUDNN_HALF=1**" để tăng tốc độ huấn luyện . Trên Google Colab có hỗ trợ GPU Tesla T4, là GPU có kiến trúc mới nhất trên Colab và có TensorCores. Vì vậy, trong trường hợp chắc chắn sẽ có được GPU Tesla T4 thì bạn đọc có thể sửa "**CUDNN_HALF=1**".

Bước 3: Tạo file yolo-tinyv4-obj.cfg.

Tạo file yolo-tinyv4-obj.cfg với nội dung tương tự file yolov4-tiny.cfg trong thư mục darknet/cfg, sau đó chỉnh sửa một số dòng:

- Dòng 6: Thay đổi batch=64. Nghĩa là: batch = số ảnh (cả file annotation) được đưa vào huấn luyện trong một batch.
- Dòng 7: Thay đổi subdivisions=16. Trong một batch được chia thành nhiều block, mỗi block chứa batch/subdivisions ảnh được đưa vào GPU xử lý tại một thời điểm. Weights của mô hình được update sau mỗi batch.
- Dòng 20: Thay đổi max_batches=classes*2000, không nhỏ hơn số ảnh trong tập huấn luyện, và không nhỏ hơn 6000 (theo [đây](#)). VD: max_batches=6000
- Dòng 22: Thay đổi steps= 80%, 90% max_batches. VD: steps=4800,5400. Sau khi huấn luyện được 80%, 90% max_batches, learning_rate sẽ được nhân với một tỷ lệ (dòng 23 trong file), mặc định là 0.1.
- Thay đổi classes=1 trong mỗi layer [yolo], dòng 217, 266.
- Thay đổi filters trong mỗi layer [convolutional] trước layer [yolo] theo công thức filters=(số class+5)*3. Trong bài toán này filters=18.

Chi tiết ý nghĩa của các tham số trong file cfg, bạn đọc có thể xem tại [đây](#).

Bước 4: Tạo file obj.names chứa tên của các class, sau đó lưu trong thư mục darknet/data.



Hình 14.13: Nội dung file obj.names

Bước 5: Tạo file obj.data, sau đó lưu trong thư mục darknet/data.

```

classes=1
train = data/train.txt
valid = data/valid.txt
names = data/obj.names
backup = backup/

```

Hình 14.14: Nội dung file obj.data

Bước 6: Đưa toàn bộ thư mục chứa ảnh và file annotation ở trên vào thư mục darknet/data.

Bước 7: Download pre-trained weights của YOLO-Tinyv4 tại [đây](#), lưu trong thư mục darknet.

Bước 8: Nén thư mục darknet thành file darknet.zip, sau đó đưa lên Google Drive.

14.3.3 Huấn luyện model trên colab

Để thực thi các lệnh command line trong colab sử dụng thêm ! trước mỗi câu lệnh.

Kiểm tra cấu hình GPU dùng command line: !nvidia-smi

```

[1] !nvidia-smi
[Mon Jul 27 16:06:42 2020]
+-----+
| NVIDIA-SMI 450.51.05      Driver Version: 418.67      CUDA Version: 10.1 |
+-----+
| GPU  Name     Persistence-M | Bus-Id     Disp.A  Volatile Uncorr. ECC | | | | | | |
| Fan  Temp   Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
| |          |          |          |          |          |          |          |
|-----+
| 0  Tesla P100-PCIE... Off  00000000:00:04.0 Off |           0 | | | | | |
| N/A   42C   P0    27W / 250W |    0MiB / 16280MiB |      0%  Default |
|          |          |          |          |          |          |          |
+-----+
+-----+
| Processes:
| GPU  GI CI     PID   Type  Process name          GPU Memory Usage |
| ID   ID          |          |          |          |          |
|-----|
| No running processes found
+-----+

```

Hình 14.15: GPU được cung cấp là Tesla P100

#Sau khi mount với drive. Chuyển đến thư mục chứa file darknet.zip vừa tải lên
#Ví dụ tôi để ở thư mục gốc của Google Drive

cd drive/My\ Drive

#Giải nén file darknet.zip
!unzip darknet.zip

#Chuyển đến thư mục darknet
cd darknet

```

#Tạo thư mục backup để lưu lại weights khi huấn luyện
#Tên thư mục phải trùng với link folder backup trong file obj.data trên
!mkdir backup

#Tạo file train.txt, valid.txt theo đoạn code
import os
import numpy as np
#"obj" là tên thư mục chứa cả ảnh và file annotation.
lst_files = os.listdir("data/obj/")
lst_images = []

for file in lst_files:
    if ".txt" not in file:
        lst_images.append(file)

#Tách 200 ảnh ra làm tập validation
random_idx = np.random.randint(0, len(lst_images), 200)

#Tạo file train.txt được đặt trong thư mục darknet/data
with open("data/train.txt", "w") as f:
    for idx in range(len(lst_images)):
        if idx not in random_idx:
            f.write("data/obj/" + lst_images[idx] + "\n")

#Tạo file valid.txt được đặt trong thư mục darknet/data
with open("data/valid.txt", "w") as f:
    for idx in random_idx:
        f.write("data/obj/" + lst_images[idx] + "\n")

#Biên dịch darknet (chỉ cần biên dịch một lần, lần sau dùng bỏ qua bước này)
!make

#Phân quyền thực thi module darknet
!chmod +x ./darknet

```

Bắt đầu quá trình huấn luyện sử dụng command line:

```
!./darknet detector train data/obj.data yolo-tinyv4-obj.cfg yolov4-tiny.conv.29 -map \
-dont_show > yolotinv4_lisenceplate.log
```

Cú pháp tổng quát để huấn luyện:

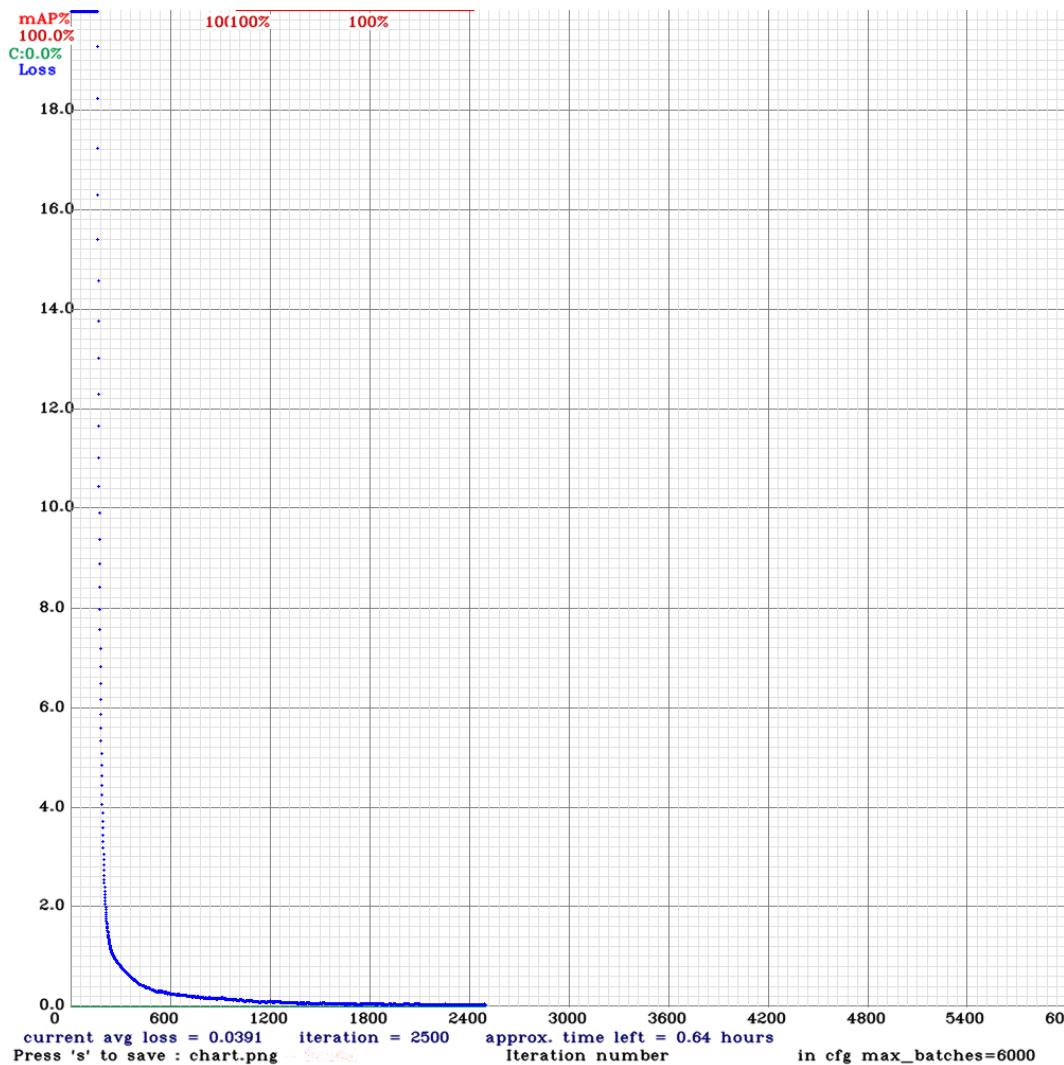
```
!./darknet detector train [data config file] [model config file] [pre-trained weights] \
-map > [file log saved]
```

-map: Dùng để hiển thị mAP được tính trên tập validation.

Nếu bạn gặp lỗi: **CUDA Error: out of memory: File exists** thì hãy quay lại sửa subdivisions=32 trong file yolo-tinyv4-obj.cfg

Theo dõi quá trình huấn luyện

Quá trình huấn luyện sẽ được lưu vào file yolotinyv4_lisenceplate.log, ngoài ra darknet tự động tạo ra ảnh chart.png lưu trong thư mục darknet và được cập nhật liên tục để theo dõi trực tiếp thông số của quá trình huấn luyện.



Hình 14.16: chart.png

Nhận xét: Quá trình huấn luyện hội tụ rất nhanh. Vì vậy có thể dừng sớm sau 2000 batches.

14.4 Dự đoán

Sau khi huấn luyện xong, toàn bộ weights sẽ được lưu trong folder backup.

```
#Danh sách các weights được lưu
!ls backup/
```

```
yolo-tinyv4-obj_1000.weights  yolo-tinyv4-obj_best.weights
yolo-tinyv4-obj_2000.weights  yolo-tinyv4-obj_last.weights
```

Để dự báo một bức ảnh sử dụng cú pháp:

```
!./darknet detector test [data config file] [model config file] [best-weights]
[image path]
#cụ thể như sau
!./darknet detector test data/obj.data yolo-tinyv4-obj.cfg \
backup/yolo-tinyv4-obj_best.weights test1.jpg
```

Kết quả dự đoán được lưu thành file predictions.jpg

```
#Hàm sau được dùng để hiển thị kết quả dự đoán lên colab
def show(path):
    import cv2
    import matplotlib.pyplot as plt

    image = cv2.imread(path)
    original_width, original_height = image.shape[1], image.shape[0]
    resized_image = cv2.resize(image, (2*original_width, 2*original_height) \
        , interpolation = cv2.INTER_CUBIC)

    resized_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB)
    plt.figure(figsize=(20,10))
    plt.axis("off")
    plt.imshow(resized_image)
    plt.show()

show("predictions.jpg")
```



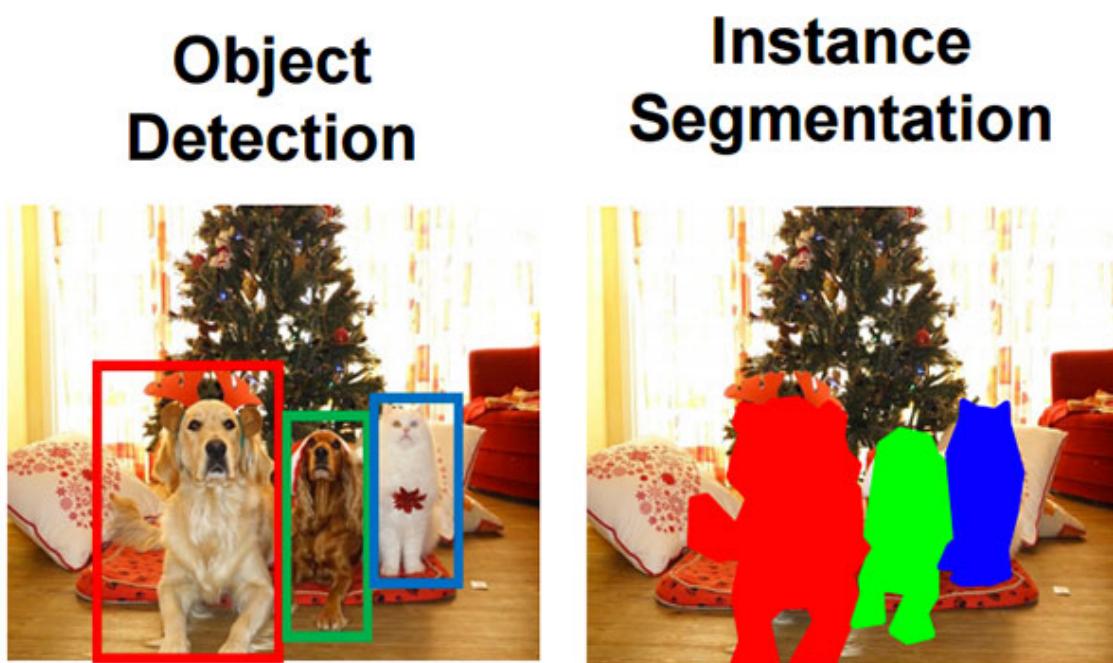
Hình 14.17: Kết quả dự đoán

Code và dataset đã gán nhãn mọi người có thể lấy ở [đây](#).

15. Image segmentation với U-Net

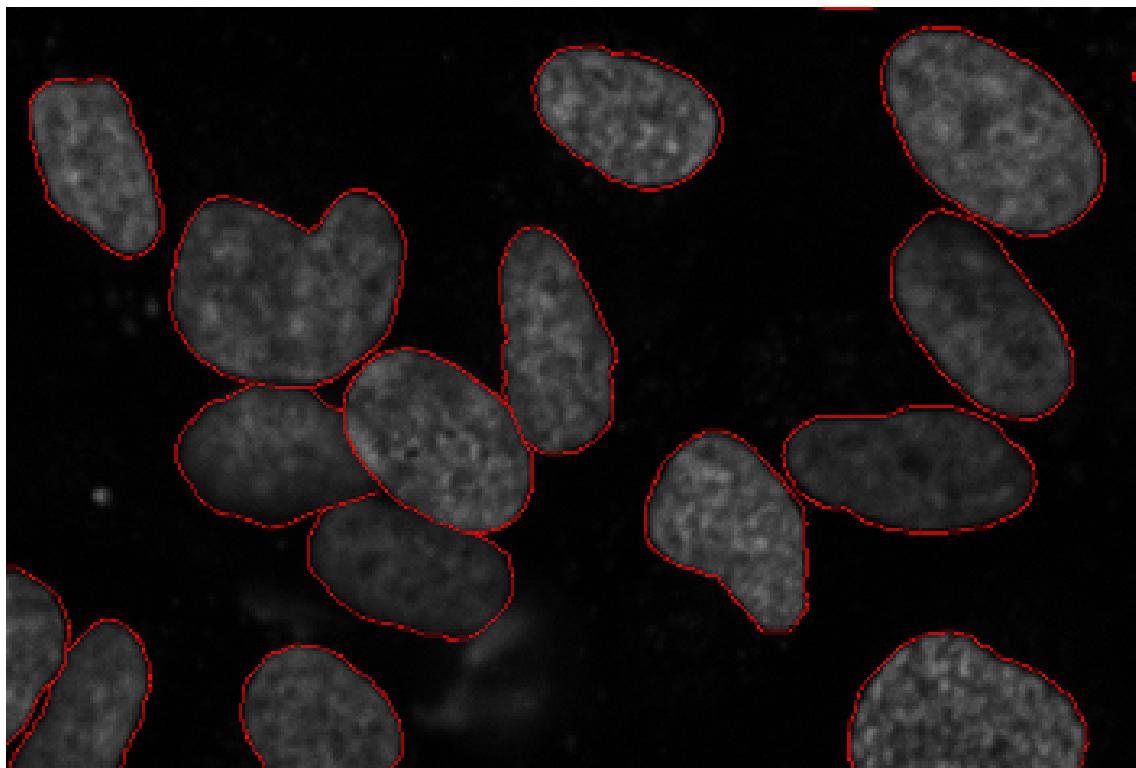
15.1 Bài toán image segmentation

Bài trước bạn đã được giới thiệu về object detection, đi tìm các bounding box quanh các đối tượng trong ảnh và sau đó phân loại các bounding box. Tuy nhiên là các bounding box thì không biểu thị được đúng hình dạng của đối tượng và có nhiều nhiễu ở trong bounding box đấy ví dụ như trong bounding box màu đỏ có cả một phần của cây thông cũng như cái gối => Image segmentation ra đời để chia ảnh thành nhiều vùng khác nhau hay tìm được đúng hình dạng của các đối tượng.



Hình 15.1: So sánh object detection và segmentation [4]

Cùng thử lấy ví dụ tại sao cần image segmentation nhé. Ung thư là một căn bệnh hiểm nghèo và cần được phát hiện sớm để điều trị. Vì hình dạng của các tế bào ung thư là một trong những yếu tố quyết định độ ác tính của bệnh, nên ta cần image segmentation để biết được chính xác hình dạng của các tế bào ung thư để có các chẩn đoán xác định. Rõ ràng object detection ở đây không giải quyết được vấn đề.

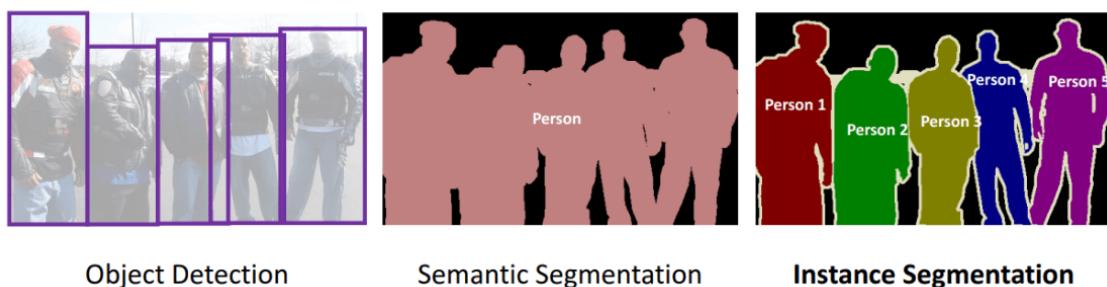


Hình 15.2: Ví dụ về segmentation [15]

15.1.1 Phân loại bài toán image segmentation

Bài toán image segmentation được chia ra làm 2 loại:

- **Semantic segmentation:** Thực hiện segment với từng lớp khác nhau, ví dụ: tất cả người là 1 lớp, tất cả ô tô là 1 lớp.
- **Instance segmentation:** Thực hiện segment với từng đối tượng trong một lớp. Ví dụ có 3 người trong ảnh thì sẽ có 3 vùng segment khác nhau cho mỗi người.



Hình 15.3: Phân loại semantic segmentation và instance segmentation [27]

Cần áp dụng kiểu segmentation nào thì phụ thuộc vào bài toán. Ví dụ: cần segment người trên

đường cho ô tô tự lái, thì có thể dùng semantic segmentation vì không cần thiết phải phân biệt ai với ai, nhưng nếu cần theo dõi mọi hành vi của mọi người trên đường thì cần instance segmentation thì cần phân biệt mọi người với nhau.

15.1.2 Ứng dụng bài toán segmentation

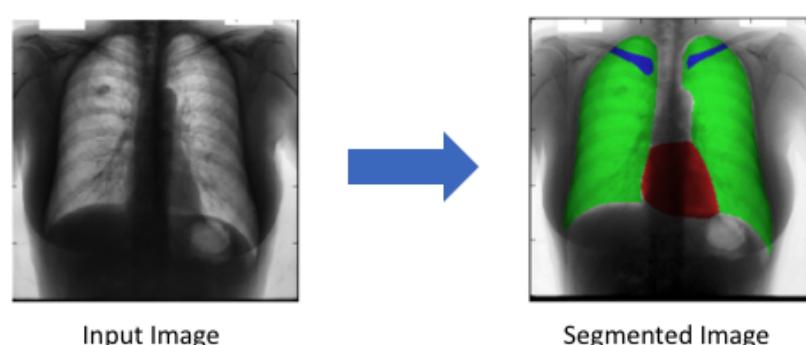
Ô tô tự lái

Segmentation dùng để xác định đường, các xe ô tô, người đi bộ,... để hỗ trợ cho ô tô tự lái



Chẩn đoán trong y học

Segmentation được ứng dụng rất nhiều trong y học để hỗ trợ việc chẩn đoán bệnh. Ví dụ phân tích ảnh X-quang.



Hình 15.4: Ứng dụng segmentation [16]

15.2 Mạng U-Net với bài toán semantic segmentation

Như trong bài xử lý ảnh ta đã biết thì ảnh bản chất là một ma trận của các pixel. Trong bài toán image segmentation, ta cần phân loại mỗi pixel trong ảnh. Ví dụ như trong hình trên với semantic segmentation, với mỗi pixel trong ảnh ta cần xác định xem nó là background hay là người. Thêm nữa là ảnh input và output có cùng kích thước.

U-Net được phát triển bởi Olaf Ronneberger et al. để dùng cho image segmentation trong y học. Kiến trúc có 2 phần đối xứng nhau được gọi là encoder (phần bên trái) và decoder (phần bên phải).

15.2.1 Kiến trúc mạng U-Net

2

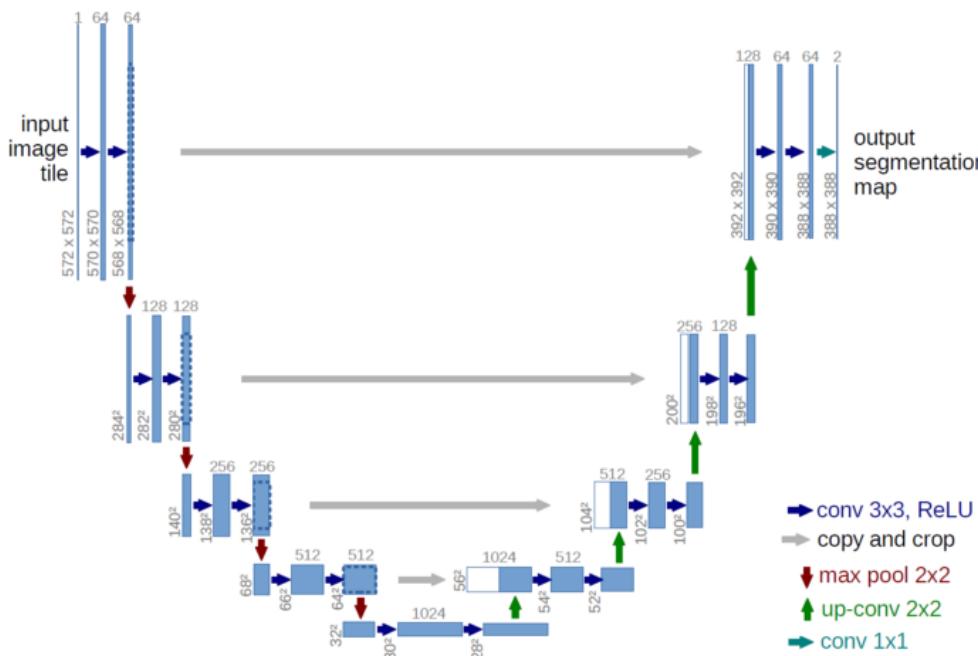


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Hình 15.5: Mạng U-Net [20]

Nhận xét:

- Thực ra phần encoder chỉ là ConvNet bình thường (conv, max pool) với quy tắc quen thuộc từ bài VGG, các layer sau thì width, height giảm nhưng depth tăng.
- Phần decoder có mục đích là khôi phục lại kích thước của ảnh gốc, ta thấy có up-conv là Conv với stride > 1 thì để giảm kích thước của ảnh giống như max pool, thì up-conv dùng để tăng kích thước của ảnh.
- Bạn thấy các đường màu xám, nó nối layer trước với layer hiện tại được dùng rất phổ biến

trong các CNN ngày nay như DenseNet để tránh vanishing gradient cũng như mang được các thông tin cần thiết từ layer trước tới layer sau.

