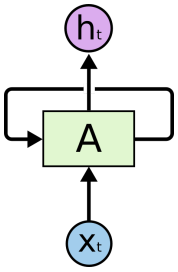
Bài 2 - Lý thuyết về mạng LSTM part 2

22 Apr 2019 - phamdinhkhanh

Menu

1. Mạng nơ ron truy hồi (RNN - Recurrent Neural Network)

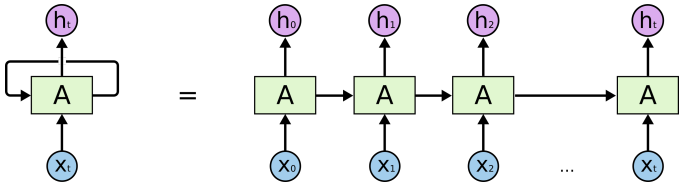
Trong lý thuyết về ngôn ngữ, ngữ nghĩa của một câu được tạo thành từ mối liên kết của những từ trong câu theo một cấu trúc ngữ pháp. Nếu xét từng từ một đứng riêng lẻ ta không thể hiểu được nội dụng của toàn bộ câu, nhưng dựa trên những từ xung quanh ta có thể hiểu được trọn vẹn một câu nói. Như vậy cần phải có một kiến trúc đặc biệt hơn cho các mạng nơ ron biểu diễn ngôn ngữ nhằm mục đích liên kết các từ liền trước với các từ ở hiện tại để tạo ra mối liên hệ xâu chuỗi. Mạng nơ ron truy hồi đã được thiết kế đặc biệt để giải quyết yêu cầu này:



Hình 1: Mạng nơ ron truy hồi với vòng lặp

Hình trên biểu diễn kiến trúc của một mạng nơ ron truy hồi. Trong kiến trúc này mạng nơ ron sử dụng một đầu vào là một véc tơ $\mathbf{x_t}$ và trả ra đầu ra là một giá trị ẩn h_t. Đầu vào được đấu với một thân mạng nơ ron A có tính chất truy hồi và thân này được đấu tới đầu ra h_t.

Vòng lặp A ở thân mạng nơ ron là điểm mấu chốt trong nguyên lý hoạt động của mạng nơ ron truy hồi. Đây là chuỗi sao chép nhiều lần của cùng một kiến trúc nhằm cho phép các thành phần có thể kết nối liền mạch với nhau theo mô hình chuỗi. Đầu ra của vòng lặp trước chính là đầu vào của vòng lặp sau. Nếu trải phẳng thân mạng nơ ron A ta sẽ thu được một mô hình dạng:



Hình 2: Cấu trúc trải phẳng của mạng nơ ron truy hồi

Kiến trúc mạng nơ ron truy hồi này tỏ ra khá thành công trong các tác vụ của deep learning như: Nhận diện tiếng nói (*speech recognition*), các mô hình ngôn ngữ, mô hình dịch, chú thích hình ảnh (*image captioning*),….