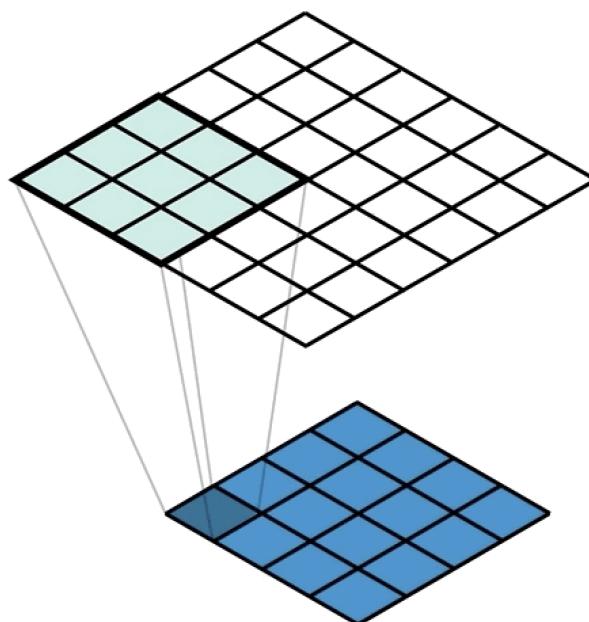
Loss function

Vì bài toán là phân loại cho mỗi pixel nên loss function sẽ là tổng cross-entropy loss cho mỗi pixel trong toàn bộ bức ảnh.

Transposed convolution

Hình ở trên có kích thước là 6*6, hình ở dưới có kích thước là 4*4, kernel có kích thước 3*3.

Nếu ta thực hiện phép tính convolution với input là hình ở trên, padding = 0, stride = 1 và kernel 3*3 thì output sẽ là hình ở dưới.



Phép tính transposed convolution thì sẽ ngược lại, input là hình ở dưới, padding = 0, stride = 1 và kernel 3*3 thì output sẽ là hình ở trên. Các ô vuông ở hình trên bị đè lên nhau thì sẽ được cộng dồn. Các quy tắc về stride và padding thì tương tự với convolution.

Mọi người có thể xem thêm ở [đây](#).

15.2.2 Code

Nhận xét:

- Đây là phần code để tạo model, mô hình encoder và decoder đối xứng nhau hoàn toàn.
- Các conv layer đều dùng 3*3 và padding là same để giữ nguyên kích thước.
- Max pooling kích thước 2*2, mỗi lần pooling width, height giảm 1 nửa, nhưng layer sau đó depth tăng gấp đôi.
- Up sampling kích thước 2*2, mỗi lần pooling width, height tăng gấp đôi, nhưng layer sau đó depth giảm một nửa.
- Concatenate dùng để nối layer đối xứng ở encoder với layer hiện tại có cùng kích thước.
- Dropout cũng được sử dụng để tránh overfitting

- Relu activation được dùng trong các layer trừ output layer dùng sigmoid.
- Như bạn biết sigmoid dùng cho binary classification tuy nhiên nếu bạn output ra nhiều ra giá trị (ví dụ cả 1 ma trận trong bài này) và để loss là binary_crossentropy thì Keras sẽ hiểu loss function là tổng của binary_crossentropy của từng pixel trong ảnh.

```

inputs = Input(input_size)
conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(inputs)
conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(pool1)
conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(pool2)
conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(pool3)
conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv4)
drop4 = Dropout(0.5)(conv4)
pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(pool4)
conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv5)
drop5 = Dropout(0.5)(conv5)

up6 = Conv2D(512, 2, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(drop5))
merge6 = concatenate([drop4,up6], axis = 3)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(merge6)
conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv6)

up7 = Conv2D(256, 2, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv6))
merge7 = concatenate([conv3,up7], axis = 3)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(merge7)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv7)

```

```

up8 = Conv2D(128, 2, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv7))
merge8 = concatenate([conv2,up8], axis = 3)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(merge8)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv8)

up9 = Conv2D(64, 2, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv8))
merge9 = concatenate([conv1,up9], axis = 3)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(merge9)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv9)
conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same', \
kernel_initializer = 'he_normal')(conv9)
conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

model = Model(input = inputs, output = conv10)

model.compile(optimizer = Adam(lr = 1e-4), loss = 'binary_crossentropy', \
metrics = ['accuracy'])

```

Ngoài ra thì ảnh cũng được scale về [0,1] và mask cũng đưa về 0 và 1 với bài toán segmentation chỉ gồm 1 lớp và background

```

img = img / 255
mask = mask /255
mask[mask > 0.5] = 1
mask[mask <= 0.5] = 0

```

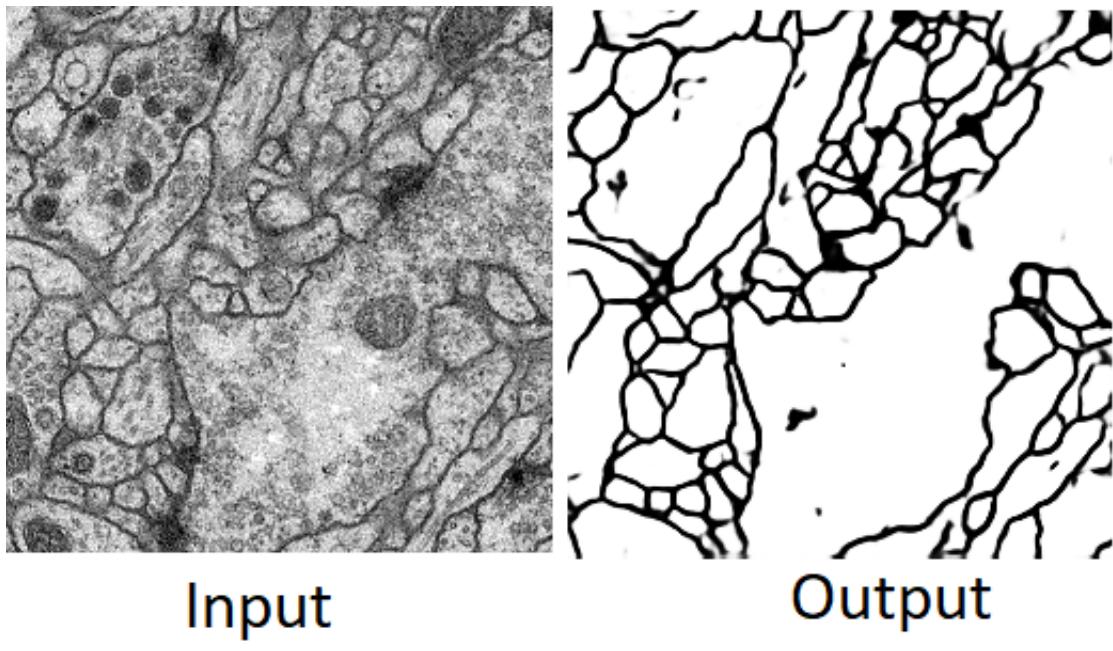
Bên cạnh đó thì data augmentation cũng được sử dụng để tăng lượng ảnh cho training set

```

data_gen_args = dict(rotation_range=0.2,
                     width_shift_range=0.05,
                     height_shift_range=0.05,
                     shear_range=0.05,
                     zoom_range=0.05,
                     horizontal_flip=True,
                     fill_mode='nearest')

```

Kết quả sau khi train model



15.3 Bài tập

1. Đọc và hiểu code U-Net ở link trên github.
2. Tự tìm bộ dataset mới về segmentation ảnh và chạy trên code U-Net.



Recurrent Neural Network

16	Recurrent neural network	227
16.1	Recurrent Neural Network là gì?	
16.2	Mô hình bài toán RNN	
16.3	Bài tập	
17	Long short term memory (LSTM)	233
17.1	Giới thiệu về LSTM	
17.2	Mô hình LSTM	
17.3	LSTM chống vanishing gradient	
17.4	Bài tập	
18	Ứng dụng thêm mô tả cho ảnh	237
18.1	Ứng dụng	
18.2	Dataset	
18.3	Phân tích bài toán	
18.4	Các bước chi tiết	
18.5	Python code	
19	Seq2seq và attention	257
19.1	Giới thiệu	
19.2	Mô hình seq2seq	
19.3	Cơ chế attention	

16. Recurrent neural network

Deep learning có 2 mô hình lớn là Convolutional Neural Network (CNN) cho bài toán có input là ảnh và Recurrent neural network (RNN) cho bài toán dữ liệu dạng chuỗi (sequence). Tôi đã giới thiệu về Convolutional Neural Network (CNN) và các ứng dụng của deep learning trong computer vision bao gồm: classification, object detection, segmentation. Có thể nói là tương đối đầy đủ các dạng bài toán liên quan đến CNN. Bài này tôi sẽ giới thiệu về RNN.

16.1 Recurrent Neural Network là gì?

Bài toán: Cần phân loại hành động của người trong video, input là video 30s, output là phân loại hành động, ví dụ: đứng, ngồi, chạy, đánh nhau, bắn súng,...

Khi xử lý video ta hay gặp khái niệm FPS (frame per second) tức là bao nhiêu frame (ảnh) mỗi giây. Ví dụ 1 FPS với video 30s tức là lấy ra từ video 30 ảnh, mỗi giây một ảnh để xử lý.

Ta dùng 1 FPS cho video input ở bài toán trên, tức là lấy ra 30 ảnh từ video, ảnh 1 ở giây 1, ảnh 2 ở giây 2,... ảnh 30 ở giây 30. Bây giờ input là 30 ảnh: ảnh 1, ảnh 2,... ảnh 30 và output là phân loại hành động. Nhận xét:

- Các ảnh có thứ tự: ảnh 1 xảy ra trước ảnh 2, ảnh 2 xảy ra trước ảnh 3,... Nếu ta đảo lộn các ảnh thì có thể thay đổi nội dung của video. Ví dụ: nội dung video là cảnh bắn nhau, thứ tự đúng là A bắn trúng người B và B chết, nếu ta đảo thứ tự ảnh thành người B chết xong A mới bắn thì rõ ràng bây giờ A không phải là kẻ giết người => nội dung video bị thay đổi.
- Ta có thể dùng CNN để phân loại 1 ảnh trong 30 ảnh trên, nhưng rõ ràng là 1 ảnh không thể mô tả được nội dung của cả video. Ví dụ: Cảnh người cướp điện thoại, nếu ta chỉ dùng 1 ảnh là người đấy cầm điện thoại lúc cướp xong thì ta không thể biết được cả hành động cướp.

=> Cần một mô hình mới có thể giải quyết được bài toán với input là sequence (chuỗi ảnh 1->30)
=> Recurrent Neural Network (RNN) ra đời.

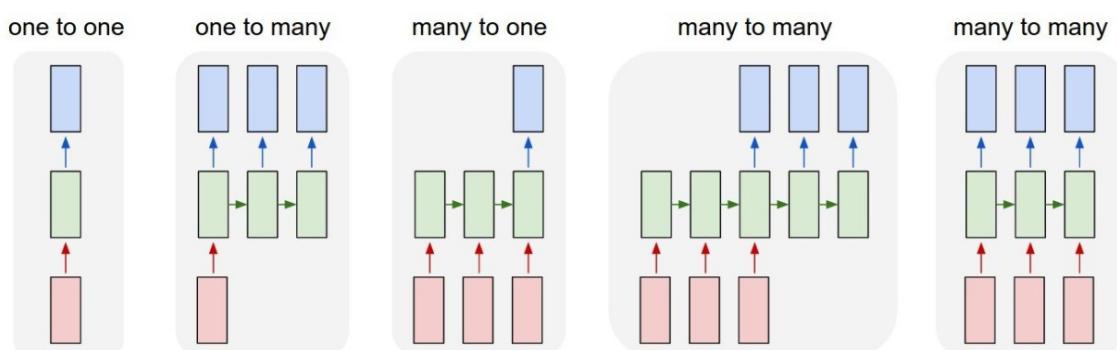
16.1.1 Dữ liệu dạng sequence

Dữ liệu có thứ tự như các ảnh tách từ video ở trên được gọi là sequence, time-series data.

Trong bài toán dự đoán đột quy tim cho bệnh nhân bằng các dữ liệu tim mạch khám trước đó. Input là dữ liệu của những lần khám trước đó, ví dụ $i1$ là lần khám tháng 1, $i2$ là lần khám tháng 2,..., $i8$ là lần khám tháng 8. ($i1, i2, \dots, i8$) được gọi là sequence data. RNN sẽ học từ input và dự đoán xem bệnh nhân có bị đột quy tim hay không.

Ví dụ khác là trong bài toán dịch tự động với input là 1 câu, ví dụ "tôi yêu Việt Nam" thì vị trí các từ và sự xếp xắp cực kì quan trọng đến nghĩa của câu và dữ liệu input các từ ['tôi', 'yêu', 'việt', 'nam'] được gọi là sequence data. **Trong bài toán xử lý ngôn ngữ (NLP) thì không thể xử lý cả câu được và người ta tách ra từng từ (chữ) làm input, giống như trong video người ta tách ra các ảnh (frame) làm input.**

16.1.2 Phân loại bài toán RNN



Hình 16.1: Các dạng bài toán RNN

One to one: mẫu bài toán cho Neural Network (NN) và Convolutional Neural Network (CNN), 1 input và 1 output, ví dụ với bài toán phân loại ảnh MNIST input là ảnh và output là ảnh đầy là số nào.

One to many: bài toán có 1 input nhưng nhiều output, ví dụ với bài toán caption cho ảnh, input là 1 ảnh nhưng output là nhiều chữ mô tả cho ảnh đầy, dưới dạng một câu.



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."

Hình 16.2: Ví dụ image captioning [10]

Many to one: bài toán có nhiều input nhưng chỉ có 1 output, ví dụ bài toán phân loại hành động trong video, input là nhiều ảnh (frame) tách ra từ video, output là hành động trong video

Many to many: bài toán có nhiều input và nhiều output, ví dụ bài toán dịch từ tiếng anh sang tiếng việt, input là 1 câu gồm nhiều chữ: "I love Vietnam" và output cũng là 1 câu gồm nhiều chữ "Tôi yêu Việt Nam". Để ý là độ dài sequence của input và output có thể khác nhau.

16.1.3 Ứng dụng bài toán RNN

Về cơ bản nếu bạn thấy sequence data hay time-series data và bạn muốn áp dụng deep learning thì bạn nghĩ ngay đến RNN. Dưới đây là một số ứng dụng của RNN:

- **Speech to text:** Chuyển giọng nói sang text.
- **Sentiment classification:** Phân loại bình luận của người dùng, tích cực hay tiêu cực.
- **Machine translation:** Bài toán dịch tự động giữa các ngôn ngữ.
- **Video recognition:** Nhận diện hành động trong video.
- **Heart attack:** Dự đoán đột quỵ tim.

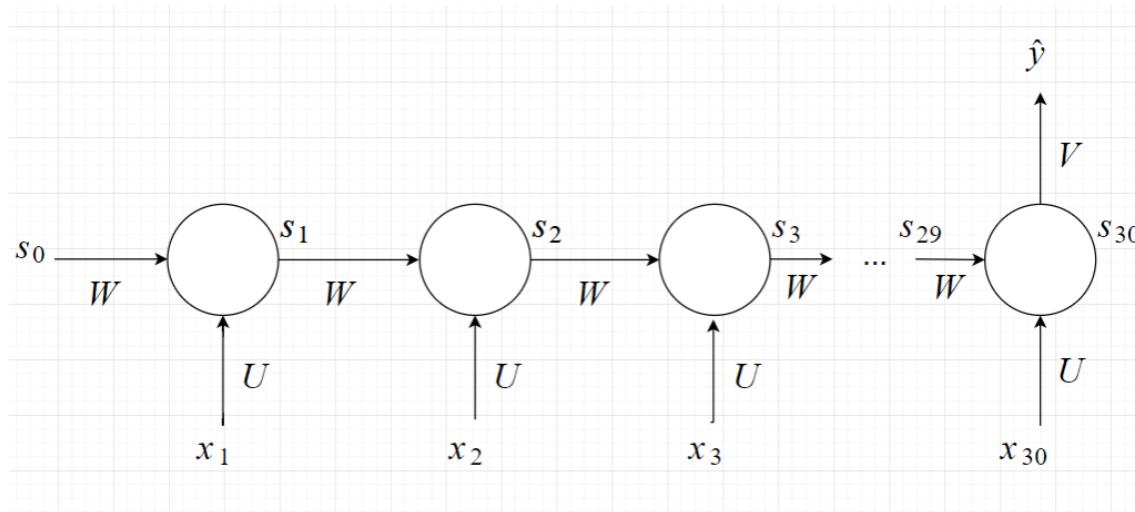
16.2 Mô hình bài toán RNN

16.2.1 Mô hình RNN

Bài toán: Nhận diện hành động trong video 30s. Đây là dạng bài toán many to one trong RNN, tức nhiều input và 1 output.

Input ta sẽ tách video thành 30 ảnh (mỗi giây một ảnh). Các ảnh sẽ được cho qua pretrained model CNN để lấy ra các feature (feature extraction) vector có kích thước $n \times 1$. Vector tương ứng với ảnh ở giây thứ i là x_i .

Output là vector có kích thước $d \times 1$ (d là số lượng hành động cần phân loại), softmax function được sử dụng như trong bài phân loại ảnh.



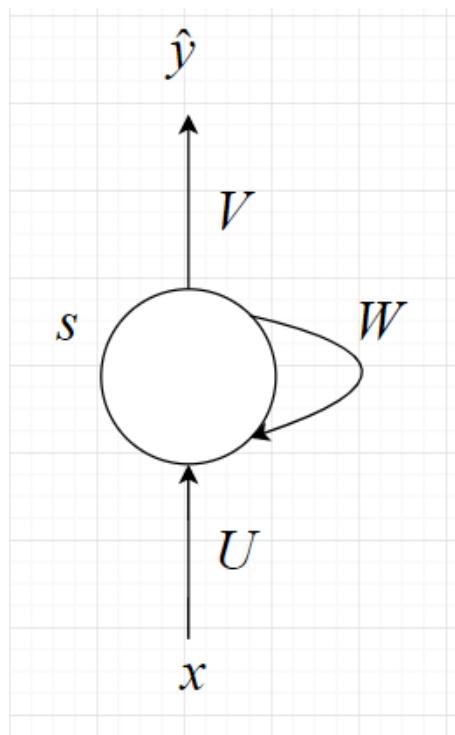
Hình 16.3: Mô hình RNN cho bài toán.

Ta có:

- Mô hình có 30 input và 1 output, các input được cho vào model đúng với thứ tự ảnh trong video x_1, x_2, \dots, x_{30} .

- Mỗi hình tròn được gọi là 1 state, state t có input là x_t và s_{t-1} (output của state trước); output là $s_t = f(U * x_t + W * s_{t-1})$. f là activation function thường là Tanh hoặc ReLU.
- Có thể thấy s_t mang cả thông tin từ state trước (s_{t-1}) và input của state hiện tại $\Rightarrow s_t$ giống như memory nhớ các đặc điểm của các input từ x_1 đến x_t .
- s_0 được thêm vào chỉ cho chuẩn công thức nên thường được gán bằng 0 hoặc giá trị ngẫu nhiên. Có thể hiểu là ban đầu chưa có dữ liệu gì để học thì memory rỗng.
- Do ta chỉ có 1 output, nên sẽ được đặt ở state cuối cùng, khi đó s_{30} học được thông tin từ tất cả các input. $\hat{y} = g(V * s_{30})$. g là activation function, trong bài này là bài toán phân loại nên sẽ dùng softmax.

Ta thấy là ở mỗi state các hệ số W, U là giống nhau nên model có thể được viết lại thành:



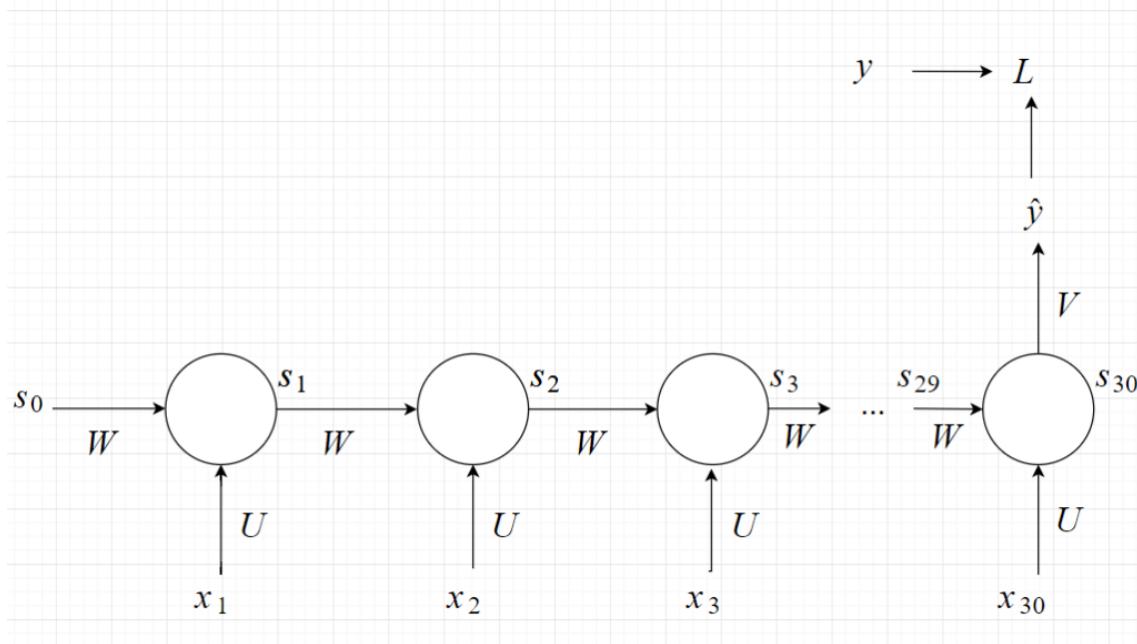
Hình 16.4: Mô hình RNN rút gọn

Tóm lại:

- x_i là vector có kích thước $n*1$, s_i là vector có kích thước $m*1$, y_i là vector có kích thước $d*1$.
U là ma trận có kích thước $m*n$, W là ma trận có kích thước $m*m$ và V là ma trận có kích thước $d*m$.
- $s_0 = 0, s_t = f(U * x_t + W * s_{t-1})$ với $t \geq 1$
- $\hat{y} = g(V * s_{30})$

16.2.2 Loss function

Loss function của cả mô hình bằng tổng loss của mỗi output, tuy nhiên ở mô hình trên chỉ có 1 output và là bài toán phân loại nên categorical cross entropy loss sẽ được sử dụng.



Hình 16.5: Loss function

16.2.3 Backpropagation Through Time (BPTT)

Có 3 tham số ta cần phải tìm là W , U , V . Để thực hiện gradient descent, ta cần tính: $\frac{\partial L}{\partial U}$, $\frac{\partial L}{\partial V}$, $\frac{\partial L}{\partial W}$.

Tính đạo hàm với V thì khá đơn giản:

$$\frac{\partial L}{\partial V} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial V}$$

Tuy nhiên với U , W thì lại khác.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial s_{30}} * \frac{\partial s_{30}}{\partial W}$$

Do $s_{30} = f(W * s_{29} + U * x_{30})$ có s_{29} phụ thuộc vào W . Nên áp dụng công thức hồi cấp 3 bạn học: $(f(x) * g(x))' = f'(x) * g(x) + f(x) * g'(x)$. Ta có

$$\frac{\partial s_{30}}{\partial W} = \frac{\partial s'_{30}}{\partial W} + \frac{\partial s_{30}}{\partial s_{29}} * \frac{\partial s_{29}}{\partial W}, \text{ trong đó } \frac{\partial s'_{30}}{\partial W} \text{ là đạo hàm của } s_{30} \text{ với } W \text{ khi coi } s_{29} \text{ là constant với } W.$$

Tương tự trong biểu thức s_{29} có s_{28} phụ thuộc vào W , s_{28} có s_{27} phụ thuộc vào W ... nên áp dụng công thức trên và chain rule:

$$\frac{\partial L}{\partial W} = \sum_{i=0}^{30} \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial s_{30}} * \frac{\partial s_{30}}{\partial s_i} * \frac{\partial s'_i}{\partial W}, \text{ trong đó } \frac{\partial s_{30}}{\partial s_i} = \prod_{j=i}^{29} \frac{\partial s_{j+1}}{\partial s_j} \text{ và } \frac{\partial s'_i}{\partial W} \text{ là đạo hàm của } s_i \text{ với } W \text{ khi coi } s_{i-1} \text{ là constant với } W.$$

Nhìn vào công thức tính đạo hàm của L với W ở trên ta có thể thấy hiện tượng vanishing gradient ở các state đầu nên ta cần mô hình tốt hơn để giảm hiện tượng vanishing gradient => Long short term memory (LSTM) ra đời và sẽ được giới thiệu ở bài sau. Vì trong bài toán thực tế liên quan đến

time-series data thì LSTM được sử dụng phổ biến hơn là mô hình RNN thuần nên bài này không có code, bài sau sẽ có code ứng dụng với LSTM.

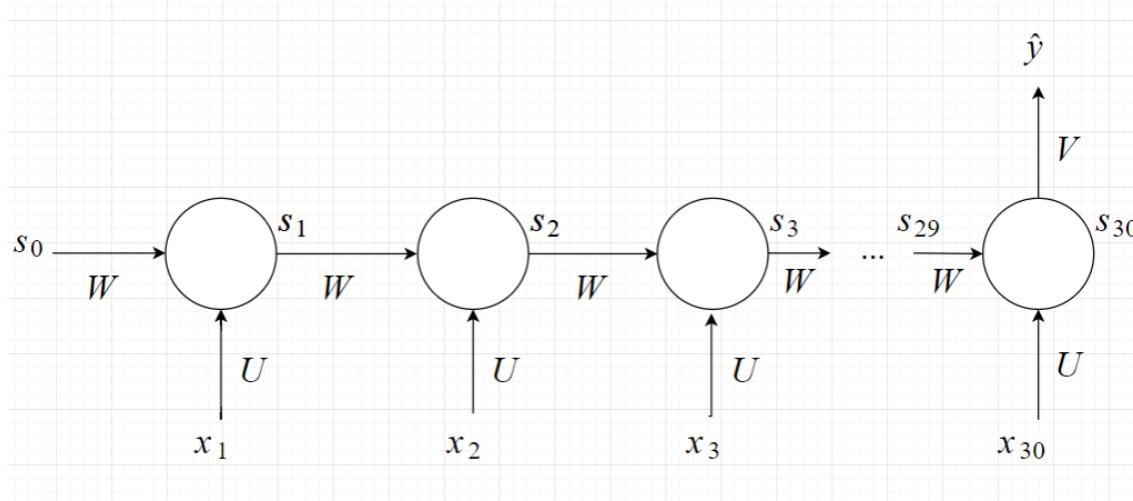
16.3 Bài tập

1. Hệ số trong RNN là gì?
2. Thiết kế và train model RNN dự báo giá Bitcoin, tải dữ liệu ở [đây](#).
3. Tự tìm hiểu và sử dụng mô hình Bidirectional cho bài toán trên.

17. Long short term memory (LSTM)

17.1 Giới thiệu về LSTM

Bài trước tôi đã giới thiệu về recurrent neural network (RNN). RNN có thể xử lý thông tin dạng chuỗi (sequence/ time-series). Như ở bài dự đoán hành động trong video ở bài trước, RNN có thể mang thông tin của frame (ảnh) từ state trước tới các state sau, rồi ở state cuối là sự kết hợp của tất cả các ảnh để dự đoán hành động trong video.



Hình 17.1: Mô hình RNN

Đạo hàm của L với W ở state thứ i: $\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial s_{30}} * \frac{\partial s_{30}}{\partial s_i} * \frac{\partial s'_i}{\partial W}$, trong đó $\frac{\partial s_{30}}{\partial s_i} = \prod_{j=i}^{29} \frac{\partial s_{j+1}}{\partial s_j}$

Giả sử activation là tanh function, $s_t = \tanh(U * x_t + W * s_{t-1})$

$$\frac{\partial s_t}{\partial s_{t-1}} = (1 - s_t^2) * W \Rightarrow \frac{\partial s_{30}}{\partial s_i} = W^{30-i} * \prod_{j=i}^{29} (1 - s_j^2).$$

Ta có $s_j < 1, W < 1 \Rightarrow$ Ở những state xa thì $\frac{\partial s_{30}}{\partial s_i} \approx 0$ hay $\frac{\partial L}{\partial W} \approx 0$, hiện tượng vanishing gradient

Ta có thể thấy là các state càng xa ở trước đó thì càng bị vanishing gradient và các hệ số không được update với các frame ở xa. Hay nói cách khác là RNN không học được từ các thông tin ở trước đó xa do vanishing gradient.

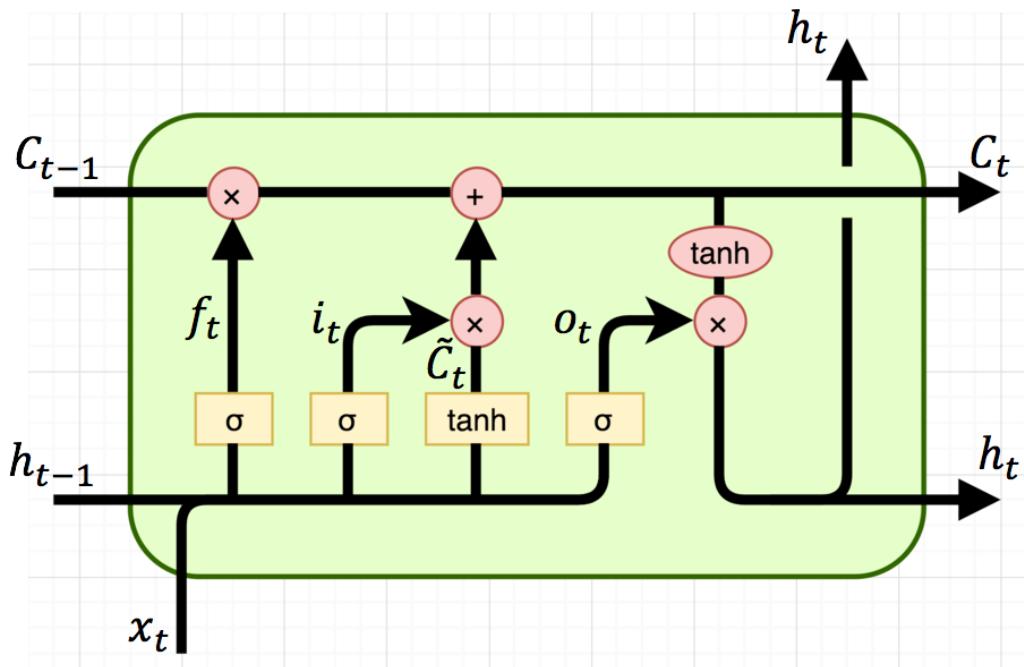
Như vậy về lý thuyết là RNN có thể mang thông tin từ các layer trước đến các layer sau, nhưng thực tế là thông tin chỉ mang được qua một số lượng state nhất định, sau đó thì sẽ bị vanishing gradient, hay nói cách khác là model chỉ học được từ các state gần nó \Rightarrow short term memory.

Cùng thử lấy ví dụ về short term memory nhé. Bài toán là dự đoán từ tiếp theo trong đoạn văn. Đoạn đầu tiên "Mặt trời mọc ở hướng ...", ta có thể chỉ sử dụng các từ trước trong câu để đoán là đông. Tuy nhiên, với đoạn, "Tôi là người Việt Nam. Tôi đang sống ở nước ngoài. Tôi có thể nói trôi chảy tiếng ..." thì rõ ràng là chỉ sử dụng từ trong câu đấy hoặc câu trước là không thể dự đoán được từ cần diễn là Việt. Ta cần các thông tin từ state ở trước đó rất xa \Rightarrow cần long term memory điều mà RNN không làm được \Rightarrow Cần một mô hình mới để giải quyết vấn đề này \Rightarrow Long short term memory (LSTM) ra đời.

17.2 Mô hình LSTM

Ở state thứ t của mô hình LSTM:

- Output: c_t, h_t , ta gọi c là cell state, h là hidden state.
- Input: c_{t-1}, h_{t-1}, x_t . Trong đó x_t là input ở state thứ t của model. c_{t-1}, h_{t-1} là output của layer trước. h đóng vai trò khá giống như s ở RNN, trong khi c là điểm mới của LSTM.



Hình 17.2: Mô hình LSTM [25]

Cách đọc biểu đồ trên: bạn nhìn thấy kí hiệu σ , tanh ý là bước đẩy dùng sigma, tanh activation function. Phép nhân ở đây là element-wise multiplication, phép cộng là cộng ma trận.

f_t, i_t, o_t tương ứng với forget gate, input gate và output gate.

- Forget gate: $f_t = \sigma(U_f * x_t + W_f * h_{t-1} + b_f)$
- Input gate: $i_t = \sigma(U_i * x_t + W_i * h_{t-1} + b_i)$
- Output gate: $o_t = \sigma(U_o * x_t + W_o * h_{t-1} + b_o)$

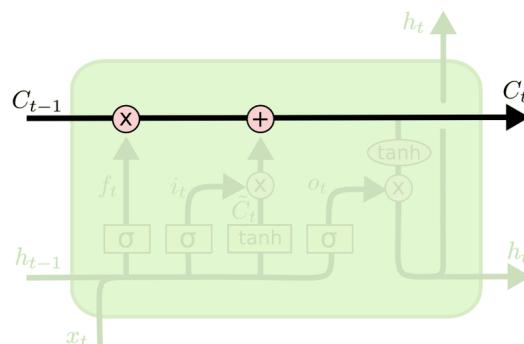
Nhận xét: $0 < f_t, i_t, o_t < 1$; b_f, b_i, b_o là các hệ số bias; hệ số W, U giống như trong bài RNN.

$\tilde{c}_t = \tanh(U_c * x_t + W_c * h_{t-1} + b_c)$, bước này giống hệt như tính s_t trong RNN.

$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$, **forget gate** quyết định xem cần lấy bao nhiêu từ cell state trước và **input gate** sẽ quyết định lấy bao nhiêu từ input của state và hidden layer của layer trước.

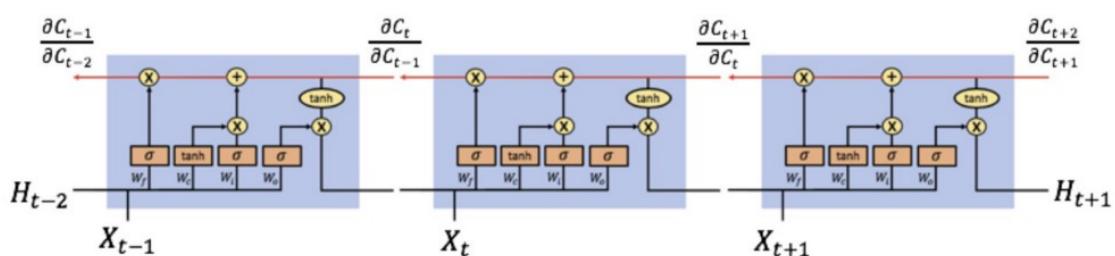
$h_t = o_t * \tanh(c_t)$, **output gate** quyết định xem cần lấy bao nhiêu từ cell state để trở thành output của hidden state. Ngoài ra h_t cũng được dùng để tính ra output y_t cho state t.

Nhận xét: h_t, \tilde{c}_t khá giống với RNN, nên model có short term memory. Trong khi đó c_t giống như một băng chuyền ở trên mô hình RNN vậy, thông tin nào cần quan trọng và dùng ở sau sẽ được gửi vào và dùng khi cần => có thể mang thông tin từ đi xa => long term memory. Do đó mô hình LSTM có cả short term memory và long term memory.



Hình 17.3: cell state trong LSTM

17.3 LSTM chống vanishing gradient



Hình 17.4: Mô hình LSTM [9]

Ta cũng áp dụng thuật toán back propagation through time cho LSTM tương tự như RNN.

Thành phần chính gây ra vanishing gradient trong RNN là $\frac{\partial s_{t+1}}{\partial s_t} = (1 - s_t^2) * W$, trong đó $s_t, W < 1$.

Tương tự trong LSTM ta quan tâm đến $\frac{\partial c_t}{\partial c_{t-1}} = f_t$. Do $0 < f_t < 1$ nên về cơ bản thì LSTM vẫn bị vanishing gradient nhưng bị ít hơn so với RNN. Hơn thế nữa, khi mang thông tin trên cell state thì ít khi cần phải quên giá trị cell cũ, nên $f_t \approx 1 \Rightarrow$ Tránh được vanishing gradient.

Do đó LSTM được dùng phổ biến hơn RNN cho các toán thông tin dạng chuỗi. Bài sau tôi sẽ giới thiệu về ứng dụng LSTM cho image captioning.

17.4 Bài tập

1. Mô hình LSTM tốt hơn mô hình RNN ở điểm nào?
2. Dùng mô hình LSTM cho bài toán dự đoán bitcoin ở bài RNN.

18. Ứng dụng thêm mô tả cho ảnh

Ở những bài trước tôi đã giới thiệu về mô hình Recurrent Neural Network (RNN) cho bài toán dữ liệu dạng chuỗi. Tuy nhiên RNN chỉ có short term memory và bị vanishing gradient. Tiếp đó tôi đã giới thiệu về Long short term memory (LSTM) có cả short term memory và long term memory, hơn thế nữa tránh được vanishing gradient. Bài này tôi sẽ viết về ứng dụng của LSTM cho ứng dụng image captioning.

18.1 Ứng dụng



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."

Hình 18.1: Ví dụ image captioning [10]

Ta có thể thấy ngay 2 ứng dụng của image captioning:

- Để giúp những người già mắt kém hoặc người mù có thể biết được cảnh vật xung quanh hay hỗ trợ việc di chuyển. Quy trình sẽ là: Image -> text -> voice.
- Giúp google search có thể tìm kiếm được hình ảnh dựa vào caption.

18.2 Dataset

Dữ liệu dùng trong bài này là Flickr8k Dataset. Mọi người tải ở [đây](#). Dữ liệu gồm 8000 ảnh, 6000 ảnh cho training set, 1000 cho dev set (validation set) và 1000 ảnh cho test set.

Bạn tải về có 2 folder: Flicker8k_Dataset và Flicker8k_Text. Flicker8k_Dataset chứa các ảnh với tên là các id khác nhau. Flicker8k_Text chứa:

- Flickr_8k.testImages, Flickr_8k.devImages, Flickr_8k.trainImages, Flickr_8k.devImages chứa id các ảnh dùng cho việc test, train, validation.
- Flickr8k.token chứa các caption của ảnh, mỗi ảnh chứa 5 captions.

Ví dụ ảnh ở hình 18.2 có 5 captions:

- A child in a pink dress is climbing up a set of stairs in an entry way.
- A girl going into a wooden building.
- A little girl climbing into a wooden playhouse.
- A little girl climbing the stairs to her playhouse.
- A little girl in a pink dress going into a wooden cabin.

Thực ra 1 ảnh nhiều caption cũng hợp lý vì bức ảnh có thể được mô tả theo nhiều cách khác nhau. Một ảnh 5 caption sẽ cho ra 5 training set khác nhau: (ảnh, caption 1), (ảnh, caption 2), (ảnh, caption 3), (ảnh, caption 4), (ảnh, caption 5). Như vậy training set sẽ có $6000 * 5 = 40000$ dataset.

18.3 Phân tích bài toán

Input là ảnh và output là text, ví dụ "man in black shirt is playing guitar".

Nhìn chung các mô hình machine learning hay deep learning đều không xử lý trực tiếp với text như 'man', 'in', 'black',... mà thường phải quy đổi (encode) về dạng số. Từng từ sẽ được encode sang dạng vector với độ dài số định, phương pháp đây gọi là word embedding.

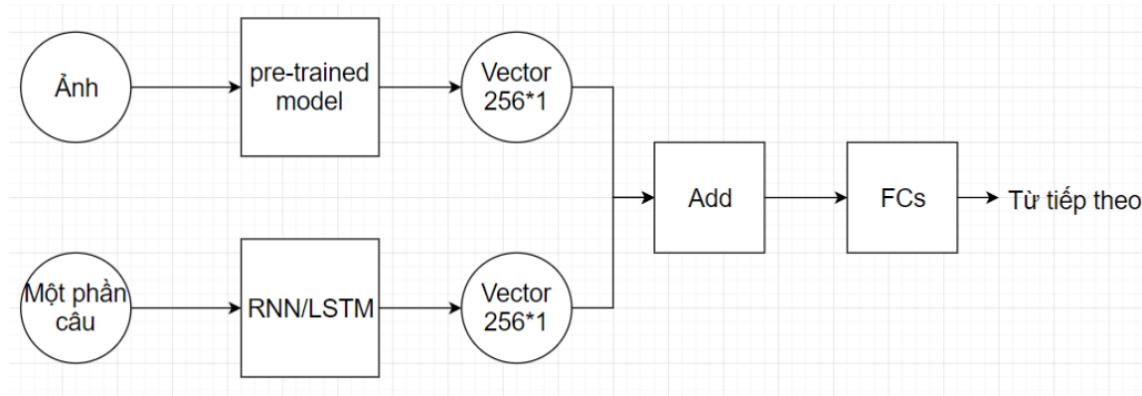
Nhìn thấy output là text nghĩ ngay đến RNN và sử dụng mô hình LSTM.

Input là ảnh thường được extract feature qua pre-trained model với dataset lớn như ImageNet và model phổ biến như VGG16, ResNet, quá trình được gọi là embedding và output là 1 vector.

Ý tưởng sẽ là dùng embedding của ảnh và dùng các từ phía trước để dự đoán từ tiếp theo trong caption.

Ví dụ:

- Embedding vector + A -> girl
- Embedding vector + A girl -> going
- Embedding vector + A girl going -> into
- Embedding vector + A girl going into -> a.
- Embedding vector + A girl going into a -> wooden building .
- Embedding vector + A girl going into a wooden -> building .



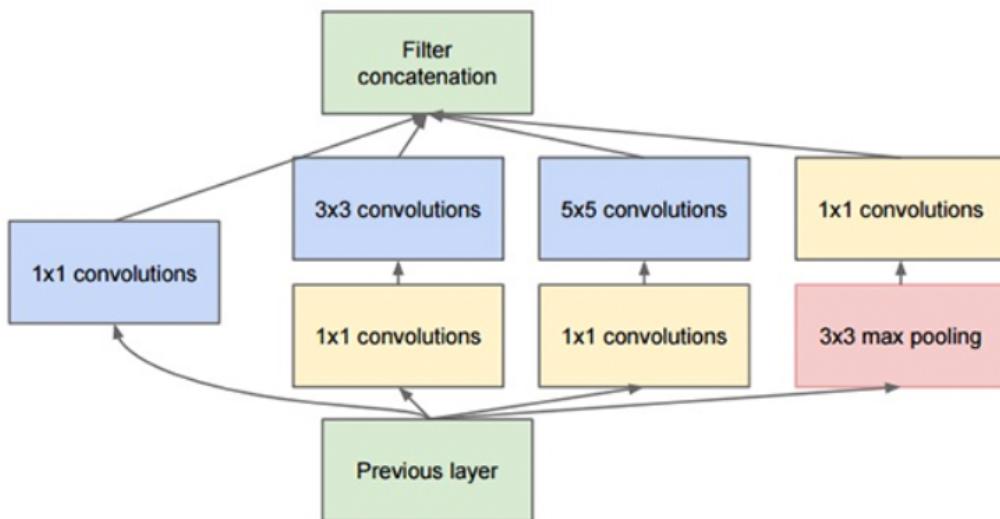
Hình 18.3: Mô hình của bài toán

Để dự đoán từ tiếp theo ta sẽ xây dựng từ điển các từ xuất hiện trong training set (ví dụ 2000 từ) và bài toán trở thành bài toán phân loại từ, xem từ tiếp theo là từ nào, khá giống như bài phân loại ảnh.

18.4 Các bước chi tiết

18.4.1 Image embedding với Inception

Có lẽ cái tên GoogLeNet sẽ quen thuộc hơn và gấp nhiều hơn so với Inception, GoogLeNet là version 1 của Inception, hiện giờ mô hình phổ biến là Inception v3.



Hình 18.4: Mô hình Googlenet, Going Deeper with Convolutions, Szegedy et al

Thay vì trong mỗi Conv layer chỉ dùng 1 kernel size nhất định như 3×3 , 5×5 , thì giờ ở một layer có nhiều kernel size khác nhau, do đó mô hình có thể học được nhiều thuộc tính khác nhau của ảnh trong mỗi layer.

Cụ thể hơn mọi người xem thêm ở [đây](#).

Ta sẽ sử dụng pre-trained model Inception v3 với dataset Imagenet. Do là pre-trained model

yêu cầu ảnh đầu vào là 229*229 nên ra sẽ resize ảnh về kích thước này. Sau khi qua pre-trained model ta sẽ lấy được embedding vector của ảnh, kích thước 256*1

18.4.2 Text preprocessing

Ta xử lý text qua một số bước cơ bản.

- Chuyển chữ hoa thành chữ thường, "Hello" -> "hello"
- Bỏ các kí tự đặc biệt như "
- Loại bỏ các chữ có số như hey199

Sau đó ta sẽ thêm 2 từ "startseq" và "endseq" để biểu thị sự bắt đầu và kết thúc của caption. Ví dụ: "startseq a girl going into a wooden building endseq". "endseq" dùng khi test ảnh thì biết kết thúc của caption.

Ta thấy có 8763 chữ khác nhau trong số 40000 caption. Tuy nhiên ta không quan tâm lắm những từ mà chỉ xuất hiện 1 vài lần, vì nó giống như là nhiễu vậy và không tốt cho việc học và dự đoán từ của model, nên ta chỉ giữ lại những từ mà xuất hiện trên 10 lần trong số tất cả các caption. Sau khi bỏ những từ xuất hiện ít hơn 10 lần ta còn 1651 từ.

Tuy nhiên do độ dài các sequence khác nhau, ví dụ: "A", " A girl going", " A girl going into a wooden", nên ta cần padding thêm để các chuỗi có cùng độ dài bằng với độ dài của chuỗi dài nhất là 34. Do đó số tổng số từ (từ điển) ta có là 1651 + 1 (từ dùng để padding).

18.4.3 Word embedding

Để có thể đưa text vào mô hình deep learning, việc đầu tiên chúng ta cần làm là số hóa các từ đầu vào (embedding). Ở phần này chúng ta sẽ thảo luận về các mô hình nhúng từ (word embedding) và sự ra đời của mô hình word2vec rất nổi tiếng được google giới thiệu vào năm 2013.

Các phương pháp trước đây

One hot encoding

Phương pháp này là phương pháp đơn giản nhất để đưa từ về dạng số hóa vector với chiều bằng với kích thước bộ từ điển. Mỗi từ sẽ được biểu diễn bởi 1 vector mà giá trị tại vị trí của từ đó trong từ điển bằng 1 và giá trị tại các vị trí còn lại đều bằng 0.

Ví dụ: Ta có 3 câu đầu vào: "Tôi đang đi học", "Mình đang bận nhé", "Tôi sẽ gọi lại sau". Xây dựng bộ từ điển: "Tôi, đang, đi, học, Mình, bận, nhé, sẽ, gọi, lại, sau". Ta có các biểu diễn one hot encoding của từng từ như sau:

Tôi: [1,0,0,0,0,0,0,0,0,0],
 đang: [0,1,0,0,0,0,0,0,0,0],
 ...
 Minh: [0,0,0,0,1,0,0,0,0,0],
 ...
 sau: [0,0,0,0,0,0,0,0,0,1].

Cách biểu diễn này rất đơn giản, tuy nhiên ta có thể nhận thấy ngay các hạn chế của phương pháp này. Trước hết, one hot encoding không thể hiện được thông tin về ngữ nghĩa của từ, ví dụ như khoảng cách(vector(Tôi) - vector(Mình)) = khoảng cách(vector(Tôi) - vector(đang)), trong khi rõ ràng từ "Tôi" và từ "Mình" trong ngữ cảnh như trên có ý nghĩa rất giống nhau còn từ "Tôi" và từ "đang" lại khác nhau hoàn toàn. Tiếp nữa, mỗi từ đều được biểu diễn bằng một vector có độ dài bằng kích thước bộ từ điển, như bộ từ điển của google gồm 13 triệu từ, thì mỗi one hot vector sẽ dài 13 triệu chiều. Cách biểu diễn này tốn rất nhiều tài nguyên nhưng thông tin biểu diễn được lại rất

hạn hẹp.

=> Cần một cách biểu diễn từ ít chiều hơn và mang nhiều thông tin hơn.