Top

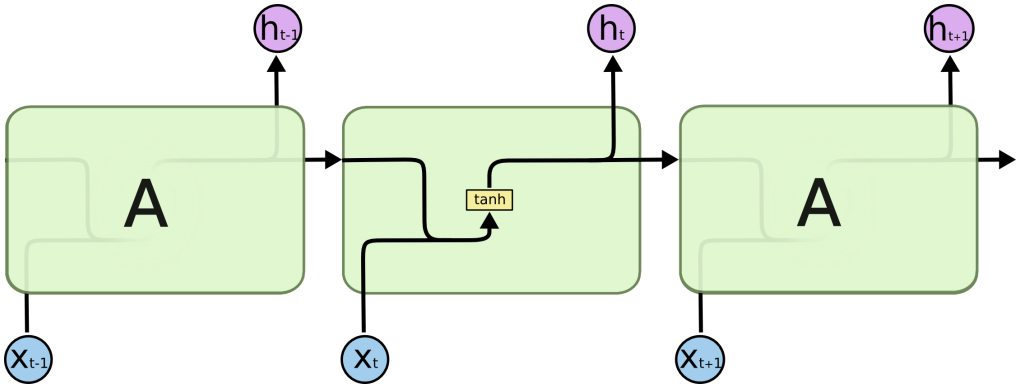
2. Hạn chế của mạng nơ ron truy hồi

Một trong những điểm đặc biệt của RNN đó là nó có khả năng kết nối các thông tin liền trước với nhiệm vụ hiện tại, chẳng hạn như trong câu văn: 'học sinh đang tới *trường học*'. Dường như trong một ngữ cảnh ngắn hạn, từ *trường học* có thể được dự báo ngay tức thì mà không cần thêm các thông tin từ những câu văn khác gần đó. Tuy nhiên có những tình huống đòi hỏi phải có nhiều thông tin hơn chẳng hạn như: 'hôm qua Bống đi học nhưng không mang áo mưa. Trên đường đi học trời mưa. Cặp sách của Bống bị *ướt*'. Chúng ta cần phải học để tìm ra từ *ướt* ở một ngữ cảnh dài hơn so với chỉ 1 câu. Tức là cần phải biết các sự kiện trước đó như *trời mưa*, *không mang áo mưa* để suy ra sự kiện bị *ướt*. Những sự liên kết ngữ nghĩa dài như vậy được gọi là `phụ thuộc dài hạn` (*long-term dependencies*). Về mặt lý thuyết mạng RNN có thể giải quyết được những sự phụ thuộc trong dài hạn. Tuy nhiên trên thực tế RNN lại cho thấy khả năng học trong dài hạn kém hơn. Để hiểu thêm lý do tại sao mạng RNN lại không có khả năng học trong dài hạn cùng đọc bài Leanring Long - Term Dependencies with Gradient Descent is Difficult (http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf). Một trong những nguyên nhân chính được giải thích đó là sự triệt tiêu đạo hàm của hàm cost function sẽ diễn ra khi trải quả chuỗi dài các tính toán truy hồi. Một phiên bản mới của mạng RNN là mạng LSTM ra đời nhằm khắc phục hiện tường này nhờ một cơ chế đặc biệt.

3. Mạng trí nhớ ngắn hạn định hướng dài hạn (LSTM - Long short term memory)

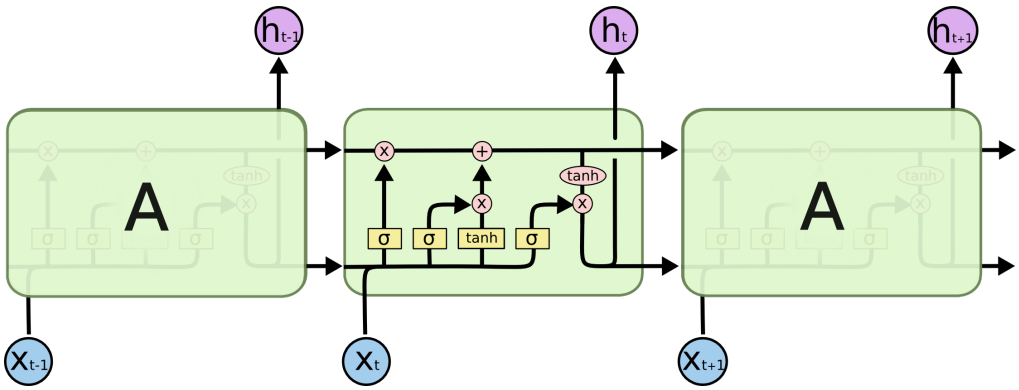
Mạng *trí nhớ ngắn hạn định hướng dài hạn* còn được viết tắt là LSTM làm một kiến trúc đặc biệt của RNN có khả năng học được sự phục thuộc trong dài hạn (*long-term dependencies*) được giới thiệu bởi Hochreiter & Schmidhuber (1997) (http://www.bioinf.jku.at/publications/older/2604.pdf). Kiến trúc này đã được phổ biến và sử dụng rộng rãi cho tới ngày nay. LSTM đã tỏ ra khắc phục được rất nhiều những hạn chế của RNN trước đây về triệt tiêu đạo hàm. Tuy nhiên cấu trúc của chúng có phần phức tạp hơn mặc dù vẫn dữ được tư tưởng chính của RNN là sự sao chép các kiến trúc theo dạng chuỗi.