Co-occurrence Matrix

Năm 1957, nhà ngôn ngữ học J.R. Firth phát biểu rằng: "Bạn sẽ biết nghĩa của một từ nhờ những từ đi kèm với nó.". Điều này cũng khá dễ hiểu. Ví dụ nhắc đến Việt Nam, người ta thường có các cụm từ quen thuộc như "Chiến tranh Việt Nam", "Cafe Việt Nam", "Việt Nam rồng vàng biển bạc", dựa vào những từ xung quanh ta có thể hiểu hoặc mường tượng ra được "Việt Nam" là gì, như thế nào. Co-occurrence Matrix được xây dựng dựa trên nhận xét trên, co-occurrence đảm bảo quan hệ ngữ nghĩa giữa các từ, dựa trên số lần xuất hiện của các cặp từ trong "context window". Một context window được xác định dựa trên kích thước và hướng của nó, ví dụ của context window:

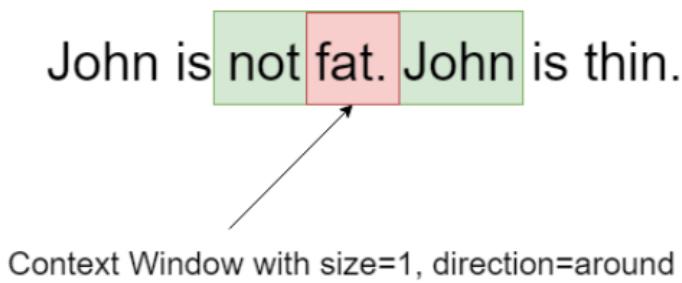


Figure 1-Ví dụ về Context Window

Hình 18.5: Ví dụ về context window

Co-occurrence matrix là một ma trận vuông đối xứng, mỗi hàng, mỗi cột sẽ làm vector đại diện cho từ tương ứng. Từ ví dụ trên ta tiếp tục xây dựng co-occurrence matrix:

	John	is	not	fat	thin
John	0	2	0	1	0
is	2	0	1	0	1
not	0	1	0	1	0
fat	1	0	1	0	0
thin	0	1	0	0	0

Figure 2-Ví dụ về Co-occurrence Matrix

Hình 18.6: Ví dụ về co-occurrence matrix

Trong đó, giá trị tại ô $[i, j]$ là số lần xuất hiện của từ i nằm trong context window của từ j .

Cách biểu diễn trên mặc dù đã giữ được thông tin về ngữ nghĩa của một từ, tuy vẫn còn các hạn chế như sau:

- Khi kích thước bộ từ điển tăng, chiều vector cũng tăng theo.
- Lưu trữ co-occurrence matrix cần rất nhiều tài nguyên về bộ nhớ.

- Các mô hình phân lớp bị gấp vần đề với biểu diễn thưa (có rất nhiều giá trị 0 trong ma trận).

Để làm giảm kích thước của co-occurrence matrix người ta thường sử dụng phép SVD (Singular Value Decomposition) để giảm chiều ma trận. Ma trận thu được sau SVD có chiều nhỏ hơn, dễ lưu trữ hơn và ý nghĩa của từ cũng cô đọng hơn. Tuy nhiên, SVD có độ phức tạp tính toán cao, tăng nhanh cùng với chiều của ma trận ($O(mn^2)$) với m là chiều của ma trận trước SVD, n là chiều của ma trận sau SVD và $n < m$), ngoài ra phương pháp này cũng gặp khó khăn khi thêm các từ vựng mới vào bộ từ điển.

=> Cần phương pháp khác lưu trữ được nhiều thông tin và vector biểu diễn nhỏ.

Word to vec (Word2vec)

Với tư tưởng rằng ngữ cảnh và ý nghĩa của một từ có sự tương quan mật thiết đến nhau, năm 2013 nhóm của Mikolov đề xuất một phương pháp mang tên Word2vec.

Ý tưởng chính của Word2vec

- Thay thế việc lưu thông tin số lần xuất hiện của các từ trong context window như co-occurrence matrix, word2vec học cách dự đoán các từ lân cận.
- Tính toán nhanh hơn và có thể transfer learning khi thêm các từ mới vào bộ từ điển.

Phương pháp:

Với mỗi từ t trong bộ từ điển ta dự đoán các từ lân cận trong bán kính m của nó.

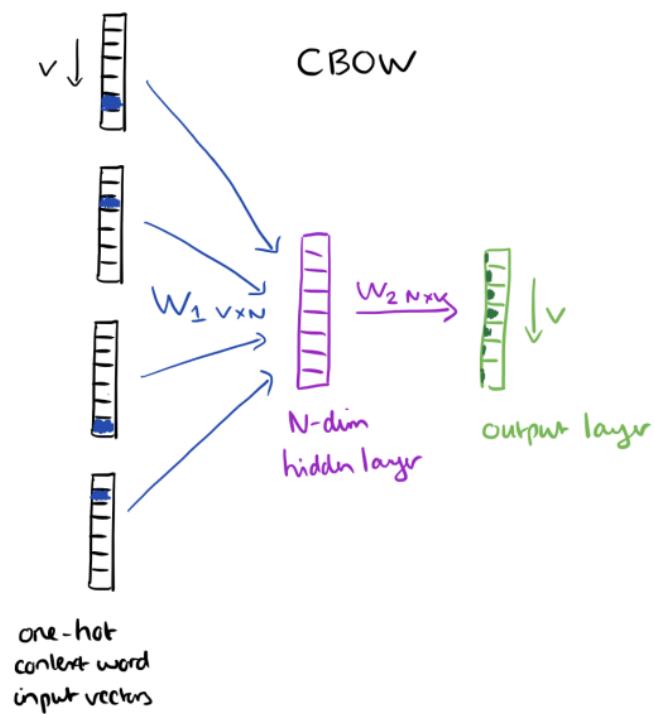
Hàm mục tiêu nhằm tối ưu xác suất xuất hiện của các từ ngữ cảnh (context word) đối với từ đang xét hiện tại: $J(\theta) = -\frac{1}{T} \prod_{t=1}^T \prod_{j=-m, j \neq 0}^m p(w_{t+j}|w_t; \theta)$

Có 2 kiến trúc khác nhau của word2vec, là CBoW và Skip-Gram:

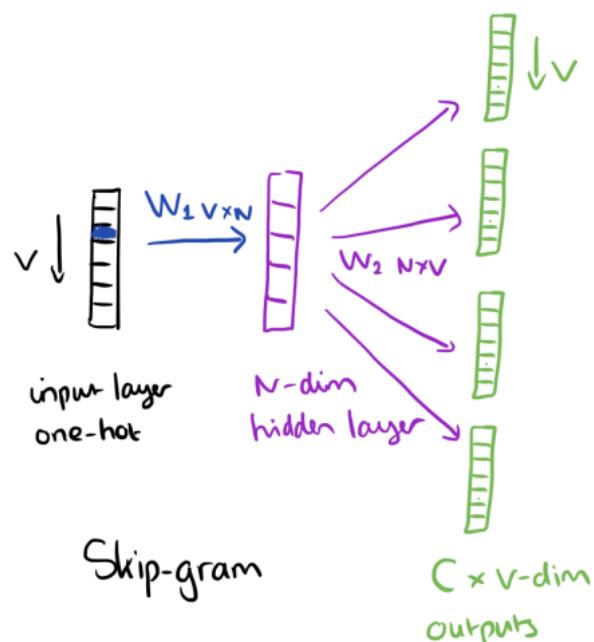
- Cbow: Cho trước ngữ cảnh ta dự đoán xác suất từ đích. Ví dụ: "I ... you", với đầu vào là 2 từ "I" và "you" ta cố gắng dự đoán từ còn thiếu, chẳng hạn "love".
- Skip-Gram: Cho từ đích ta dự đoán xác suất các từ ngữ cảnh (nằm trong context window) của nó. Ví dụ: "... love ...", cho từ "love" ta dự đoán các từ là ngữ cảnh của nó, chẳng hạn "I", "you".

Trong bài báo giới thiệu word2vec, Mikolov và cộng sự có so sánh và cho thấy 2 mô hình này cho kết quả tương đối giống nhau.

Chi tiết mô hình:



Hình 18.7: Mô hình Cbow



Hình 18.8: Mô hình Skip-Gram

Do 2 kiến trúc khá giống nhau nên ta chỉ thảo luận về Skip-Gram.

Mô hình Skip-Gram sẽ input từ đích và dự đoán ra các từ ngữ cảnh. Thay vì input từ đích và output ra nhiều từ ngữ cảnh trong 1 mô hình, họ xây dựng model để input từ đích và output ra 1 từ ngữ cảnh.

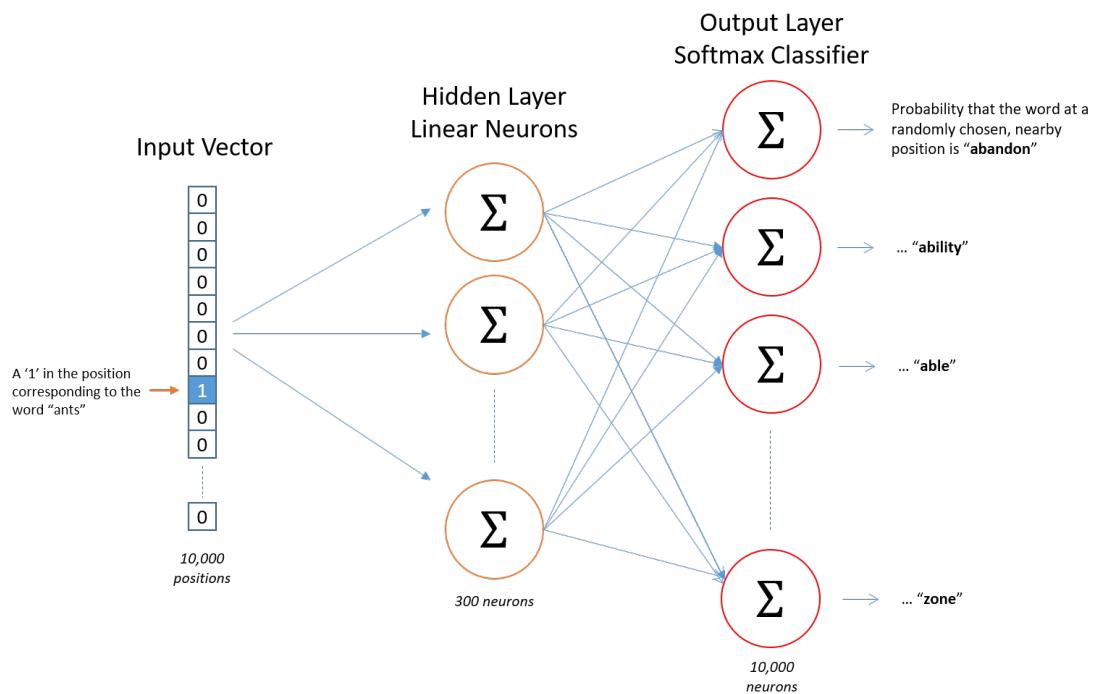
Mô hình là một mạng neural network 2 lớp, với chỉ 1 hidden layer. Input là một từ trong từ điển đã được mã hóa thành dạng one hot vector chiều $V * 1$ với V là kích thước từ điển. Hidden layer không sử dụng activation function có N node, trong đó N chính là độ dài vector embedding của mỗi từ. Output layer có V node, sau đó softmax activation được sử dụng để chuyển về dạng xác suất. Categorical cross entropy loss function được học để dự đoán được từ ngữ cảnh với input là từ đích.

Ví dụ của xây dựng training data:

Source Text	Training Samples
The quick brown fox jumps over the lazy dog. →	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog. →	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog. →	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog. →	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

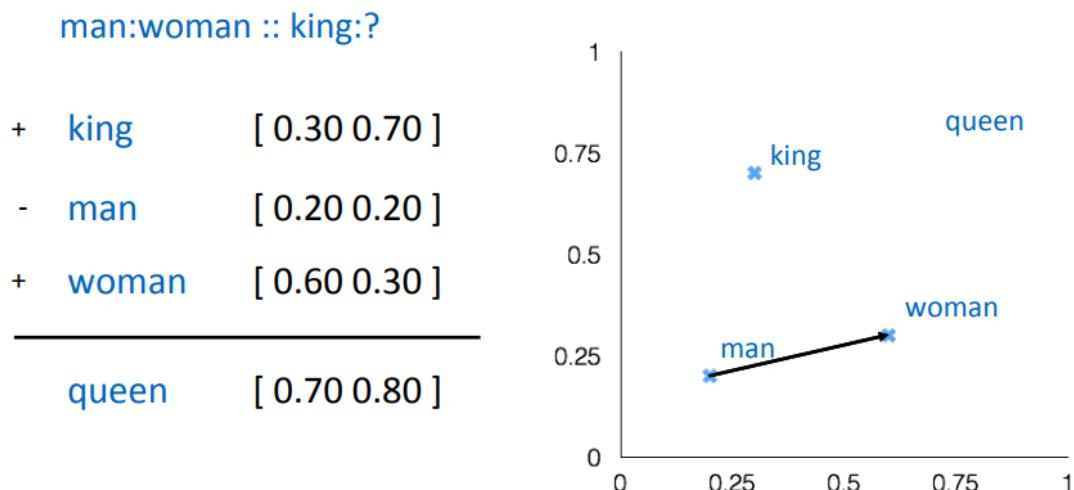
Hình 18.9: Ví dụ của xây dựng training data, window size = 2, tức là lấy 2 từ bên trái và 2 từ bên phải mỗi từ trung tâm làm từ ngữ cảnh (context word)

Ví dụ của model Skip-gram:



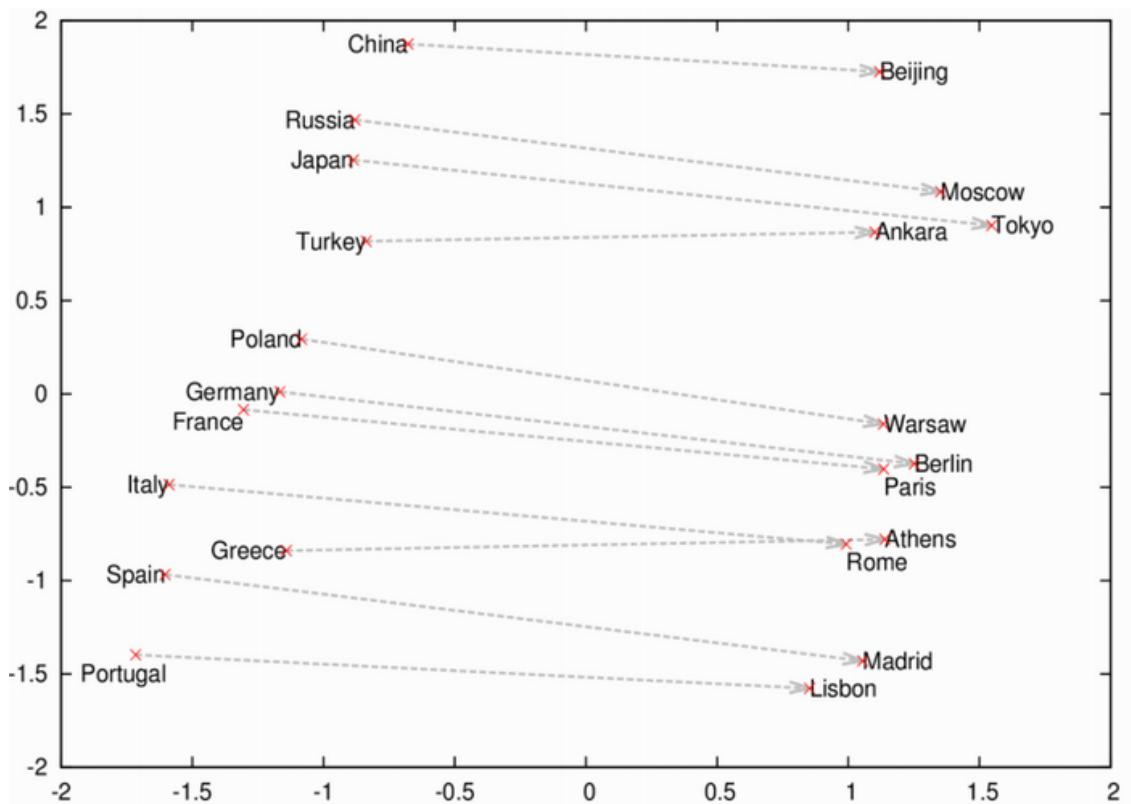
Hình 18.10: Ví dụ của xây dựng Skip-gram

Một số kết quả của Word2vec:



Hình 18.11: Vị trí các word vector trong không gian

Ví dụ trên là ví dụ kinh điển của Word2vec cho thấy các vector biểu diễn tốt quan hệ về mặt ngữ nghĩa của từ vựng như thế nào.



Hình 18.12: Mối quan hệ của đất nước và thủ đô tương ứng

Pre-trained GLOVE Model được sử dụng cho quá trình word embedding.

Mọi người vào link [này](#) để tải file glove.6B.zip

Từng dòng trong file sẽ lưu text và encoded vector khích thước 200*1

18.4.4 Output

Bài toán là dự đoán từ tiếp theo trong chuỗi ở input với ảnh hiện tại, nên output là từ nào trong số 1652 từ trong từ điển mà ta có. Với bài toán phân loại thì softmax activation và categorical_crossentropy loss function được sử dụng.

18.4.5 Model

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 34)	0	
input_2 (InputLayer)	(None, 2048)	0	
embedding_1 (Embedding)	(None, 34, 200)	330400	input_3[0] [0]
dropout_1 (Dropout)	(None, 2048)	0	input_2[0] [0]
dropout_2 (Dropout)	(None, 34, 200)	0	embedding_1[0] [0]

```
dense_1 (Dense)           (None, 256)      524544    dropout_1[0][0]  
-----  
lstm_1 (LSTM)             (None, 256)      467968    dropout_2[0][0]  
-----  
add_1 (Add)               (None, 256)      0         dense_1[0][0]  
                                lstm_1[0][0]  
-----  
dense_2 (Dense)           (None, 256)      65792     add_1[0][0]  
-----  
dense_3 (Dense)           (None, 1652)     424564    dense_2[0][0]  
=====  
Total params: 1,813,268  
Trainable params: 1,813,268  
Non-trainable params: 0
```

18.5 Python code

```
# -*- coding: utf-8 -*-  
  
# Commented out IPython magic to ensure Python compatibility.  
#Thêm thư viện  
import numpy as np  
from numpy import array  
import pandas as pd  
import matplotlib.pyplot as plt  
# %matplotlib inline  
import string  
import os  
from PIL import Image  
import glob  
from pickle import dump, load  
from time import time  
  
from keras.preprocessing import sequence  
from keras.models import Sequential  
from keras.layers import LSTM, Embedding, TimeDistributed, Dense, RepeatVector,\  
Activation, Flatten, Reshape, concatenate, \  
Dropout, BatchNormalization  
from keras.optimizers import Adam, RMSprop  
from keras.layers.wrappers import Bidirectional  
from keras.layers.merge import add  
from keras.applications.inception_v3 import InceptionV3  
from keras.preprocessing import image  
from keras.models import Model  
from keras import Input, layers  
from keras import optimizers  
from keras.applications.inception_v3 import preprocess_input  
from keras.preprocessing.text import Tokenizer  
from keras.preprocessing.sequence import pad_sequences  
from keras.utils import to_categorical
```

```

# Đọc file các caption
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text

filename = "Flickr8k/Flickr8k_text/Flickr8k.token.txt"

doc = load_doc(filename)
print(doc[:300])

# Lưu caption dưới dạng key value:
#id_image : ['caption 1', 'caption 2', 'caption 3', 'caption 4', 'caption 5']
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
            continue
        # take the first token as the image id, the rest as the description
        image_id, image_desc = tokens[0], tokens[1:]
        # extract filename from image id
        image_id = image_id.split('.')[0]
        # convert description tokens back to string
        image_desc = ' '.join(image_desc)
        # create the list if needed
        if image_id not in mapping:
            mapping[image_id] = list()
        # store description
        mapping[image_id].append(image_desc)
    return mapping

descriptions = load_descriptions(doc)
print('Loaded: %d' % len(descriptions))

descriptions['1000268201_693b08cb0e']

# Preprocessing text
def clean_descriptions(descriptions):
    # prepare translation table for removing punctuation
    table = str.maketrans('', '', string.punctuation)
    for key, desc_list in descriptions.items():

```

```
        for i in range(len(desc_list)):
            desc = desc_list[i]
            # tokenize
            desc = desc.split()
            # convert to lower case
            desc = [word.lower() for word in desc]
            # remove punctuation from each token
            desc = [w.translate(table) for w in desc]
            # remove hanging 's' and 'a'
            desc = [word for word in desc if len(word)>1]
            # remove tokens with numbers in them
            desc = [word for word in desc if word.isalpha()]
            # store as string
            desc_list[i] = ' '.join(desc)

# clean descriptions
clean_descriptions(descriptions)

descriptions['1000268201_693b08cb0e']

# Lưu description xuống file
def save_descriptions(descriptions, filename):
    lines = list()
    for key, desc_list in descriptions.items():
        for desc in desc_list:
            lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()

save_descriptions(descriptions, 'descriptions.txt')

# Lấy id ảnh tương ứng với dữ liệu train, test, dev
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)

# load training dataset (6K)
filename = 'Flickr8k/Flickr8k_text/Flickr_8k.trainImages.txt'
```

```

train = load_set(filename)
print('Dataset: %d' % len(train))

# Folder chứa dữ ảnh
images = 'Flickr8k/Flicker8k_Dataset/'
# Lấy lấy các ảnh jpg trong thư mục
img = glob.glob(images + '*.jpg')

# File chứa các id ảnh để train
train_images_file = 'Flickr8k/Flickr8k_text/Flickr_8k.trainImages.txt'
# Read the train image names in a set
train_images = set(open(train_images_file, 'r').read().strip().split('\n'))

# Create a list of all the training images with their full path names
train_img = []

for i in img: # img is list of full path names of all images
    if i[len(images):] in train_images: # Check if the image belongs to training set
        train_img.append(i) # Add it to the list of train images

# File chứa các id ảnh để test
test_images_file = 'Flickr8k/Flickr8k_text/Flickr_8k.testImages.txt'
# Read the validation image names in a set# Read the test image names in a set
test_images = set(open(test_images_file, 'r').read().strip().split('\n'))

# Create a list of all the test images with their full path names
test_img = []

for i in img: # img is list of full path names of all images
    if i[len(images):] in test_images: # Check if the image belongs to test set
        test_img.append(i) # Add it to the list of test images

# Thêm 'startseq', 'endseq' cho chuỗi
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            # create list
            if image_id not in descriptions:
                descriptions[image_id] = list()
            # wrap description in tokens
            desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
            descriptions[image_id].append(desc)

```

```
# store
descriptions[image_id].append(desc)
return descriptions

# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))

# Load ảnh, resize về khích thước mà Inception v3 yêu cầu.
def preprocess(image_path):
    # Convert all the images to size 299x299 as expected by the inception v3 model
    img = image.load_img(image_path, target_size=(299, 299))
    # Convert PIL image to numpy array of 3-dimensions
    x = image.img_to_array(img)
    # Add one more dimension
    x = np.expand_dims(x, axis=0)
    # preprocess the images using preprocess_input() from inception module
    x = preprocess_input(x)
    return x

# Load the inception v3 model
model = InceptionV3(weights='imagenet')

# Tao model mới, bỏ layer cuối từ inception v3
model_new = Model(model.input, model.layers[-2].output)

# Image embedding thành vector (2048, )
def encode(image):
    image = preprocess(image) # preprocess the image
    fea_vec = model_new.predict(image) # Get the encoding vector for the image
    fea_vec = np.reshape(fea_vec, fea_vec.shape[1]) # reshape from (1, 2048) to (2048, )
    return fea_vec

# Gọi hàm encode với các ảnh trong training set
start = time()
encoding_train = {}
for img in train_img:
    encoding_train[img[len(images):]] = encode(img)
print("Time taken in seconds =", time()-start)

# Lưu image embedding lại
with open("Flickr8k/Pickle/encoded_train_images.pkl", "wb") as encoded_pickle:
    dump(encoding_train, encoded_pickle)

# Encode test image
start = time()
encoding_test = {}
for img in test_img:
    encoding_test[img[len(images):]] = encode(img)
```

```

print("Time taken in seconds =", time()-start)

# Save the bottleneck test features to disk
with open("Flickr8k/Pickle/encoded_test_images.pkl", "wb") as encoded_pickle:
    dump(encoding_test, encoded_pickle)

train_features = load(open("Flickr8k/Pickle/encoded_train_images.pkl", "rb"))
print('Photos: train=%d' % len(train_features))

# Tao list các training caption
all_train_descriptions = []
for key, val in train_descriptions.items():
    for cap in val:
        all_train_descriptions.append(cap)
len(all_train_descriptions)

# Chỉ lấy các từ xuất hiện trên 10 lần
word_count_threshold = 10
word_counts = {}
nsents = 0
for sent in all_train_descriptions:
    nsents += 1
    for w in sent.split(' '):
        word_counts[w] = word_counts.get(w, 0) + 1

vocab = [w for w in word_counts if word_counts[w] >= word_count_threshold]
print('preprocessed words %d -> %d' % (len(word_counts), len(vocab)))

ixtoword = {}
wordtoix = {}

ix = 1
for w in vocab:
    wordtoix[w] = ix
    ixtoword[ix] = w
    ix += 1

vocab_size = len(ixtoword) + 1 # Thêm 1 cho từ dùng để padding
vocab_size

# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc

# calculate the length of the description with the most words
def max_length(descriptions):

```

```
lines = to_lines(descriptions)
return max(len(d.split()) for d in lines)

# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)

# data generator cho việc train theo từng batch model.fit_generator()
def data_generator(descriptions, photos, wordtoix, max_length, num_photos_per_batch):
    X1, X2, y = list(), list(), list()
    n=0
    # loop for ever over images
    while 1:
        for key, desc_list in descriptions.items():
            n+=1
            # retrieve the photo feature
            photo = photos[key+'.jpg']
            for desc in desc_list:
                # encode the sequence
                seq = [wordtoix[word] for word in desc.split(' ') if word in wordtoix]
                # split one sequence into multiple X, y pairs
                for i in range(1, len(seq)):
                    # split into input and output pair
                    in_seq, out_seq = seq[:i], seq[i]
                    # pad input sequence
                    in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                    # encode output sequence
                    out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
                    # store
                    X1.append(photo)
                    X2.append(in_seq)
                    y.append(out_seq)
                # yield the batch data
                if n==num_photos_per_batch:
                    yield [[array(X1), array(X2)], array(y)]
                    X1, X2, y = list(), list(), list()
                    n=0

# Load Glove model
glove_dir = ''
embeddings_index = {} # empty dictionary
f = open(os.path.join(glove_dir, 'glove.6B.200d.txt'), encoding="utf-8")

for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
```

```

print('Found %s word vectors.' % len(embeddings_index))

embeddings_index['the']

embedding_dim = 200

# Get 200-dim dense vector for each of the 10000 words in our vocabulary
embedding_matrix = np.zeros((vocab_size, embedding_dim))

for word, i in wordtoix.items():
    #if i < max_words:
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # Words not found in the embedding index will be all zeros
        embedding_matrix[i] = embedding_vector

embedding_matrix.shape

# Tao model
inputs1 = Input(shape=(2048,))
fe1 = Dropout(0.5)(inputs1)
fe2 = Dense(256, activation='relu')(fe1)
inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
se2 = Dropout(0.5)(se1)
se3 = LSTM(256)(se2)
decoder1 = add([fe2, se3])
decoder2 = Dense(256, activation='relu')(decoder1)
outputs = Dense(vocab_size, activation='softmax')(decoder2)
model = Model(inputs=[inputs1, inputs2], outputs=outputs)

model.summary()

# Layer 2 dùng GLOVE Model nên set weight thẳng và không cần train
model.layers[2].set_weights([embedding_matrix])
model.layers[2].trainable = False

model.compile(loss='categorical_crossentropy', optimizer='adam')

model.optimizer.lr = 0.0001
epochs = 10
number_pics_per_bath = 6
steps = len(train_descriptions)//number_pics_per_bath

for i in range(epochs):
    generator = data_generator(train_descriptions, train_features, wordtoix, max_length, n)
    model.fit_generator(generator, epochs=1, steps_per_epoch=steps, verbose=1)

model.save_weights('./model_weights/model_30.h5')

```

```

images = 'Flickr8k/Flicker8k_Dataset/'

with open("Flickr8k/Pickle/encoded_test_images.pkl", "rb") as encoded_pickle:
    encoding_test = load(encoded_pickle)

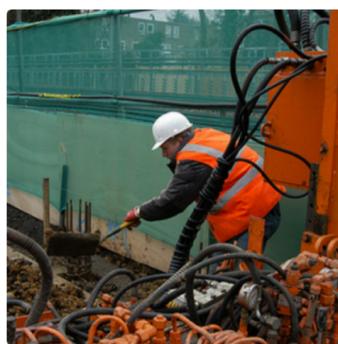
# Với mỗi ảnh mới khi test, ta sẽ bắt đầu chuỗi với 'startseq' rồi
# sau đó cho vào model để dự đoán từ tiếp theo.
# Ta thêm từ vừa được dự đoán vào chuỗi và tiếp tục cho đến khi gặp 'endseq'
# là kết thúc hoặc cho đến khi chuỗi dài 34 từ.
def greedySearch(photo):
    in_text = 'startseq'
    for i in range(max_length):
        sequence = [wordtoix[w] for w in in_text.split() if w in wordtoix]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = model.predict([photo, sequence], verbose=0)
        yhat = np.argmax(yhat)
        word = ixtoword[yhat]
        in_text += ' ' + word
        if word == 'endseq':
            break
    final = in_text.split()
    final = final[1:-1]
    final = ' '.join(final)
    return final

z=5
pic = list(encoding_test.keys())[z]
image = encoding_test[pic].reshape((1,2048))
x=plt.imread(images+pic)
plt.imshow(x)
plt.show()
print(greedySearch(image))

```



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."

Hình 18.13: Các caption dự đoán từ model

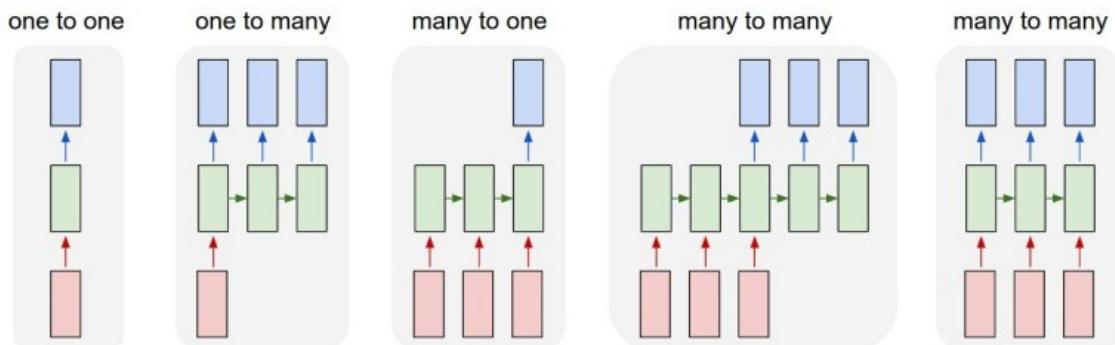


Hình 18.2: Ảnh trong Flickr8k dataset

19. Seq2seq và attention

19.1 Giới thiệu

Mô hình RNN ra đời để xử lý các dữ liệu dạng chuỗi (sequence) như text, video.



Hình 19.1: Các dạng bài toán RNN

Bài toán RNN được phân làm một số dạng:

- **One to one:** mẫu bài toán cho Neural Network (NN) và Convolutional Neural Network (CNN), 1 input và 1 output, ví dụ với bài toán phân loại ảnh MNIST input là ảnh và output ảnh đấy là số nào.
- **One to many:** bài toán có 1 input nhưng nhiều output, ví dụ với bài toán caption cho ảnh, input là 1 ảnh nhưng output là nhiều chữ mô tả cho ảnh đấy, dưới dạng một câu.
- **Many to one:** bài toán có nhiều input nhưng chỉ có 1 output, ví dụ bài toán phân loại hành động trong video, input là nhiều ảnh (frame) tách ra từ video, output là hành động trong video.
- **Many to many:** bài toán có nhiều input và nhiều output, ví dụ bài toán dịch từ tiếng anh sang tiếng việt, input là 1 câu gồm nhiều chữ: "I love Vietnam" và output cũng là 1 câu gồm nhiều chữ "Tôi yêu Việt Nam". Để ý là độ dài sequence của input và output có thể khác nhau.

Mô hình sequence to sequence (seq2seq) sinh ra để giải quyết bài toán many to many và rất thành công trong các bài toán: dịch, tóm tắt đoạn văn. Bài này mình sẽ cùng tìm hiểu về mô hình seq2seq với bài toán dịch từ tiếng anh sang tiếng việt.

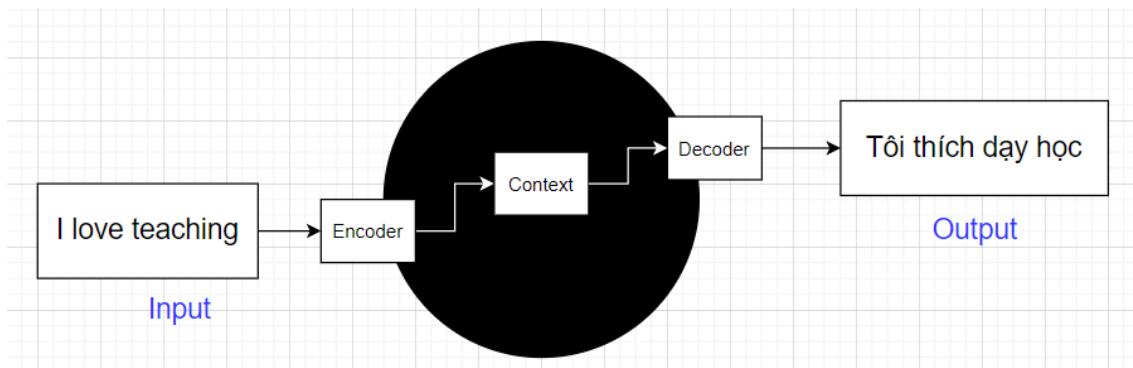
19.2 Mô hình seq2seq

Input của mô hình seq2seq là một câu tiếng anh và output là câu dịch tiếng việt tương ứng, độ dài hai câu này có thể khác nhau. Ví dụ: input: I love teaching -> output: Tôi thích dạy học, input 1 câu 3 từ, output 1 câu 4 từ.



Hình 19.2: Seq2seq model

Mô hình seq2seq gồm 2 thành phần là encoder và decoder.

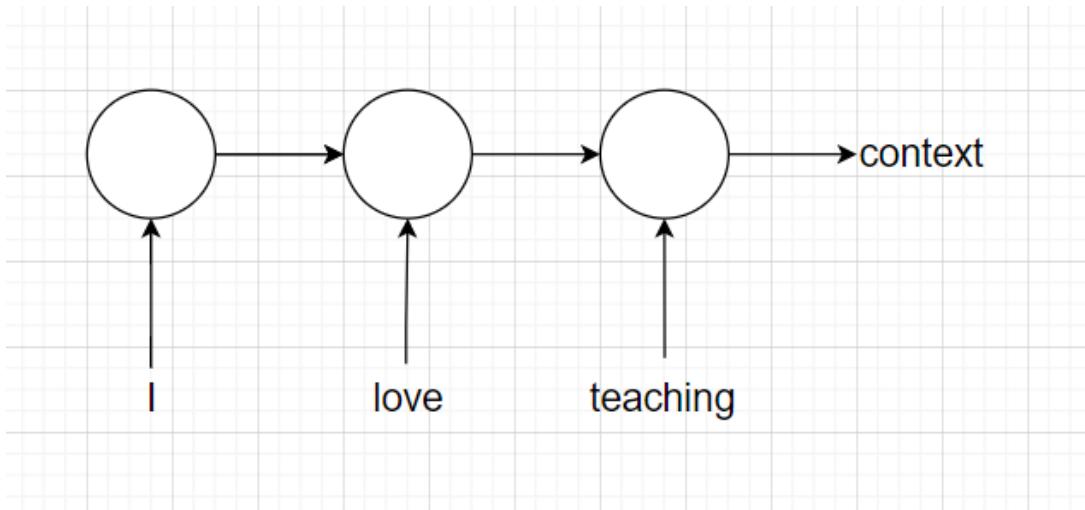


Hình 19.3: Seq2seq model

Encoder nhận input là câu tiếng anh và output ra context vector, còn decoder nhận input là context vector và output ra câu tiếng việt tương ứng. Phần encoder sử dụng mô hình RNN (nói là mô hình RNN nhưng có thể là các mô hình cải tiến như GRU, LSTM) và context vector được dùng là hidden states ở node cuối cùng. Phần decoder cũng là một mô hình RNN với s_0 chính là context vector rồi dần dần sinh ra các từ ở câu dịch.

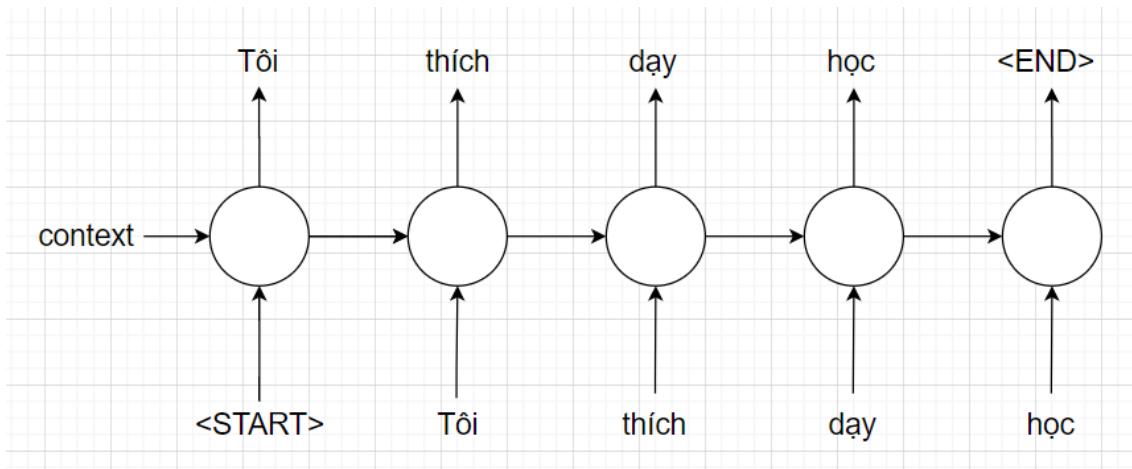
Phần decoder này giống với bài toán image captioning. Ở bài image captioning mình cũng cho ảnh qua pre-trained model để lấy được embedding vector, sau đó cho embedding vector làm s_0 của mô hình RNN rồi sinh ra caption tương ứng với ảnh.

Bài trước mình đã biết model RNN chỉ nhận input dạng vector nên dữ liệu ảnh (từ) sẽ được encode về dạng vector trước khi cho vào model.



Hình 19.4: Mô hình encoder

Các từ trong câu tiếng anh sẽ được embedding thành vector và cho vào mô hình RNN, hidden state ở node cuối cùng sẽ được dùng làm context vector. Về mặt lý thuyết thì context vector sẽ mang đủ thông tin của câu tiếng anh cần dịch và sẽ được làm input cho decoder.



Hình 19.5: Mô hình decoder

2 tag `<START>` và `<END>` được thêm vào câu output để chỉ từ bắt đầu và kết thúc của câu dịch. Mô hình decoder nhận input là context vector. Ở node đầu tiên context vector và tag `<START>` sẽ output ra chữ đầu tiên trong câu dịch, rồi tiếp tục mô hình sinh chữ tiếp theo cho đến khi gặp tag `<END>` hoặc đến `max_length` của câu output thì dừng lại.

Ví dụ code seq2seq cho bài toán dịch với mô hình LSTM mọi người tham khảo ở [đây](#).

Vấn đề: Mô hình seq2seq encode cả câu tiếng anh thành 1 context vector, rồi dùng context vector để sinh ra các từ trong câu dịch ứng tiếng việt. Như vậy khi câu dài thì rất khó cho decoder chỉ dùng 1 context vector có thể sinh ra được câu output chuẩn. Thêm vào đó các mô hình RNN đều bị mất ít nhiều thông tin ở các node ở xa nên bản thân context vector cũng khó để học được thông tin ở các từ ở phần đầu của encoder.

=> Cần có cơ chế để lấy được thông tin các từ ở input cho mỗi từ cần dự đoán ở output thay vì chỉ dựa vào context vector => **Attention** ra đời.

19.3 Cơ chế attention

19.3.1 Motivation

Attention tiếng anh nghĩa là chú ý, hay tập trung. Khi dịch ở mỗi từ tiếng việt ta cần chú ý đến 1 vài từ tiếng anh ở input, hay nói cách khác là có 1 vài từ ở input có ảnh hưởng lớn hơn để dịch từ đấy.



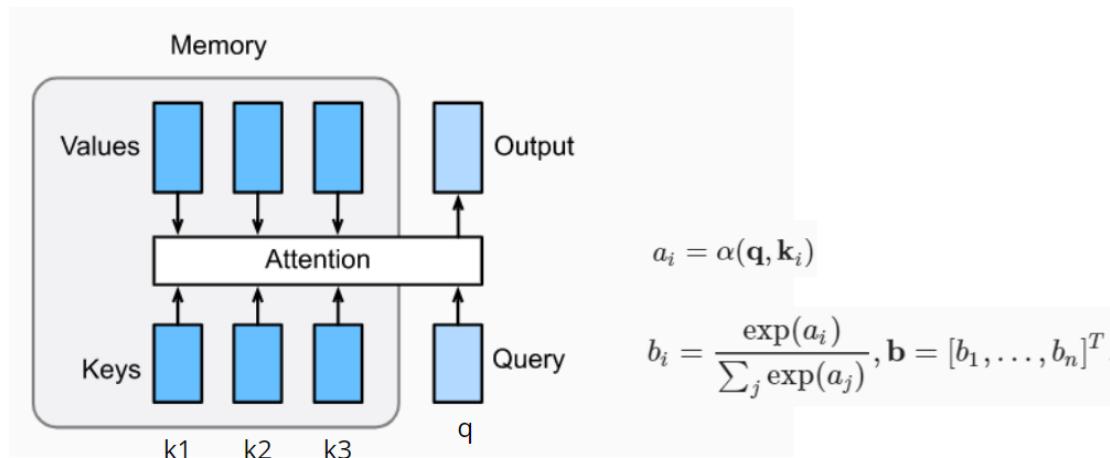
Hình 19.6: Dịch tiếng anh sang tiếng việt, độ quan trọng các từ khi dịch

Ta thấy từ I có trọng số ảnh hưởng lớn tới việc dịch từ tôi, hay từ teaching có ảnh hưởng nhiều tới việc dịch từ dạy và từ học.

=> Do đó khi dịch mỗi từ ta cần chú ý đến các từ ở câu input tiếng anh và đánh trọng số khác nhau cho các từ để dịch chuẩn hơn.

19.3.2 Cách hoạt động

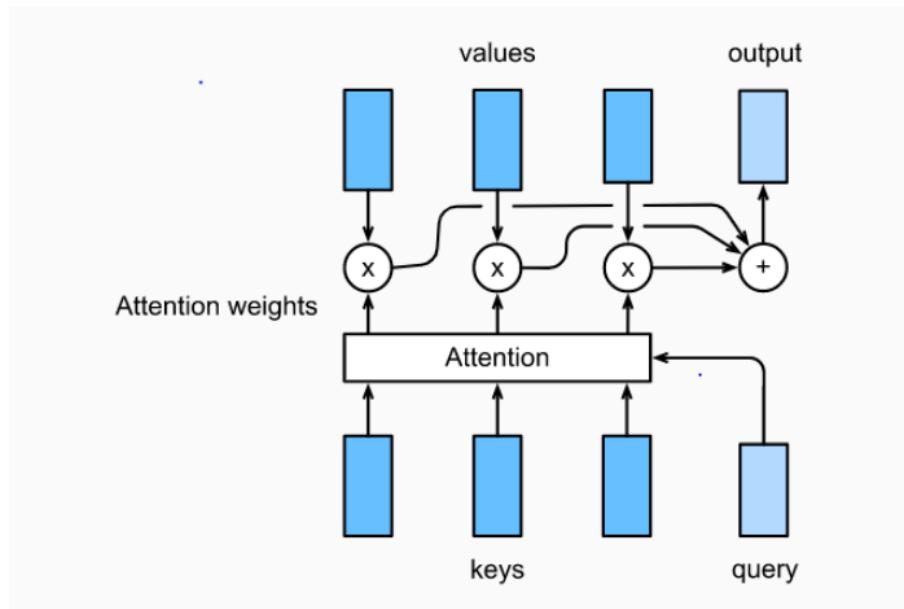
Attention sẽ định nghĩa ra 3 thành phần query, key, value.



Hình 19.7: Các thành phần attention

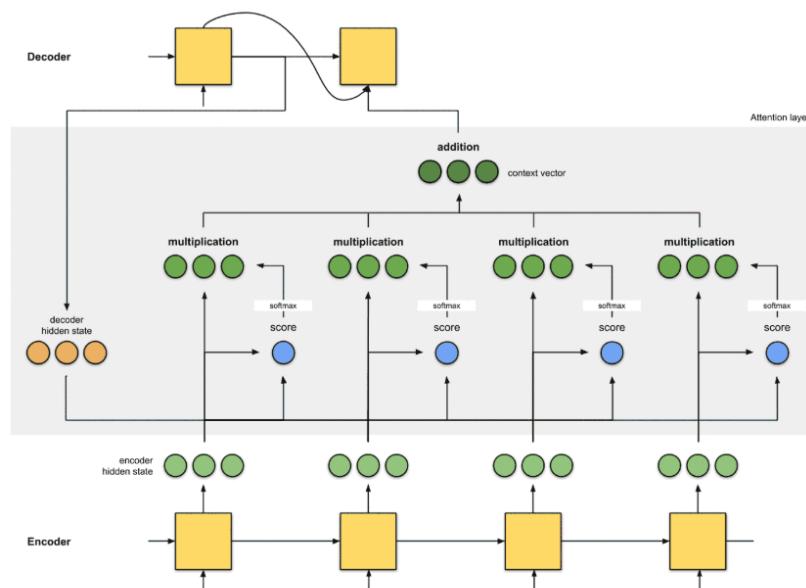
Query (q) lấy thông tin từ từ tiếp theo cần dịch (ví dụ từ dạy). Mỗi từ trong câu input tiếng anh sẽ cho ra 2 thành phần tương ứng là key và value, từ thứ i kí hiệu là k_i, v_i .

Mỗi bộ q, k_i qua hàm α sẽ cho ra a_i tương ứng, a_i chính là độ ảnh hưởng của từ thứ i trong input lên từ cần dự đoán. Sau đó các giá trị a_i được normalize theo hàm softmax được b_i .



Hình 19.8: Các thành phần attention

Cuối cùng các giá trị v_i được tính tổng lại theo hệ số b_i , $\text{output} = \sum_{i=1}^N b_i * v_i$, trong đó N là số từ trong câu input. Việc normalize các giá trị a_i giúp output cùng scale với các giá trị value.



Hình 19.9: Các bước trong attention

Ở phần encoder, thông thường mỗi từ ở input thì hidden state ở mỗi node được lấy làm cả giá trị key và value của từ đấy. Ở phần decoder, ở node 1 gọi input là x_1 , output y_1 và hidden state s_1 ; ở node 2 gọi input là x_2 , output y_2 . Query là hidden state của node trước của node cần dự đoán từ tiếp theo (s_1). Các bước thực hiện:

- Tính score: $a_i = \alpha(q, k_i)$
- Normalize score: b_i

- Tính output: $\text{output_attention} = \sum_{i=1}^N b_i * v_i$
- Sau đó kết hợp hidden state ở node trước s_1 , input node hiện tại x_2 và giá trị output_attention để dự đoán từ tiếp theo y_2 .

Name	Alignment score function
Content-base attention	$\text{score}(s_t, h_i) = \text{cosine}[s_t, h_i]$
Additive(*)	$\text{score}(s_t, h_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[s_t; h_i])$
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a s_t)$ Note: This simplifies the softmax alignment to only depend on the target position.
General	$\text{score}(s_t, h_i) = s_t^\top \mathbf{W}_a h_i$ where \mathbf{W}_a is a trainable weight matrix in the attention layer.
Dot-Product	$\text{score}(s_t, h_i) = s_t^\top h_i$
Scaled Dot-Product(^)	$\text{score}(s_t, h_i) = \frac{s_t^\top h_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.