Một mạng RNN tiêu chuẩn sẽ có kiến trúc rất đơn giản chẳng hạn như đối với kiến trúc gồm một tầng ẩn là hàm tanh như bên dưới.



Hình 3: Sự lặp lại kiến trúc module trong mạng RNN chứa một tầng ẩn

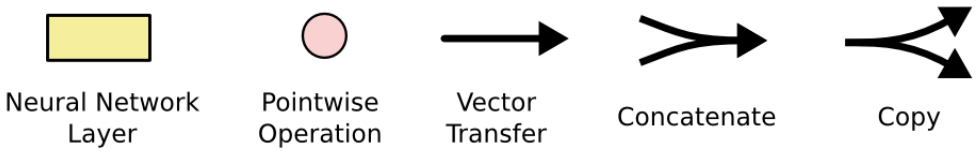
LSTM cũng có một chuỗi dạng như thế nhưng phần kiến trúc lặp lại có cấu trúc khác biệt hơn. Thay vì chỉ có một tầng đơn, chúng có tới 4 tầng ẩn (3 sigmoid và 1 tanh) tương tác với nhau theo một cấu trúc đặc biệt.



Top

Hình 4: Sự lặp lại kiến trúc module trong mạng LSTM chứa 4 tầng ẩn (3 sigmoid và 1 tanh) tương tác

Các kí hiệu có thể diễn giải như sau:

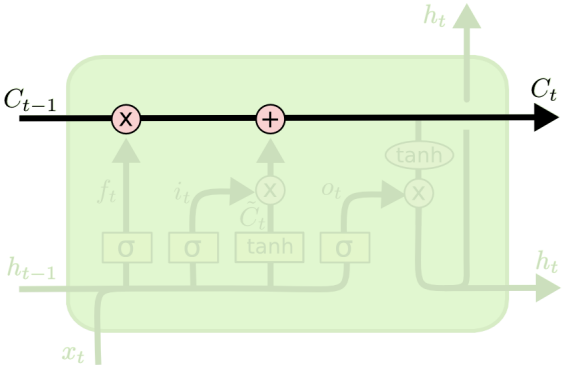


Hình 5: Diễn giải các kí hiệu trong đồ thị mạng nơ ron (áp dụng chung cho toàn bộ bài)

Trong sở đồ tính toán trên, mỗi một phép tính sẽ triển khai trên một véc tơ. Trong đó hình tròn màu hồng biểu diễn một toán tử đối với véc tơ như phép cộng véc tơ, phép nhân vô hướng các véc tơ. Màu vàng thể hiện hàm activation mà mạng nơ ron sử dụng để học trong tầng ẩn, thông thường là các hàm phi tuyến sigmoid và tanh. Kí hiệu 2 đường thẳng nhập vào thể hiện phép chập kết quả trong khi kí hiệu 2 đường thẳng rẽ nhánh thể hiện cho nội dung véc tơ trước đó được sao chép để đi tới một phần khác của mạng nơ ron.

4. Ý tưởng đằng sau LSTM

Ý tưởng chính của LSTM là thành phần ô trạng thái (cell state) được thể hiện qua đường chạy ngang qua đỉnh đồ thị như hình vẽ bên dưới:

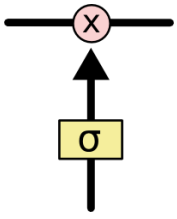


Hình 6: Đường đi của ô trạng thái (cell state) trong mạng LSTM

Ô trạng thái là một dạng băng chuyền chạy thẳng xuyên suốt toàn bộ chuỗi với chỉ một vài tương tác tuyến tính nhỏ giúp cho thông tin có thể truyền dọc theo đồ thị mạng nơ ron ổn định.

LSTM có khả năng xóa và thêm thông tin vào ô trạng thái và điều chỉnh các luồng thông tin này thông qua các cấu trúc gọi là cổng.

Cổng là cơ chế đặc biệt để điều chỉnh luồng thông tin đi qua. Chúng được tổng hợp bởi một tầng ẩn của hàm activation sigmoid và với một toán tử nhân như đồ thị.



Hình 7: Một cổng của hàm sigmoid trong LSTM

Top

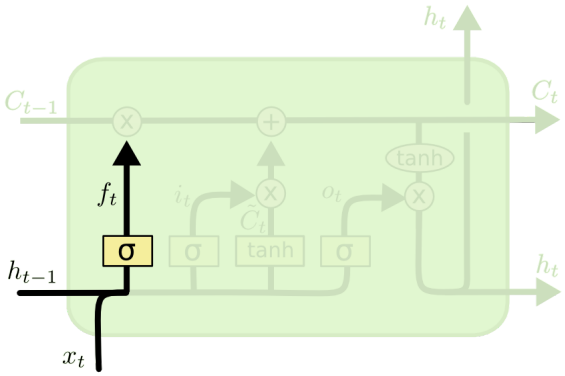
Hàm sigmoid sẽ cho đầu ra là một giá trị xác xuất nằm trong khoảng từ 0 đến 1, thể hiện rằng có bao nhiêu phần thông tin sẽ đi qua cổng. Giá trị bằng 0 ngụ ý rằng không cho phép thông tin nào đi qua, giá trị bằng 1 sẽ cho toàn bộ thông tin đi qua.

Một mạng LSTM sẽ có 3 cổng có kiến trúc dạng này để bảo vệ và kiểm soát các ô trạng thái.

5. Thứ tự các bước của LSTM

Bước đầu tiên trong LSTM sẽ quyết định xem thông tin nào chúng ta sẽ cho phép đi qua ô trạng thái (cell state). Nó được kiểm soát bởi hàm sigmoid trong một tầng gọi là tầng quên (*forget gate layer*). Đầu tiên nó nhận đầu vào là 2 giá trị h_{t-1} và $\mathbf{x_t}$ và trả về một giá trị nằm trong khoảng 0 và 1 cho mỗi giá trị của ô trạng thái C_{t-1}. Nếu giá trị bằng 1 thể hiện 'giữ toàn bộ thông tin' và bằng 0 thể hiện 'bỏ qua toàn bộ chúng'.

Trở lại ví dụ về ngôn ngữ, chúng ta đang cố gắng dự báo từ tiếp theo dựa trên toàn bộ những từ trước đó. Trong những bài toán như vậy, ô trạng thái có thể bao gồm loại của chủ ngữ hiện tại, để cho đại từ ở câu tiếp theo được sử dụng chính xác. Chẳng hạn như chúng ta đang mô tả về một người bạn là con trai thì các đại từ nhân xưng ở tiếp theo phải là anh, thằng, hắn thay vì cô, con ấy (xin lỗi vì lấy ví dụ hơi thô). Tuy nhiên chủ ngữ không phải khi nào cũng cố định. Khi chúng ta nhìn thấy một chủ ngữ mới, chúng ta muốn quên đi loại của một chủ ngữ cũ. Do đó tầng quên cho phép cập nhật thông tin mới và lưu giữ giá trị của nó khi có thay đổi theo thời gian.