Hình 19.10: Một số hàm α hay được sử dụng

Nhận xét: Cơ chế attention không chỉ dùng context vector mà còn sử dụng hidden state ở từng từ trong input với trọng số ảnh hưởng tương ứng, nên việc dự đoán từ tiếp theo sẽ tốt hơn cũng như không sợ tình trạng từ ở xa bị mất thông tin ở context vector.

Ngoài ra các mô hình deep learning hay bị nói là hộp đen (black box) vì mô hình không giải thích được, attention phần nào giúp visualize được kết quả dự đoán, ví dụ từ nào ở output ảnh hưởng nhiều bởi từ nào trong input. Do đó model học được quan hệ giữa các từ trong input và output để đưa ra kết quả dự đoán.

Lúc đầu cơ chế attention được dùng trong bài toán seq2seq, về sau do ý tưởng attention quá hay nên được dùng trong rất nhiều bài toán khác, ví dụ như trong CNN người ta dùng cơ chế attention để xem pixel nào quan trọng đến việc dự đoán, feature map nào quan trọng hơn trong CNN layer,. Giống như resnet, attention cũng là 1 đột phá trong deep learning. Mọi người để ý thì các mô hình mới hiện tại đều sử dụng cơ chế attention.

Generative Adversarial Networks (GAN)

20 Giới thiệu về GAN 265

- 20.1 Ứng dụng của GAN
- 20.2 GAN là gì?
- 20.3 Cấu trúc mạng GAN
- 20.4 Code
- 20.5 Bài tập

21 Deep Convolutional GAN (DCGAN) .. 281

- 21.1 Cấu trúc mạng
- 21.2 Tips
- 21.3 Code
- 21.4 Bài tập

22 Conditional GAN (cGAN) 291

- 22.1 Fashion-MNIST
- 22.2 Cấu trúc mạng
- 22.3 Code
- 22.4 Bài tập

Bibliography 299

- Articles
- online
- Books

20. Giới thiệu về GAN

Mô hình GAN được giới thiệu bởi Ian J. Goodfellow vào năm 2014 và đã đạt được rất nhiều thành công lớn trong Deep Learning nói riêng và AI nói chung. Yann LeCun, VP and Chief AI Scientist, Facebook, từng mô tả về GAN: "The most interesting idea in the last 10 years in Machine Learning". Để mọi người thấy được các ứng dụng của GAN, phần dưới tôi sẽ trình bày một vài ứng dụng điển hình của GAN.

20.1 Ứng dụng của GAN

20.1.1 Generate Photographs of Human Faces

Ví dụ về ảnh mặt người do GAN sinh ra từ 2014 đến 2017. Mọi người có thể thấy chất lượng ảnh sinh ra tốt lên đáng kể theo thời gian.



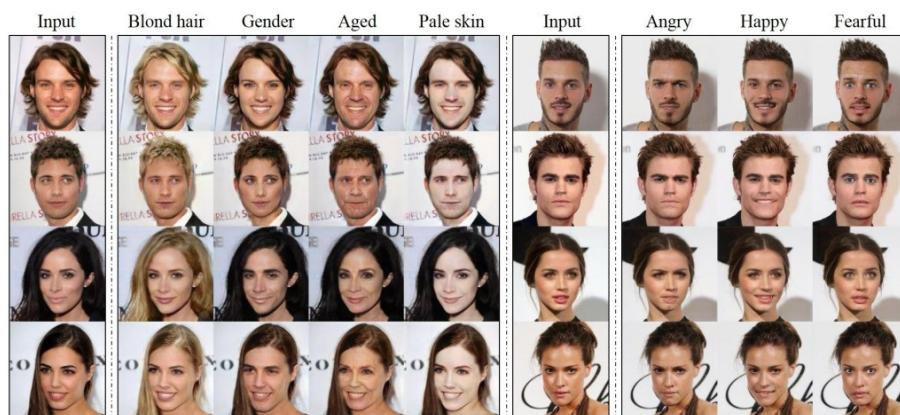
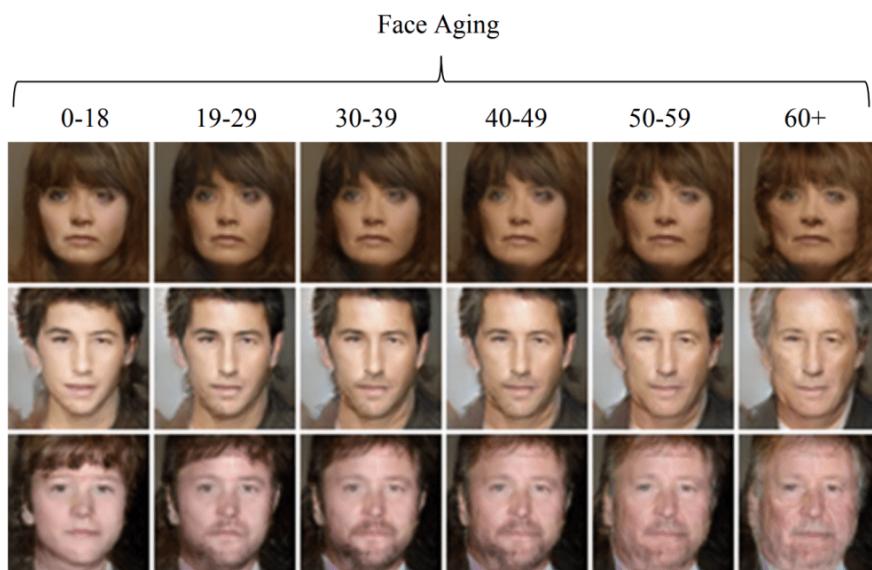
Hình 20.1: Ảnh mặt GAN sinh ra qua các năm ,Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation, 2018.

Hình dưới là ảnh sinh ra bởi GAN năm 2018, phải để ý rất chi tiết thì mới có thể phân biệt được ảnh mặt đấy là sinh ra hay ảnh thật.

Hình 20.2: [StyleGAN](#)

20.1.2 Image editing

Chắc mọi người vẫn nhớ tới FaceApp làm mưa làm gió trong thời gian vừa qua. Nó là một ứng dụng của GAN để sửa các thuộc tính của khuôn mặt như màu tóc, da, giới tính, cảm xúc hay độ tuổi.

Hình 20.3: [StarGAN](#)Hình 20.4: [Age-cGAN](#)

20.1.3 Generate Anime characters

Việc thuê các họa sĩ thiết kế các nhân vật anime rất đắt đỏ thế nên GAN được sử dụng để tự động sinh ra các nhân vật anime.

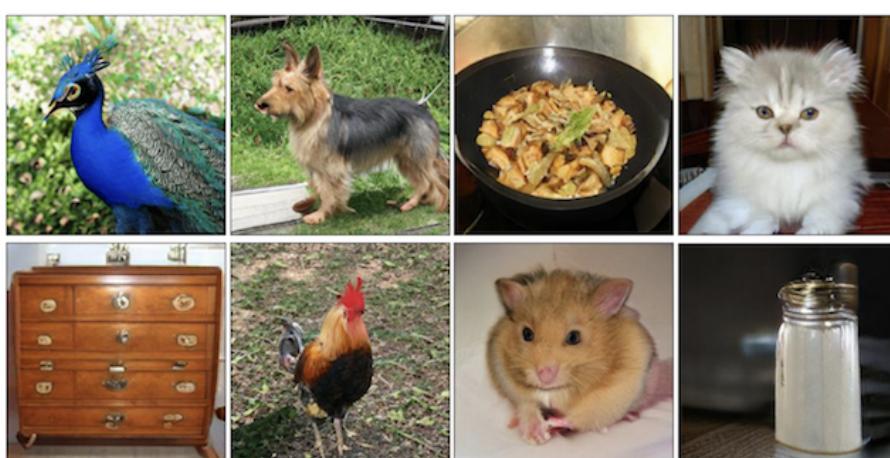


Figure 7: Generated samples

Hình 20.5: Ảnh anime sinh ra, [nguồn](#)

20.1.4 Generate Realistic Photographs

Năm 2018, Andrew Brock cho ra paper [bigGAN](#) với có khả năng sinh ra các ảnh tự nhiên rất khó phân biệt với ảnh chụp thường.



Hình 20.6: Example of Realistic Synthetic Photographs Generated with BigGAN Taken from Large Scale GAN Training for High Fidelity Natural Image Synthesis, 2018.

20.1.5 Image-to-Image Translation

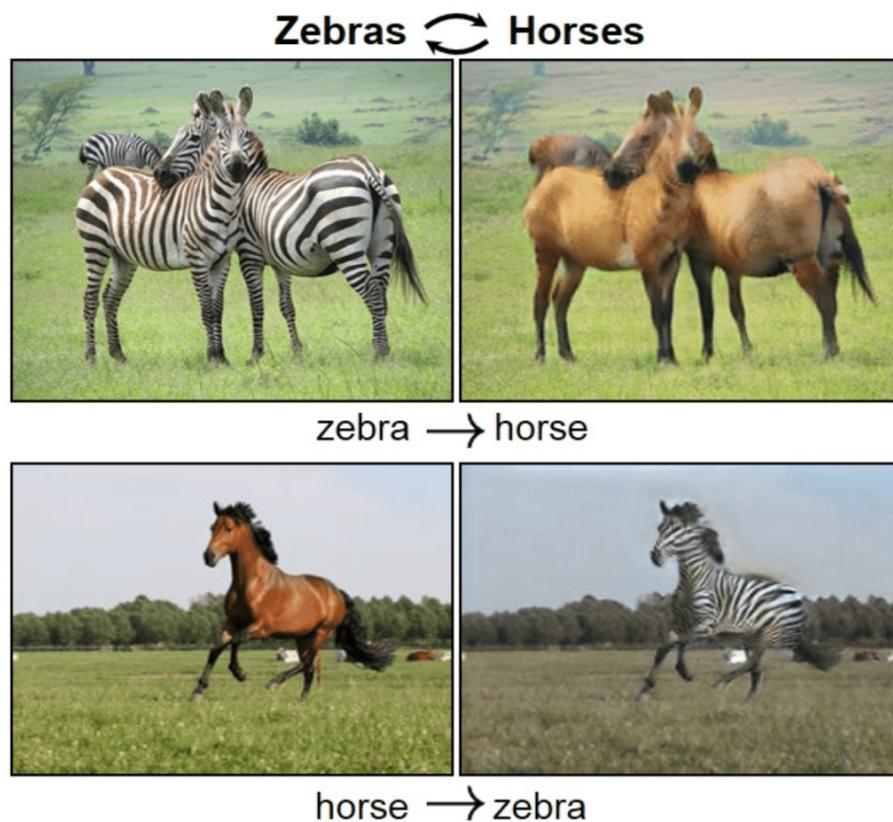
Ví dụ điển hình của mô hình image to image translation là Pix2pix. Input là 1 ảnh và output là 1 ảnh tương ứng, ví dụ input là ảnh không màu, output là ảnh màu. Mọi người có thể vào [đây](#) thử, input là bản phác (draft) con mèo, output là ảnh con mèo hay input là các khối block, output là ảnh ngôi nhà.



Hình 20.7: Ví dụ ảnh draft sang ảnh màu, taken from Image-to-Image Translation with Conditional Adversarial Networks, 2016.

20.1.6 Unsupervised Image-to-image translation

Bài toán trên (Image-to-image translation) là supervised tức là ta có các dữ liệu thành cặp (input, output) như bản phác thảo của cái cặp và ảnh màu của nó. Tuy nhiên khi muốn chuyển dữ liệu từ domain này sang domain khác (từ ngựa thường sang ngựa vằn) thì ta không có sẵn các cặp dữ liệu để train, đó gọi là bài toán unsupervised.



Hình 20.8: CycleGAN

20.1.7 Super-resolution

GAN có thể dùng để tăng chất lượng của ảnh từ độ phân giải thấp lên độ phân giải cao với kết quả rất tốt.



Figure 2: From left to right: bicubic interpolation, deep residual network optimized for MSE, deep residual generative adversarial network optimized for a loss more sensitive to human perception, original HR image. Corresponding PSNR and SSIM are shown in brackets. [4× upscaling]

Hình 20.9: SRGAN

20.1.8 Text to image

GAN có thể học sinh ra ảnh với input là một câu.



Hình 20.10: StackGAN

20.1.9 Generate new human pose

Với ảnh người đứng và dáng đứng mới, GAN có thể sinh ra người với dáng đứng mới.



Hình 20.11: Pose Guided Person Image Generation

20.1.10 Music generation

Ngoài các ứng dụng với ảnh, GAN có thể áp dụng để sinh ra những bản nhạc.

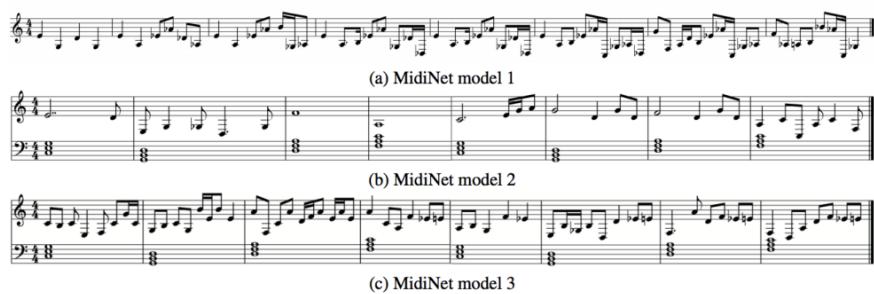


Figure 3. Example result of the melodies (of 8 bars) generated by different implementations of MidiNet.

Hình 20.12: MidiNet

20.2 GAN là gì?

GAN thuộc nhóm generative model. Generative là tính từ nghĩa là khả năng sinh ra, model nghĩa là mô hình. Vậy hiểu đơn giản generative model nghĩa là mô hình có khả năng sinh ra dữ liệu. Hay nói cách khác, GAN là mô hình có khả năng sinh ra dữ liệu mới. Ví dụ như những ảnh mặt người ở trên

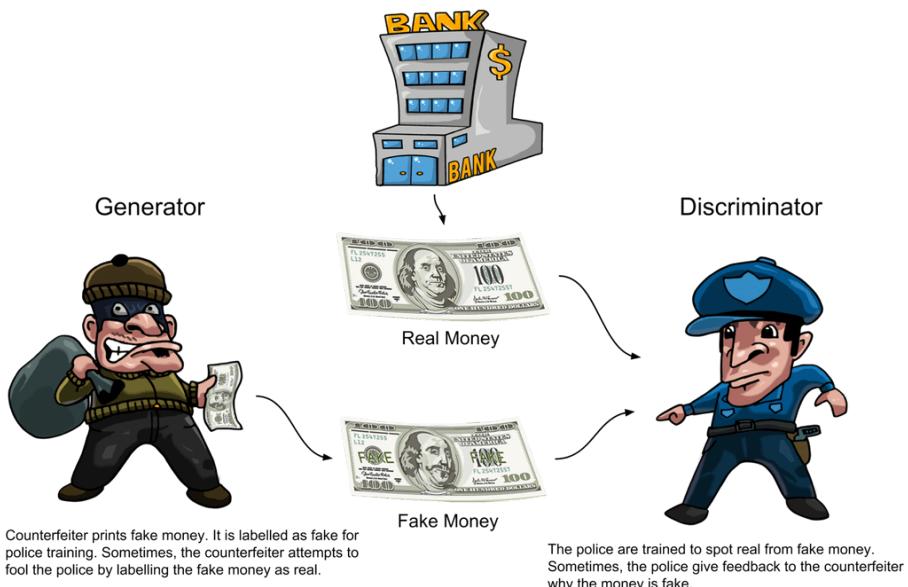
bạn thấy là do GAN sinh ra, không phải mặt người thật. Dữ liệu sinh ra nhìn như thật nhưng không phải thật.

GAN viết tắt cho Generative Adversarial Networks. Generative giống như ở trên, Network có nghĩa là mạng (mô hình), còn Adversarial là đối nghịch. Tên gọi như vậy là do GAN được cấu thành từ 2 mạng gọi là Generator và Discriminator, luôn đối nghịch друг với nhau trong quá trình train mạng GAN. Chi tiết sẽ được trình bày ở phần dưới.

Tóm lại GAN là mạng để sinh dữ liệu mới giống với dữ liệu trong dataset có sẵn và có 2 mạng trong GAN là Generator và Discriminator.

20.3 Cấu trúc mạng GAN

GAN cấu tạo gồm 2 mạng là Generator và Discriminator. Trong khi Generator sinh ra các dữ liệu giống như thật thì Discriminator cố gắng phân biệt đâu là dữ liệu được sinh ra từ Generator và đâu là dữ liệu thật có.



Hình 20.13: Minh họa các mạng trong GAN, [nguồn](#)

Ví dụ bài toán giờ là dùng GAN để generate ra tiền giả mà có thể dùng để chi tiêu được. Dữ liệu có là tiền thật.

Generator giống như người làm tiền giả còn Discriminator giống như cảnh sát. Người làm tiền giả sẽ cố gắng làm ra tiền giả mà cảnh sát cũng không phân biệt được. Còn cảnh sát sẽ phân biệt đâu là tiền thật và đâu là tiền giả. Mục tiêu cuối cùng là người làm tiền giả sẽ làm ra tiền mà cảnh sát cũng không phân biệt được đâu là thật và đâu là giả và thế là mang tiền đi tiêu được.

Trong quá trình train GAN thì cảnh sát có 2 việc: 1 là học cách phân biệt tiền nào là thật, tiền nào là giả, 2 là nói cho thằng làm tiền giả biết là tiền nó làm ra vẫn chưa qua mắt được và cần cải thiện hơn. Dần dần thì thằng làm tiền giả sẽ làm tiền giống tiền thật hơn và cảnh sát cũng thành thạo việc phân biệt tiền giả và tiền thật. Và mong đợi là tiền giả từ GAN sẽ đánh lừa được cảnh sát.

Ý tưởng của GAN bắt nguồn từ [zero-sum non-cooperative game](#), hiểu đơn giản như trò chơi đối kháng 2 người (cờ vua, cờ tướng), nếu một người thắng thì người còn lại sẽ thua. Ở mỗi lượt thì cả 2 đều muốn maximize cơ hội thắng của tôi và minimize cơ hội thắng của đối phương. Discriminator và Generator trong mạng GAN giống như 2 đội thủ trong trò chơi. Trong lý thuyết trò chơi thì GAN model converge khi cả Generator và Discriminator đạt tới trạng thái Nash equilibrium, tức là 2 người chơi đạt trạng thái cân bằng và đi tiếp các bước không làm tăng cơ hội thắng. "A strategy profile is a Nash equilibrium if no player can do better by unilaterally changing his or her strategy", [nguồn](#).

Bài toán: Dùng mạng GAN sinh ra các chữ số viết tay giống với dữ liệu trong [MNIST dataset](#).

20.3.1 Generator

Generator là mạng sinh ra dữ liệu, tức là sinh ra các chữ số giống với dữ liệu trong MNIST dataset. Generator có input là noise (random vector) là output là chữ số.

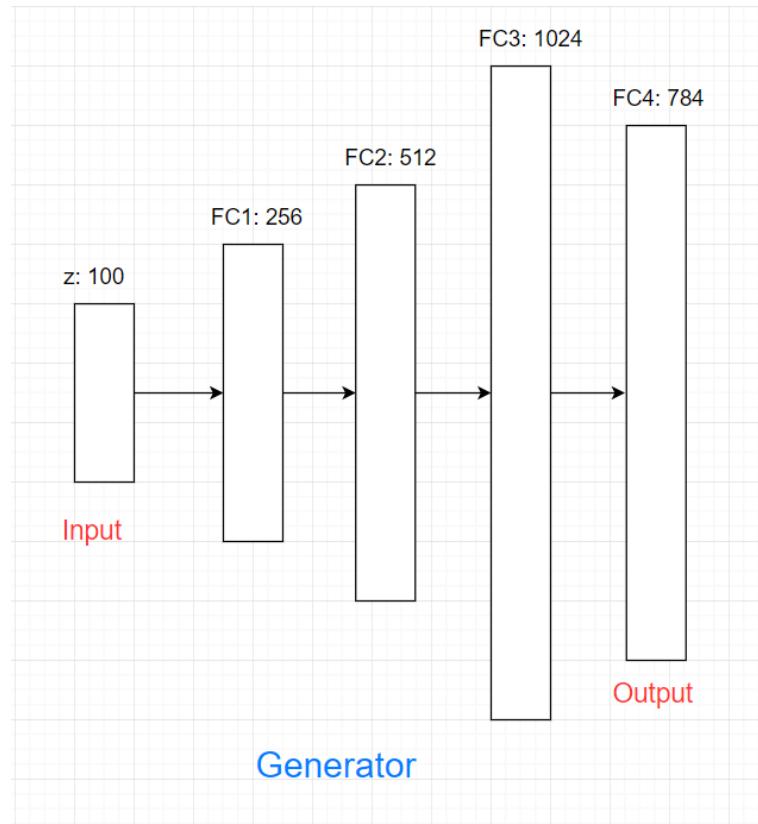
Tại sao input là noise? Vì các chữ số khi viết ra không hoàn toàn giống nhau. Ví dụ số 0 ở hàng đầu tiên có rất nhiều biến dạng nhưng vẫn là số 0. Thế nên input của Generator là noise để khi ta thay đổi noise ngẫu nhiên thì Generator có thể sinh ra một biến dạng khác của chữ viết tay. Noise cho Generator thường được sinh ra từ normal distribution hoặc uniform distribution.



Hình 20.14: MNIST dataset, [nguồn](#)

Input của Generator là noise vector 100 chiều.

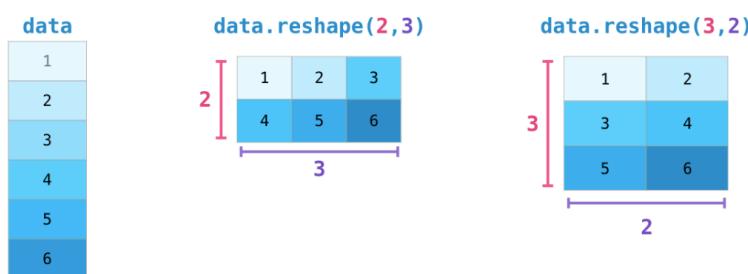
Sau đây mô hình neural network được áp dụng với số node trong hidden layer lần lượt là 256, 512, 1024.



Hình 20.15: Mô hình generator

Output layer có số chiều là 784, vì output đầu ra là ảnh giống với dữ liệu MNIST, ảnh xám kích thước 28×28 (784 pixel).

Output là vector kích thước 784×1 sẽ được reshape về 28×28 đúng định dạng của dữ liệu MNIST.