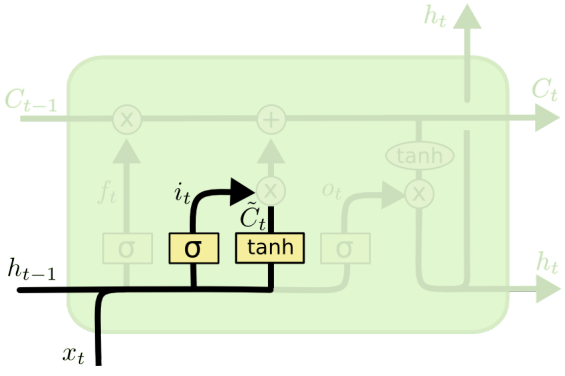


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Hình 8: Tầng cổng quên (*forget gate layer*)

Bước tiếp theo chúng ta sẽ quyết định loại thông tin nào sẽ được lưu trữ trong ô trạng thái. Bước này bao gồm 2 phần. Phần đầu tiên là một tầng ẩn của hàm sigmoid được gọi là tầng cổng vào (*input gate layer*) quyết định giá trị bao nhiêu sẽ được cập nhật. Tiếp theo, tầng ẩn hàm tanh sẽ tạo ra một véc tơ của một giá trị trạng thái mới $\tilde{C}_t$ mà có thể được thêm vào trạng thái. Tiếp theo kết hợp kết quả của 2 tầng này để tạo thành một cập nhật cho trạng thái.

Trong ví dụ của mô hình ngôn ngữ, chúng ta muốn thêm loại của một chủ ngữ mới vào ô trạng thái để thay thế phần trạng thái cũ muốn quên đi.



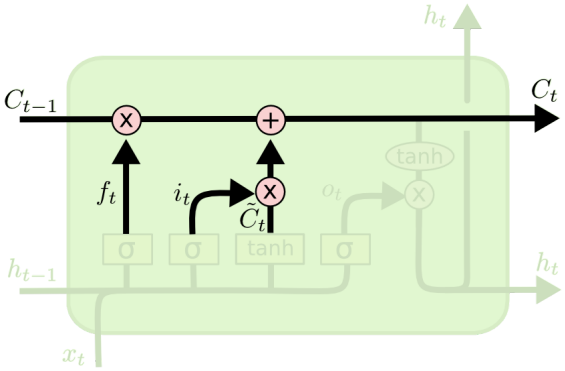
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Hình 9: Cập nhật giá trị cho ô trạng thái bằng cách kết hợp 2 kết quả từ tầng cổng vào và tầng ẩn hàm tanh

Đây là thời điểm để cập nhật một ô trạng thái cũ, C_{t-1} sang một trạng thái mới C_t. Những bước trước đó đã quyết định làm cái gì, và tại bước này chỉ cần thực hiện nó.

Top

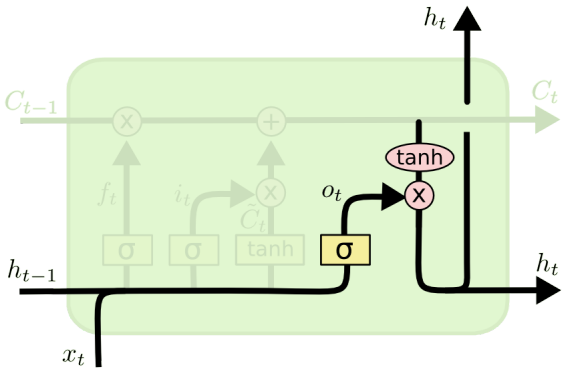
Chúng ta nhân trạng thái cũ với f_t tương ứng với việc quên những thứ quyết định được phép quên sớm. Phần tử đề cử $i_t * \tilde{C}_t$ là một giá trị mới được tính toán tương ứng với bao nhiêu được cập nhật vào mỗi giá trị trạng thái.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Hình 10: Ô trạng thái mới

Cuối cùng cần quyết định xem đầu ra sẽ trả về bao nhiêu. Kết quả ở đầu ra sẽ dựa trên ô trạng thái, nhưng sẽ là một phiên bản được lọc. Đầu tiên, chúng ta chạy qua một tầng sigmoid nơi quyết định phần nào của ô trạng thái sẽ ở đầu ra. Sau đó, ô trạng thái được đưa qua hàm tanh (để chuyển giá trị về khoảng -1 và 1) và nhân nó với đầu ra của một cổng sigmoid, do đó chỉ trả ra phần mà chúng ta quyết định.



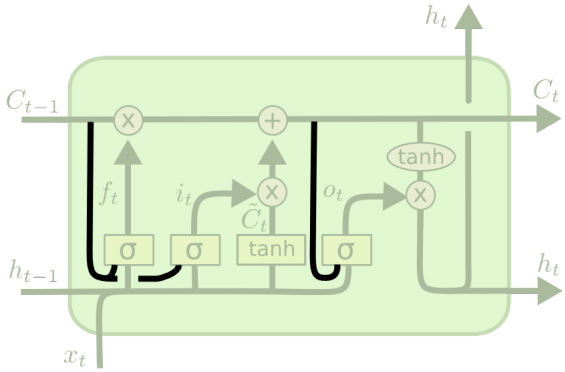
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Hình 11: Điều chỉnh thông tin ở đầu ra thông qua hàm tanh

6. Các biến thể của LSTM

Những gì mà chúng ta vừa mổ tả cho đến giờ là một mạng LSTM rất thông thường. Nhưng không phải toàn bộ LSTM đều tương tự như trên. Trên thực tế, có vẻ như hầu hết mọi bài báo liên quan đến LSTM đều sử dụng những version khác nhau đôi chút. Sự khác biệt là rất nhỏ nhưng rất đáng để đề cập một ít trong số nhứng kiến trúc này.

Một trong những biến thể nổi tiếng nhất của LSTM được giới thiệu bởi Gers & Schmidhuber (2000) (ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf) thêm một kết nối ống tiểu (*peehole connection*) để các cổng có thể kết nối trực tiếp đến các ô trạng thái.

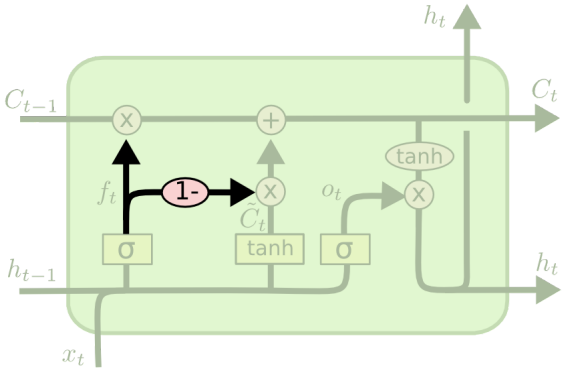


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

Hình 12: Kết nối ống tiểu (*peehole*) liên kết trực tiếp ô trạng thái với các cổng

Một biến thể khác là sử dụng cặp đôi cổng vào và cổng ra. Thay vì quyết định riêng rẽ bỏ qua thông tin nào và thêm mới thông tin nào, chúng ta sẽ quyết định chúng đồng thời. Các thông tin chỉ bị quên khi chúng ta muốn cập nhập vào một vài thông tin mới.

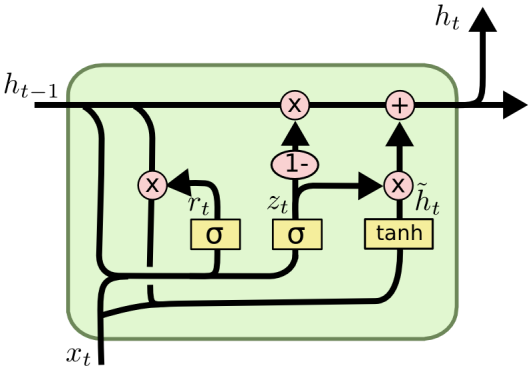
Top



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

Hình 13: Cấu trúc điều chỉnh thêm mới và bỏ qua thông tin đồng thời

Một dạng biến thể khá mạnh khác của LSTM là cổng truy hồi đơn vị (*Gated Recurrent Unit - GRU*) (https://arxiv.org/pdf/1406.1078v3.pdf) được giới thiệu bởi Cho, et al. (2014). Nó kết hợp cổng quên và cổng vào thành một cổng đơn gọi là cập nhật (*update gate*). Nó cũng nhập các ô trạng thái và trạng thái ẩn và thực hiện một số thay đổi khác. Kết quả của mô hình đơn giản hơn nhiều so với mô hình LSTM chuẩn, và đã trở nên khá phổ biến.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Hình 14: Cấu trúc cổng truy hồi đơn vị (*GRU - Gated Recurrent Unit*)