Hình 20.16: Ví dụ về reshape, [nguồn](#)

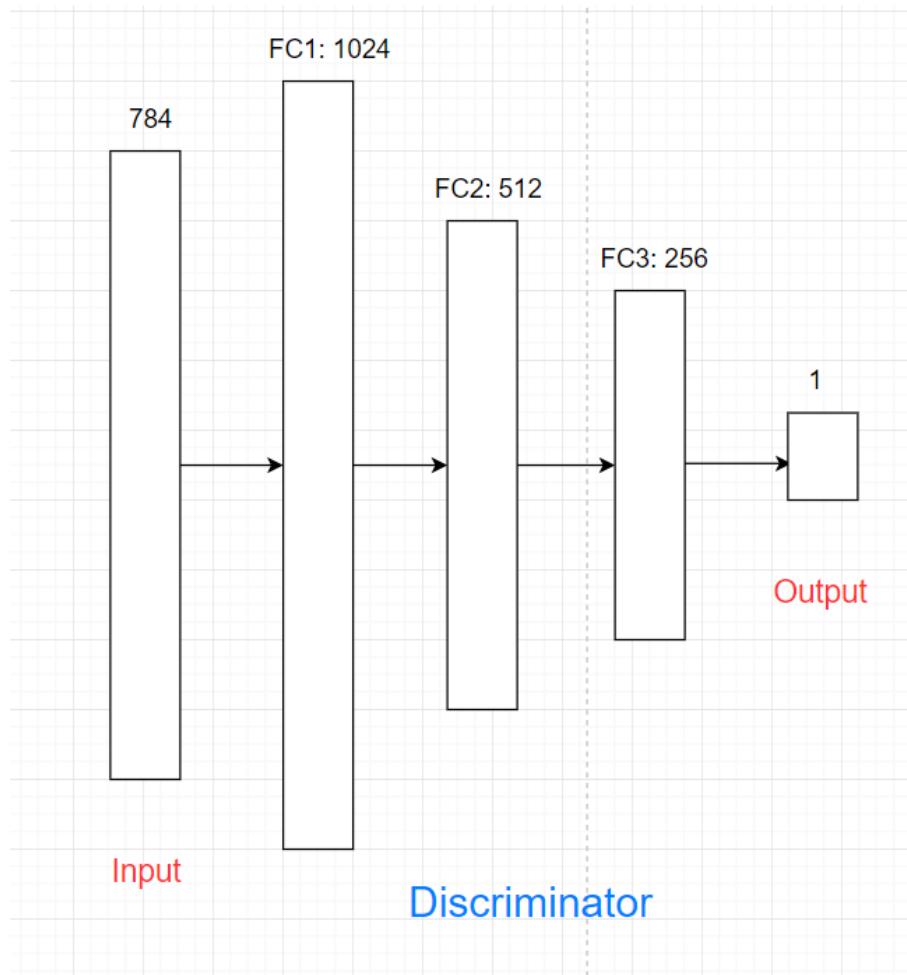
```
# Mô hình Generator
g = Sequential()
g.add(Dense(256, input_dim=z_dim, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(512, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(1024, activation=LeakyReLU(alpha=0.2)))
# Vì dữ liệu ảnh MNIST đã chuẩn hóa về [0, 1] nên hàm G khi sinh ảnh ra cũng cần sinh ra
g.add(Dense(784, activation='sigmoid'))
```

```
g.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

20.3.2 Discriminator

Discriminator là mạng để phân biệt xem dữ liệu là thật (dữ liệu từ dataset) hay giả (dữ liệu sinh ra từ Generator). Trong bài toán này thì discriminator dùng để phân biệt chữ số từ bộ MNIST và dữ liệu sinh ra từ Generator. Discriminator có input là ảnh biểu diễn bằng 784 chiều, output là ảnh thật hay ảnh giả.

Đây là bài toán binary classification, giống với logistic regression.



Hình 20.17: Ví dụ về reshape, [nguồn](#)

Input của Discriminator là ảnh kích thước 784 chiều.

Sau đây mô hình neural network được áp dụng với số node trong hidden layer lần lượt là 1024, 512, 256. Mô hình đối xứng lại với Generator.

Output là 1 node thể hiện xác suất ảnh input là ảnh thật, hàm sigmoid được sử dụng.

```
# Mô hình Discriminator
d = Sequential()
```

```

d.add(Dense(1024, input_dim=784, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(512, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(256, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
# Hàm sigmoid cho bài toán binary classification
d.add(Dense(1, activation='sigmoid'))

```

20.3.3 Loss function

Kí hiệu z là noise đầu vào của generator, x là dữ liệu thật từ bộ dataset.

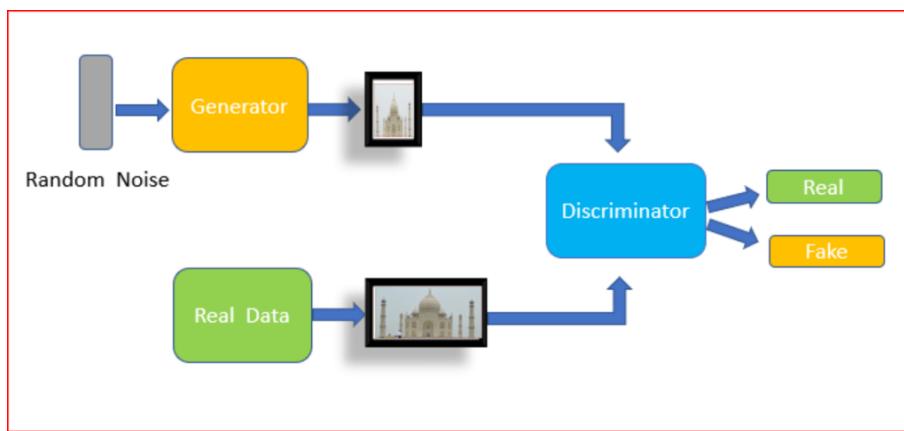
Kí hiệu mạng Generator là G, mạng Discriminator là D. G(z) là ảnh được sinh ra từ Generator. D(x) là giá trị dự đoán của Discriminator xem ảnh x là thật hay không, D(G(z)) là giá trị dự đoán xem ảnh sinh ra từ Generator là ảnh thật hay không.

Vì ta có 2 mạng Generator và Discriminator với mục tiêu khác nhau, nên cần thiết kế 2 loss function cho mỗi mạng.

Discriminator thì cố gắng phân biệt đâu là ảnh thật và đâu là ảnh giả. Vì là bài toán binary classification nên loss function dùng giống với **binary cross-entropy** loss của bài sigmoid.

```
d.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

Giá trị output của model qua hàm sigmoid nên sẽ trong (0, 1) nên Discriminator sẽ được train để input ảnh ở dataset thì output gần 1, còn input là ảnh sinh ra từ Generator thì output gần 0, hay D(x) -> 1 còn D(G(z)) -> 0.



Hình 20.18: Nguồn

Hay nói cách khác là loss function muốn maximize $D(x)$ và minimize $D(G(z))$. Ta có minimize $D(G(z))$ tương đương với maximize $(1 - D(G(z)))$. Do đó loss function của Discriminator trong paper gốc được viết lại thành.

$$\max_D V(D) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

recognize real images better
recognize generated images better

Hình 20.19: Loss Discriminator, [nguồn](#).

E là kì vọng, hiểu đơn giản là lấy trung bình của tất cả dữ liệu, hay maximize $D(x)$ với x là dữ liệu trong traning set.

Generator sẽ học để đánh lừa Discriminator rằng số nó sinh ra là số thật, hay $D(G(z)) \rightarrow 1$. Hay loss function muốn maximize $D(G(z))$, tương đương với minimize $(1 - D(G(z)))$

$$\min_G V(G) = \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))]$$

Optimize G that can fool the discriminator the most.

Hình 20.20: Loss Generator, [nguồn](#).

Do đó ta có thể viết gộp lại loss của mô hình GAN:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

Hình 20.21: Loss GAN, [nguồn](#).

Từ hàm loss của GAN có thể thấy là việc train Generator và Discriminator đối nghịch nhau, trong khi D cố gắng maximize loss V thì G cố gắng minimize V. Quá trình train GAN kết thúc khi model GAN đạt đến trạng thái cân bằng của 2 models, gọi là Nash equilibrium.

20.4 Code

Khi train Discriminator, ta lấy BATCH_SIZE (128) ảnh thật từ dataset với label là 1 (thực ra đã thử và để 0.9 thì kết quả tốt hơn). Thêm vào đó là 128 noise vector (sinh ra từ normal distribution) sau đó cho noise vector qua Generator để tạo ảnh fake, ảnh fake này có label là 0. Sau đó 128 ảnh thật + 128 ảnh fake được dùng để train Discriminator.

```
# Lấy ngẫu nhiên các ảnh từ MNIST dataset (ảnh thật)
image_batch = X_train[np.random.randint(0, X_train.shape[0], size=BATCH_SIZE)]
# Sinh ra noise ngẫu nhiên
noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))

# Dùng Generator sinh ra ảnh từ noise
generated_images = g.predict(noise)
X = np.concatenate((image_batch, generated_images))
# Tạo label
y = np.zeros(2*BATCH_SIZE)
y[:BATCH_SIZE] = 0.9 # gán label bằng 1 cho những ảnh từ MNIST dataset và 0 cho ảnh sinh
```

Train discriminator

```
d.trainable = True
d_loss = d.train_on_batch(X, y)
```

Khi train Generator ta cũng sinh ra 128 ảnh fake từ noise vector, nhưng ta gán label cho ảnh giả là 1 vì ta muốn nó học để đánh lừa được Discriminator. Khi train Generator thì ta không cập nhật các hê số của Discriminator, vì mỗi mạng có mục tiêu riêng.

```
# Train generator
noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))
# Khi train Generator gán label bằng 1 cho những ảnh sinh ra bởi Generator -> cỗ gắng lừa
y2 = np.ones(BATCH_SIZE)
# Khi train Generator thì không cập nhật hê số của Discriminator.
d.trainable = False
g_loss = gan.train_on_batch(noise, y2)

Code GAN sinh ra các chữ số giống với dữ liệu trong bộ MNIST dataset

# Thêm thư viện
from keras.datasets import mnist
from keras.utils import np_utils
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Activation, Flatten
from keras.layers.advanced_activations import LeakyReLU
from keras.optimizers import Adam, RMSprop
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm_notebook

# Lấy dữ liệu từ bộ MNIST
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# Tiền xử lý dữ liệu, reshape từ ảnh xám 28*28 thành vector 784 chiều và đưa dữ liệu từ s
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255

# Số chiều noise vector
z_dim = 100

# Optimizer
adam = Adam(lr=0.0002, beta_1=0.5)

# Mô hình Generator
g = Sequential()
g.add(Dense(256, input_dim=z_dim, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(512, activation=LeakyReLU(alpha=0.2)))
g.add(Dense(1024, activation=LeakyReLU(alpha=0.2)))
# Vì dữ liệu ảnh MNIST đã chuẩn hóa về [0, 1] nên hàm G khi sinh ảnh ra cũng cần sinh ra
g.add(Dense(784, activation='sigmoid'))
g.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])

# Mô hình Discriminator
```

```

d = Sequential()
d.add(Dense(1024, input_dim=784, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(512, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
d.add(Dense(256, activation=LeakyReLU(alpha=0.2)))
d.add(Dropout(0.3))
# Hàm sigmoid cho bài toán binary classification
d.add(Dense(1, activation='sigmoid'))
d.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])

d.trainable = False
inputs = Input(shape=(z_dim, ))
hidden = g(inputs)
output = d(hidden)
gan = Model(inputs, output)
gan.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])

# Hàm vẽ loss function
def plot_loss(losses):
    d_loss = [v[0] for v in losses["D"]]
    g_loss = [v[0] for v in losses["G"]]

    plt.figure(figsize=(10,8))
    plt.plot(d_loss, label="Discriminator loss")
    plt.plot(g_loss, label="Generator loss")

    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

# Hàm vẽ sample từ Generator
def plot_generated(n_ex=10, dim=(1, 10), figsize=(12, 2)):
    noise = np.random.normal(0, 1, size=(n_ex, z_dim))
    generated_images = g.predict(noise)
    generated_images = generated_images.reshape(n_ex, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
        plt.axis('off')
    plt.tight_layout()
    plt.show()

# Lưu giá trị loss và accuracy của Discriminator và Generator
losses = {"D": [], "G": []}

```

```

def train(epochs=1, plt_frq=1, BATCH_SIZE=128):
    # Tính số lần chạy trong mỗi epoch
    batchCount = int(X_train.shape[0] / BATCH_SIZE)
    print('Epochs:', epochs)
    print('Batch size:', BATCH_SIZE)
    print('Batches per epoch:', batchCount)

    for e in tqdm_notebook(range(1, epochs+1)):
        if e == 1 or e%plt_frq == 0:
            print('-'*15, 'Epoch %d' % e, '-'*15)
        for _ in range(batchCount):
            # Lấy ngẫu nhiên các ảnh từ MNIST dataset (ảnh thật)
            image_batch = X_train[np.random.randint(0, X_train.shape[0], size=BATCH_SIZE)]
            # Sinh ra noise ngẫu nhiên
            noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))

            # Dùng Generator sinh ra ảnh từ noise
            generated_images = g.predict(noise)
            X = np.concatenate((image_batch, generated_images))
            # Tao label
            y = np.zeros(2*BATCH_SIZE)
            y[:BATCH_SIZE] = 0.9 # gán label bằng 0.9 cho những ảnh từ MNIST dataset và

            # Train discriminator
            d.trainable = True
            d_loss = d.train_on_batch(X, y)

            # Train generator
            noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))
            # Khi train Generator gán label bằng 1 cho những ảnh sinh ra bởi Generator ->
            y2 = np.ones(BATCH_SIZE)
            # Khi train Generator thì không cập nhật hê số của Discriminator.
            d.trainable = False
            g_loss = gan.train_on_batch(noise, y2)

            # Lưu loss function
            losses["D"].append(d_loss)
            losses["G"].append(g_loss)

            # Vẽ các số được sinh ra để kiểm tra kết quả
            if e == 1 or e%plt_frq == 0:
                plot_generated()
            plot_loss(losses)

# Train GAN model
train(epochs=200, plt_frq=20, BATCH_SIZE=128)

```

20.5 Bài tập

- Tìm các mạng GAN có thể sinh ra ảnh mặt người tốt nhất hiện nay.

2. Dùng mạng GAN sinh ra các ảnh trong bộ dữ liệu cifar-10
3. Tìm hiểu về variational autoencoder (VAE), một mô hình sinh khác giống như GAN.

21. Deep Convolutional GAN (DCGAN)

Bài trước tôi đã giới thiệu về GAN, cấu trúc mạng GAN và hướng dẫn dùng GAN để sinh các số trong bộ dữ liệu MNIST. Tuy nhiên mô hình của Generator và Discriminator đều dùng Neural Network. Trong khi ở bài CNN tôi đã biết CNN xử lý dữ liệu ảnh tốt hơn và hiệu quả hơn rất nhiều so với Neural Network truyền thống. Vậy nên bài này tôi sẽ hướng dẫn áp dụng CNN vào mô hình GAN bài trước, mô hình đấy gọi là Deep Convolutional GAN (DCGAN).

Bài toán: Dùng mạng GAN sinh ra các ảnh giống với dữ liệu trong CIFAR-10 dataset.

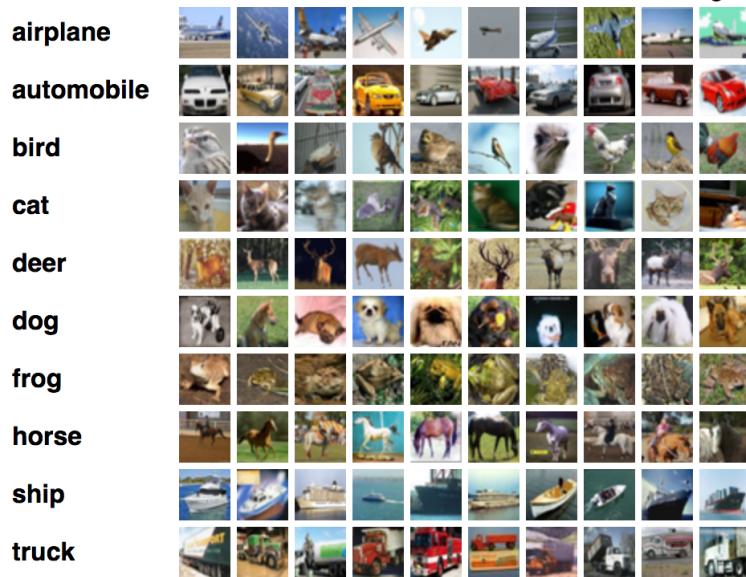
CIFAR-10 dataset bao gồm 60000 ảnh màu kích thước 32x32 thuộc 10 thể loại khác nhau. Mỗi thể loại có 6000 ảnh. Ví dụ ảnh cifar10 ở hình 21.1

21.1 Cấu trúc mạng

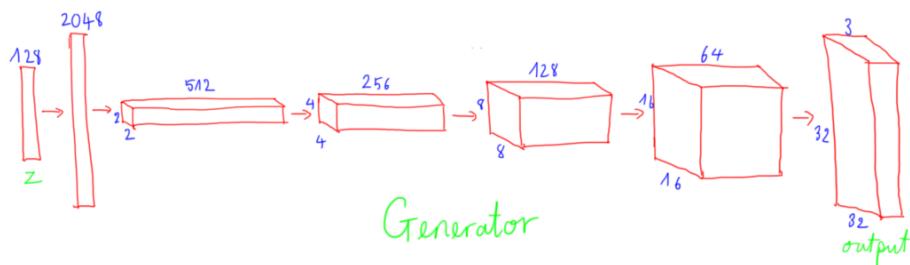
Nhắc lại bài trước 1 chút thì GAN gồm 2 mạng là generator và discriminator. Trong khi discriminator được train để phân biệt ảnh thật (trong dataset) và ảnh fake (do generator sinh ra), thì generator được train để đánh lừa discriminator. Ở bài trước thì cả generator và discriminator đều được xây bằng mạng neural network thông thường với các fully connected layer, bài này thì generator và discriminator được xây dựng bằng mô hình CNN với 2 layers chính là convolutional layer và transposed convolutional layer.

21.1.1 Generator

Mạng Generator nhằm mục đích sinh ảnh fake, input là noise vector kích thước 128 và output là ảnh fake cùng kích thước ảnh thật ($32 * 32 * 3$)



Hình 21.1: Dữ liệu cifar-10



Hình 21.2: Mô hình generator của DCGAN

Các layer trong mạng

- Dense (fully-connected) layer: $128 \times 1 \rightarrow 2048 \times 1$
- Flatten chuyển từ vector về dạng tensor 3d, $2048 \times 1 \rightarrow 2 \times 2 \times 512$
- Transposed convolution stride=2, kernel=256, $2 \times 2 \times 512 \rightarrow 4 \times 4 \times 256$
- Transposed convolution stride=2, kernel=128, $4 \times 4 \times 256 \rightarrow 8 \times 8 \times 128$
- Transposed convolution stride=2, kernel=64, $8 \times 8 \times 128 \rightarrow 16 \times 16 \times 64$
- Transposed convolution stride=2, kernel=3, $16 \times 16 \times 64 \rightarrow 32 \times 32 \times 3$

Đầu tiên thì input noise (128) được dùng full-connected layer chuyển thành 2048 ($= 2 \times 2 \times 512$). Số 2048 được chọn để reshape về dạng tensor 3d ($2 \times 2 \times 512$). Sau đó transposed convolution với stride = 2 được dùng để tăng kích thước tensor lên dần $4 \times 4 \rightarrow 8 \times 8 \rightarrow 16 \times 16 \rightarrow 32 \times 32$. Cho tới khi kích thước tensor 32×32 (bằng đúng width, height của ảnh trong CIFAR-10 dataset) thì ta dùng 3 kernel để ra đúng shape của ảnh.

Mọi người để ý thấy là khi width, height tăng thì depth sẽ giảm, cũng giống như trong mạng

CNN bình thường width, height giảm thì depth sẽ tăng.

Transposed convolution

Transposed convolution hay deconvolution có thể coi là phép toán ngược của convolution. Nếu như convolution với stride > 1 giúp làm giảm kích thước của ảnh thì transposed convolution với stride > 1 sẽ làm tăng kích thước ảnh. Ví dụ stride = 2 và padding = 'SAME' sẽ giúp gấp đôi width, height kích thước của ảnh.

Transposed convolution có 2 kiểu định nghĩa **Kiểu 1** Kiểu 1 được định nghĩa đơn giản hơn lấy từ sách Dive into deep learning. Ý tưởng đơn giản là transposed convolution là phép tính ngược của convolution.

$$\text{Sum} \left(\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} \right) \odot \begin{array}{|c|c|} \hline 1 & 0 \\ \hline 1 & 0 \\ \hline \end{array} = \text{Sum} \left(\begin{array}{|c|c|} \hline 1 & 0 \\ \hline 1 & 0 \\ \hline \end{array} \right) = 2$$

Convolution

Hình 21.3: Convolution s=1, p=0

Nếu như ở phép tính convolution thì 1 vùng kích thước 2×2 được nhân element-wise với kernel và tính tổng viết ra ở output thì ở phép tính transposed convolution mỗi phần tử ở input sẽ được nhân với kernel và ma trận kết quả cùng kích thước với kernel được viết vào output. Nếu các phần tử ở output viết đè lên nhau thì ta sẽ cộng dồn vào.

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline 1 & 3 \\ \hline 3 & 4 \\ \hline \end{array} \otimes \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 1+2 & 2 \\ \hline 1+3 & 1+2+3+4 & 2+4 \\ \hline 3 & 3+4 & 4 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 3 & 2 \\ \hline 4 & 10 & 6 \\ \hline 3 & 7 & 4 \\ \hline \end{array}$$

Transpose convolution

Hình 21.4: Transposed convolution với s=1, p=0

Stride trong transposed convolution được định nghĩa là số bước nhảy khi viết kết quả ra ma trận output

$$\begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 3 & 4 \\ \hline \end{array} \otimes \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 1 & 1 & 2 & 2 \\ \hline 1 & 1 & 2 & 2 \\ \hline 3 & 3 & 4 & 4 \\ \hline 3 & 3 & 4 & 4 \\ \hline \end{array}$$

Transpose convolution $s=2$

Hình 21.5: Transposed convolution với $s=2$, $p=0$

Với padding thì ta tính toán bình thường như với $p=0$ sau đó kết quả ta sẽ bỏ p hàng và cột ở 4 cạnh (trên, dưới, trái, phải)

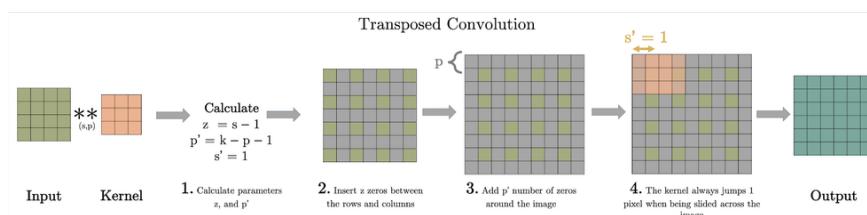
$$\begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 3 & 4 \\ \hline \end{array} \otimes \begin{array}{|c|c|} \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 1 & 1 & 2 & 2 \\ \hline 1 & 1 & 2 & 2 \\ \hline 3 & 3 & 4 & 4 \\ \hline 3 & 3 & 4 & 4 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array}$$

Transpose convolution $s=2, p=1$

Hình 21.6: Transposed convolution $s=2, p=1$

Kiểu 2 Kiểu định nghĩa thứ 2 thì phức tạp hơn nhưng lại có vẻ chuẩn và hay gấp hơn, [nguồn](#).

Ý nghĩa của stride và padding ở đây là khi ta thực hiện phép tính convolution trên output sẽ được kích thước giống input.



Hình 21.7: Các bước thực hiện transposed convolution

Mọi người để ý là bước tính z để chèn 0 vào giữa hay tính lại padding p' để thêm vào mục đích cuối cùng chỉ là để convolution trên output với s, p sẽ được input.

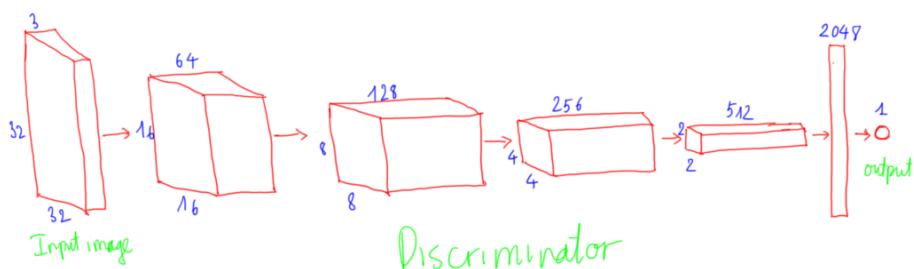
$z = s - 1 = 0$ hàng / cột chèn vào giữa, $p' = k - p - 1 = 2$ padding input. Màu xanh dương là input, xanh lá cây đậm là output.

Ta thấy phép tính convolution với input $4*4$, kernel size $3*3$, $s = 1$, $p = 0$ thì output kích thước $2*2$. Ngược lại phép tính transposed convolution : input là $2*2$, kernel size là $3*3$, $s = 1$, $p = 0$ thì output sẽ có kích thước $4*4$.

$z = s - 1 = 1$ hàng / cột chèn vào giữa, $p' = k - p - 1 = 1$ padding input. Màu xanh dương là input, xanh lá cây đậm là output. Nếu mọi người check lại thì phép tính convolution với input $5*5$, kernel size $3*3$, $s = 2$, $p = 1$ sẽ được output kích thước $3*3$.

21.1.2 Discriminator

Mạng Discriminator nhằm mục đích phân biệt ảnh thật từ dataset và ảnh fake do Generator sinh ra, input là ảnh kích thước $(32 * 32 * 3)$, output là ảnh thật hay fake (binary classification)



Hình 21.8: Mô hình discriminator của DCGAN

Mô hình discriminator đối xứng lại với mô hình generator. Ảnh input được đi qua convolution với stride = 2 để giảm kích thước ảnh từ $32*32 \rightarrow 16*16 \rightarrow 8*8 \rightarrow 4*4 \rightarrow 2*2$. Khi giảm kích thước thì depth tăng dần. Cuối cùng thì tensor shape $2*2*512$ được reshape về vector 2048 và dùng 1 lớp fully connected chuyển từ 2048d về 1d.

Loss function được sử dụng giống như bài trước về GAN.

21.2 Tips

Đây là một số tips để build model và train DCGAN

- Dùng ReLU trong generator trừ output layer
- Output layer trong generator dùng tanh $(-1, 1)$ và scale ảnh input về $(-1, 1)$ sẽ cho kết quả tốt hơn dùng sigmoid và scale ảnh về $(0, 1)$ hoặc để nguyên ảnh.
- Sử dụng LeakyReLU trong discriminator
- Thay thế max pooling bằng convolution với stride = 2
- Sử dụng transposed convolution để upsampling
- Sử dụng batch norm từ output layer trong generator và input layer trong discriminator

21.3 Code

Ảnh CIFAR-10 được scale về (-1, 1) để cùng scale với ảnh sinh ra bởi generator khi dùng tanh activation.

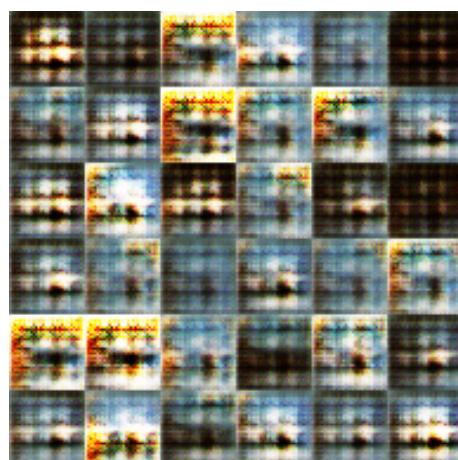
```
def get_data():
    (X_train, y_train), (X_test, y_test) = cifar10.load_data()

    X_train = X_train.astype(np.float32)
    X_test = X_test.astype(np.float32)

    X_train = 2*(X_train/255)-1
    X_test = 2*(X_train/255)-1

    return X_train, X_test
```

Ở những epoch đầu tiên thì generator chỉ sinh ra noise.



Hình 21.9: Ảnh sinh ra bởi generator sau 10 epochs

Tuy nhiên sau 150 epoch thì mạng đã học được thuộc tính của ảnh trong dữ liệu CIFAR-10 và có thể sinh ra được hình con chim, ô tô.



Hình 21.10: Ảnh sinh ra bởi generator sau 150 epochs

```
from tqdm import tqdm_notebook
import matplotlib
import matplotlib.pyplot as plt
from math import ceil
import numpy as np
from keras.models import Sequential, Model
from keras.layers import Input, ReLU, LeakyReLU, Dense
from keras.layers.core import Activation, Reshape
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.layers.core import Flatten
from keras.optimizers import SGD, Adam
from keras.datasets import cifar10
from keras.initializers import RandomNormal

def build_generator(z_dim = 128, n_filter = 64):

    init = RandomNormal(stddev=0.02)

    G = Sequential()
    G.add(Dense(2*2*n_filter*8, input_shape=(z_dim,), use_bias=True, kernel_initializer=init))

    # 2*2*512
    G.add(Reshape((2,2,n_filter*8)))
    G.add(BatchNormalization())
    G.add(LeakyReLU(0.2))

    # 4*4*256
    G.add(Conv2DTranspose(n_filter*4, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    G.add(BatchNormalization())
    G.add(LeakyReLU(0.2))

    # 8*8*128
    G.add(Conv2DTranspose(n_filter*2, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    G.add(BatchNormalization())
    G.add(LeakyReLU(0.2))

    # 16*16*64
    G.add(Conv2DTranspose(n_filter, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    G.add(BatchNormalization())
    G.add(LeakyReLU(0.2))

    # 32*32*3
    G.add(Conv2DTranspose(3, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    G.add(BatchNormalization())
    G.add(Activation('tanh'))

    print('Build Generator')
    print(G.summary())
```

```
    return G

def build_discriminator(input_shape=(32,32,3), n_filter=64):

    init = RandomNormal(stddev=0.02)

    D = Sequential()

    # 16*16*64
    D.add(Conv2D(n_filter, input_shape=input_shape, kernel_size=(5,5), strides=2, padding='valid'))
    D.add(LeakyReLU(0.2))

    # 8*8*64
    D.add(Conv2D(n_filter*2, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    D.add(BatchNormalization())
    D.add(LeakyReLU(0.2))

    # 4*4*64
    D.add(Conv2D(n_filter*4, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    D.add(BatchNormalization())
    D.add(LeakyReLU(0.2))

    # 2*2*64
    D.add(Conv2D(n_filter*8, kernel_size=(5,5), strides=2, padding='same', use_bias=True))
    D.add(BatchNormalization())
    D.add(LeakyReLU(0.2))

    D.add(Flatten())
    D.add(Dense(1, kernel_initializer=init))
    D.add(Activation('sigmoid'))

    print('Build discriminator')
    print(D.summary())

    return D

def get_data():
    (X_train, y_train), (X_test, y_test) = cifar10.load_data()

    X_train = X_train.astype(np.float32)
    X_test = X_test.astype(np.float32)

    X_train = 2*(X_train/255)-1
    X_test = 2*(X_train/255)-1

    return X_train, X_test

def plot_images(images, filename):
```

```

h, w, c = images.shape[1:]
grid_size = ceil(np.sqrt(images.shape[0]))
images = (images + 1) / 2. * 255.
images = images.astype(np.uint8)
images = (images.reshape(grid_size, grid_size, h, w, c)
          .transpose(0, 2, 1, 3, 4)
          .reshape(grid_size*h, grid_size*w, c))
#plt.figure(figsize=(16, 16))
plt.imsave(filename, images)
plt.imshow(images)
plt.show()

def plot_losses(losses_d, losses_g, filename):
    fig, axes = plt.subplots(1, 2, figsize=(8, 2))
    axes[0].plot(losses_d)
    axes[1].plot(losses_g)
    axes[0].set_title("losses_d")
    axes[1].set_title("losses_g")
    plt.tight_layout()
    plt.savefig(filename)
    #plt.close()

def train(n_filter=64, z_dim=100, lr_d=2e-4, lr_g=2e-4, epochs=300, batch_size=128,
          epoch_per_checkpoint=1, n_checkpoint_images=36, verbose=10):
    X_train, _ = get_data()
    image_shape = X_train[0].shape
    print('Image shape {}, min val {}, max val {}'.format(image_shape, np.min(X_train[0]),

    plot_images(X_train[:n_checkpoint_images], 'real_image.png')

    # Build model
    G = build_generator(z_dim, n_filter)
    D = build_discriminator(image_shape, n_filter)

    # Loss for discriminator
    D.compile(Adam(lr=lr_d, beta_1=0.5), loss='binary_crossentropy', metrics=['binary_accuracy'])

    # D(G(X))
    D.trainable = False
    z = Input(shape=(z_dim,))
    D_of_G = Model(inputs=z, outputs=D(G(z)))

    # Loss for generator
    D_of_G.compile(Adam(lr=lr_d, beta_1=0.5), loss='binary_crossentropy', metrics=['binary_accuracy'])

    # Labels for computing the losses
    real_labels = np.ones(shape=(batch_size, 1))
    fake_labels = np.zeros(shape=(batch_size, 1))
    losses_d, losses_g = [], []

```

```

# fix a z vector for training evaluation
z_fixed = np.random.uniform(-1, 1, size=(n_checkpoint_images, z_dim))

for e in tqdm_notebook(range(1, epochs+1)):
    n_steps = X_train.shape[0]//batch_size
    for i in range(n_steps):
        # Train discriminator
        D.trainable = True
        real_images = X_train[i*batch_size:(i+1)*batch_size]
        loss_d_real = D.train_on_batch(x=real_images, y=real_labels)[0]

        z = np.random.uniform(-1, 1, size=(batch_size, z_dim))
        fake_images = G.predict_on_batch(z)
        loss_d_fake = D.train_on_batch(x=fake_images, y=fake_labels)[0]

        loss_d = loss_d_real + loss_d_fake

        # Train generator

        D.trainable = False
        loss_g = D_of_G.train_on_batch(x=z, y=real_labels)[0]

        losses_d.append(loss_d)
        losses_g.append(loss_g)

    if i == 0 and e%verbose == 0:
        print('Epoch {}'.format(e))
        fake_images = G.predict(z_fixed)
        #print("\tPlotting images and losses")
        plot_images(fake_images, "Images1/fake_images_e_{}.png".format(e))
        #plot_losses(losses_d, losses_g, "losses.png")

train()

```

21.4 Bài tập

1. Dùng mạng GAN sinh ra ảnh mặt người trong bộ dữ liệu [CelebA](#)
2. Tìm hiểu cách đánh giá mạng GAN? Làm sao để đánh giá được ảnh sinh ra chất lượng có tốt không? ảnh sinh ra có đa dạng không?.

22. Conditional GAN (cGAN)

Bài trước tôi giới thiệu về DCGAN, dùng deep convolutional network trong mô hình GAN. Tuy nhiên khi ta train GAN xong rồi dùng generator để sinh ảnh mới giống trong dataset tôi không kiểm soát được là ảnh sinh ra giống category nào trong dataset. Ví dụ như dùng GAN để sinh các chữ số trong bộ MNIST, thì khi train xong và dùng generator sinh ảnh thì tôi không biết được ảnh sinh ra sẽ là số mấy ($0 \rightarrow 9$). Bài toán hôm nay muốn kiểm soát được generator sinh ra ảnh theo 1 category nhất định. Ví dụ có thể chỉ định generator sinh ra số 1 chẳng hạn. Mô hình đây gọi là Conditional GAN (cGAN).

22.1 Fashion-MNIST

Dữ liệu Fashion-MNIST về quần áo, giày dép gồm 60000 ảnh training và 10000 ảnh test. Ảnh xám kính thuộc 28×28 thuộc 10 lớp khác nhau.

0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Bài toán hôm nay sẽ dùng cGAN để sinh ra dữ liệu trong từng thể loại ở dữ liệu Fashion-MNIST như sinh ra các ảnh áo Shirt. Ví dụ dữ liệu Fashion-MNIST ở hình 22.1



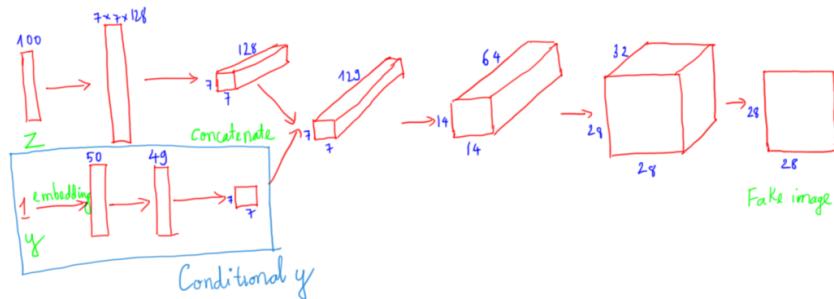
Hình 22.1: Ảnh dữ liệu Fashion-MNIST

22.2 Cấu trúc mạng

22.2.1 Generator

Ở bài trước thì Generator sinh ảnh fake từ noise vector. Cái hay của random vector là có thể sample được nhiều giá trị khác nhau, thành ra khi train xong mạng GAN thì có thể sinh được nhiều ảnh fake khác nhau. Tuy nhiên lại không thể kiểm soát xem ảnh sinh ra thuộc thể loại nào (áo, giày, dép,...).

Bên cạnh noise vector z , tôi sẽ thêm vào y , 1 số từ (0 - 9), mỗi thể loại trong dữ liệu Fashion-MNIST sẽ được encode về 1 số. Tôi mong muốn là z với số y nào thì sẽ cho ra dữ liệu tương ứng thể loại đấy.



Hình 22.2: Mô hình Generator cGAN

Input z là noise vector như bài trước, sau đây được qua dense layer về kích thước $7*7*128$ rồi

reshape về dạng 3D tensor kích thước $7 \times 7 \times 128$ (y1)

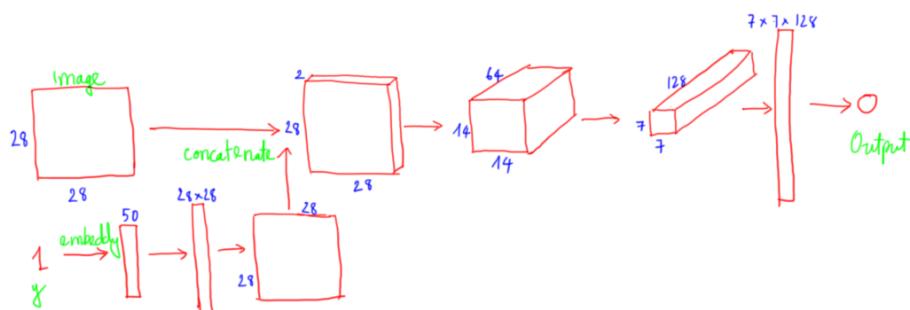
Input y là 1 số được đi qua embedding layer của keras, layer này giống như 1 dictionary map mỗi số thành một vector 50×1 , sau đó được qua dense với output 49 node cuối cùng được reshape về 3D tensor kích thước $7 \times 7 \times 1$ (y2)

Sau đó y1 và y2 được xếp chồng lên nhau thành tensor 3d kích thước $7 \times 7 \times 129$, tiếp đi qua transposed convolution để tăng kích thước lên 14×14 và 28×28 , cuối cùng cho ra output $28 \times 28 \times 1$.

22.2.2 Discriminator

Discriminator mục đích phân biệt ảnh thật trong dataset và ảnh fake sinh ra bởi generator.

Tương tự như ở trong generator bên cạnh ảnh, tôi sẽ thêm vào y, 1 số từ (0 - 9) tương ứng với thể loại trong dữ liệu Fashion-MNIST.

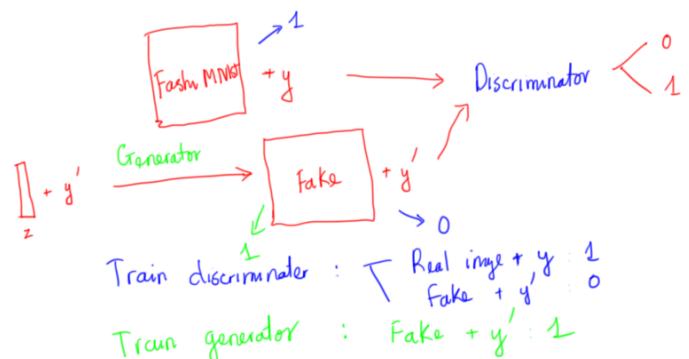


Hình 22.3: Mô hình Discriminator cGAN

Input y là 1 số được đi qua embedding layer của keras, layer này giống như 1 dictionary map mỗi số thành một vector 50×1 , sau đó được qua dense với output 28×28 node cuối cùng được reshape về 3D tensor kích thước $28 \times 28 \times 1$ (y1)

Sau đó y1 và ảnh input được xếp chồng lên nhau thành tensor 3d kích thước $28 \times 28 \times 2$, sau đó tensor đi qua convolution với stride = 2 để giảm kích thước ảnh từ $28 \times 28 \rightarrow 14 \times 14 \rightarrow 7 \times 7$. Khi giảm kích thước thì depth tăng dần. Cuối cùng thì tensor shape $2 \times 2 \times 512$ được reshape về vector 2048 và dùng 1 lớp fully connected chuyển từ 2048d về 1d.

22.2.3 Loss function



Hình 22.4: Loss function cGAN

22.3 Code

Sau khi train xong thì tôi có thể sinh ra ảnh theo từng thể loại trong Fashion-MNIST.



Hình 22.5: Ảnh sinh ra bởi cGAN

```
from numpy import expand_dims
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.fashion_mnist import load_data
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
```

```
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers import Concatenate

def define_discriminator(in_shape=(28,28,1), n_classes=10):
    # label input
    in_label = Input(shape=(1,))
    # embedding for categorical input
    li = Embedding(n_classes, 50)(in_label)
    # scale up to image dimensions with linear activation
    n_nodes = in_shape[0] * in_shape[1]
    li = Dense(n_nodes)(li)
    # reshape to additional channel
    li = Reshape((in_shape[0], in_shape[1], 1))(li)
    # image input
    in_image = Input(shape=in_shape)
    # concat label as a channel
    merge = Concatenate()([in_image, li])
    # downsample
    fe = Conv2D(64, (3,3), strides=(2,2), padding='same')(merge)
    fe = LeakyReLU(alpha=0.2)(fe)
    # downsample
    fe = Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    # flatten feature maps
    fe = Flatten()(fe)
    # dropout
    fe = Dropout(0.4)(fe)
    # output
    out_layer = Dense(1, activation='sigmoid')(fe)
    # define model
    model = Model([in_image, in_label], out_layer)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model

def define_generator(latent_dim, n_classes=10):
    # label input
    in_label = Input(shape=(1,))
    # embedding for categorical input
    li = Embedding(n_classes, 50)(in_label)
    # linear multiplication
```

```

n_nodes = 7 * 7
li = Dense(n_nodes)(li)
# reshape to additional channel
li = Reshape((7, 7, 1))(li)
# image generator input
in_lat = Input(shape=(latent_dim,))
# foundation for 7x7 image
n_nodes = 128 * 7 * 7
gen = Dense(n_nodes)(in_lat)
gen = LeakyReLU(alpha=0.2)(gen)
gen = Reshape((7, 7, 128))(gen)
# merge image gen and label input
merge = Concatenate()([gen, li])
# upsample to 14x14
gen = Conv2DTranspose(64, (4,4), strides=(2,2), padding='same')(merge)
gen = LeakyReLU(alpha=0.2)(gen)
# upsample to 28x28
gen = Conv2DTranspose(32, (4,4), strides=(2,2), padding='same')(gen)
gen = LeakyReLU(alpha=0.2)(gen)
# output
out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
# define model
model = Model([in_lat, in_label], out_layer)
return model

def define_gan(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False
    # get noise and label inputs from generator model
    gen_noise, gen_label = g_model.input
    # get image output from the generator model
    gen_output = g_model.output
    # connect image output and label input from generator as inputs to discriminator
    gan_output = d_model([gen_output, gen_label])
    # define gan model as taking noise and label and outputting a classification
    model = Model([gen_noise, gen_label], gan_output)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model

def load_real_samples():
    # load dataset
    (trainX, trainy), (_, _) = load_data()
    # expand to 3d, e.g. add channels
    X = expand_dims(trainX, axis=-1)
    # convert from ints to floats
    X = X.astype('float32')
    # scale from [0,255] to [-1,1]

```

```
X = (X - 127.5) / 127.5
return [X, trainy]

def generate_real_samples(dataset, n_samples):
    # split into images and labels
    images, labels = dataset
    # choose random instances
    ix = randint(0, images.shape[0], n_samples)
    # select images and labels
    X, labels = images[ix], labels[ix]
    # generate class labels
    y = ones((n_samples, 1))
    return [X, labels], y

def generate_latent_points(latent_dim, n_samples, n_classes=10):
    # generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)
    # generate labels
    labels = randint(0, n_classes, n_samples)
    return [z_input, labels]

def generate_fake_samples(generator, latent_dim, n_samples):
    # generate points in latent space
    z_input, labels_input = generate_latent_points(latent_dim, n_samples)
    # predict outputs
    images = generator.predict([z_input, labels_input])
    # create class labels
    y = zeros((n_samples, 1))
    return [images, labels_input], y

def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128):
    bat_per_epo = int(dataset[0].shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    # manually enumerate epochs
    for i in range(n_epochs):
        # enumerate batches over the training set
        for j in range(bat_per_epo):
            # get randomly selected 'real' samples
            [X_real, labels_real], y_real = generate_real_samples(dataset, half_batch)
            # update discriminator model weights
            d_loss1, _ = d_model.train_on_batch([X_real, labels_real], y_real)
            # generate 'fake' examples
            [X_fake, labels], y_fake = generate_fake_samples(g_model, latent_dim, n_batch)
            # update discriminator model weights
            d_loss2, _ = d_model.train_on_batch([X_fake, labels], y_fake)
            # prepare points in latent space as input for the generator
            [z_input, labels_input] = generate_latent_points(latent_dim, n_batch)
```

```
# create inverted labels for the fake samples
y_gan = ones((n_batch, 1))
# update the generator via the discriminator's error
g_loss = gan_model.train_on_batch([z_input, labels_input], y_gan)
# summarize loss on this batch
print('>%d, %d/%d, d1=% .3f, d2=% .3f g=% .3f' %
      (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))

# save the generator model
g_model.save('cgan_generator.h5')

# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train model
train(g_model, d_model, gan_model, dataset, latent_dim)
```

22.4 Bài tập

1. Sinh ra dữ liệu ở từng lớp trong bộ cifar-10.

Bibliography

Articles

- [7] Ross Girshick. “Fast r-cnn”. In: (Apr. 2015). DOI: 10.1109/ICCV.2015.169 (cited on page 196).
- [8] Ross Girshick et al. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Nov. 2013). DOI: 10.1109/CVPR.2014.81 (cited on page 195).
- [15] Alireza Norouzi et al. “Medical Image Segmentation Methods, Algorithms, and Applications”. In: *IETE Technical Review* 31 (June 2014), pages 199–213. DOI: 10.1080/02564602.2014.906861 (cited on page 218).
- [16] Alexey Novikov et al. “Fully Convolutional Architectures for Multi-Class Segmentation in Chest Radiographs”. In: *IEEE Transactions on Medical Imaging* 37 (Mar. 2018). DOI: 10.1109/TMI.2018.2806086 (cited on page 219).
- [18] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (June 2015). DOI: 10.1109/TPAMI.2016.2577031 (cited on page 199).
- [26] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (June 2014), pages 1929–1958 (cited on page 183).

Websites

- [1] *A brief survey of tensors*. URL: <https://www.slideshare.net/BertOnEarnshaw/a-brief-survey-of-tensors> (cited on page 113).
- [2] *A Gentle Introduction to Transfer Learning for Deep Learning*. URL: <https://machinelearningmastery.com/transfer-learning-for-deep-learning/> (cited on page 162).

- [3] *Convolutional Neural Networks (CNNs / ConvNets)*. URL: <https://www.qubole.com/blog/deep-learning-the-latest-trend-in-ai-and-ml> (cited on page 3).
- [4] *Convolutional Neural Networks (CNNs / ConvNets)*. URL: <http://cs231n.github.io/convolutional-networks/> (cited on pages 130, 217).
- [5] *Deep Learning Framework Power Scores 2018*. URL: <https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a> (cited on page 140).
- [6] *End to End Learning for Self-Driving Cars*. URL: <https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf> (cited on page 154).
- [9] *How LSTM networks solve the problem of vanishing gradients*. URL: <https://medium.com/datadriveninvestor/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577> (cited on page 235).
- [10] *Image Captioning in Deep Learning*. URL: <https://towardsdatascience.com/image-captioning-in-deep-learning-9cd23fb4d8d2> (cited on pages 228, 237).
- [11] *Intersection over Union (IoU) for object detection*. URL: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/> (cited on pages 201, 202).
- [12] *Kernel (image processing)*. URL: [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)) (cited on page 120).
- [13] *Machine Learning*. URL: <https://www.coursera.org/learn/machine-learning> (cited on pages 54, 164, 179, 180).
- [14] *Neurons and Nerves*. URL: <https://askabiologist.asu.edu/neuron-anatomy> (cited on page 84).
- [17] *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms*. URL: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e> (cited on page 198).
- [19] *RGB color model*. URL: https://en.wikipedia.org/wiki/RGB_color_model (cited on page 109).
- [22] *RUN FASTER RCNN USING TENSORFLOW DETECTION API*. URL: <https://data-sci.info/2017/06/27/run-faster-rcnn-tensorflow-detection-api/> (cited on page 194).
- [23] *Selective Search for Object Detection (C++ / Python)*. URL: <https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/> (cited on page 195).
- [24] *Semi-supervised learning with Generative Adversarial Networks (GANs)*. URL: <https://towardsdatascience.com/semi-supervised-learning-with-gans-9f3cb128c5e> (cited on page 173).
- [25] *Simple RNN vs GRU vs LSTM :- Difference lies in More Flexible control*. URL: <https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57> (cited on page 234).
- [27] *Understanding Semantic Segmentation with UNET*. URL: <https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47> (cited on page 218).
- [28] *Xác suất*. URL: <https://en.wikipedia.org/wiki/Probability> (cited on page 66).

Books

- [21] Adrian Rosebrock. *Deep Learning for Computer Vision* (cited on pages 163, 166, 167, 175, 200).