Chỉ có một số lượng nhỏ những biến thể LSTM là đáng chú ý. Rất nhiều những biến thể khác, như kiến trúc cổng sâu RNN (https://arxiv.org/pdf/1508.03790v2.pdf) (*Depth Gated RNN*) của Yao, et al. (2015) hay kiến trúc đồng hồ RNN (https://arxiv.org/pdf/1402.3511v1.pdf) (*Clockword RNN*) của Koutnik, et al. (2014) nhằm giải quyết vấn đề phụ thuộc dài hạn (*long - term depencies*).

Vậy những biến thể nào là tốt nhất? Greff, et al. (2015) thực hiện một so sánh biến thể LSTM (https://arxiv.org/pdf/1503.04069.pdf) và nhận thấy rằng tất cả chúng đều giống nhau. Jozefowicz, et al. (2015) đã thử nghiệm hơn mười nghìn kiến trúc RNN, tìm thấy một số hoạt động tốt hơn LSTM trên một số nhiệm vụ nhất định. Trong khi đó Jozefowicz, et al. (2015) thực hiện kiểm tra trên hơn 1000 kiến trúc RNN (http://proceedings.mlr.press/v37/jozefowicz15.pdf) khác nhau và nhận thấy một số hoạt động tốt hơn so với LSTM trong một vài tác vụ cụ thể.

7. Kết luận

Trước đó, tôi đã đề cập đến những kết quả đáng chú ý mà mọi người đang đạt được với RNNs. Về cơ bản tất cả những điều này đều đạt được bằng cách sử dụng LSTM. Chúng thực sự làm việc tốt hơn rất nhiều cho hầu hết các nhiệm vụ!

Được viết dưới dạng một hệ phương trình, LSTM trông khá là phức tạp và có phần hàn lâm. Thông qua các bước diễn giải tuần tự nguyên lý hoạt động của nó tôi hi vọng sẽ khiến chúng trở nên dễ tiếp cận hơn.

LSTM là một bước đột phá lớn mà ở đó chúng ta đã khắc phục được những hạn chế ở RNN đó là khả năng phụ thuộc dài hạn. Một số kĩ thuật học Attention gần đây được kết hợp với LSTM đã tạo ra những kết quả khá bất ngờ trong các tác vụ dịch máy cũng như phân loại nội dung, trích lọc thông tin,…. Các mô hình dịch máy của google đã ứng dụng kiểu kết hợp này trong các bài toán dịch thuật của mình và đã cải thiện được nội dung bản dịch một cách đáng kể.

Top

8. Thực hành mô hình sinh từ tự động

8.1 Xây dựng mô hình trên level kí tự

Sau đây ta sẽ áp dụng mô hình LSTM trong việc dự báo từ tiếp theo của một đoạn hoặc câu văn dựa vào bối cảnh của từ là những từ liền trước nó.

Dữ liệu được sử dụng là bộ truyện alice ở xứ sở kỳ diệu (https://gist.githubusercontent.com/phillipj/4944029/raw/75ba2243dd5ec2875f629bf5d79f6c1e4b5a8b46/alice_in_wonderland.tx đã được nhà xuất bản publish nên không vi phạm bản quyền. Mô hình dự báo sẽ được xây dựng trên level kí tự. Bên dưới, chúng ta sẽ đọc dữ liệu và chuyển các kí tự về in thường để giảm thiểu kích thước bộ mã hóa mà vẫn đảm bảo được nội dung văn bản. Dữ liệu được lưu trong kernel là file `wonderland.txt` .

```
import numpy
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
import os

filename = '../input/wonderland.txt'
raw_text = open(filename).read().lower()
```

Một dictionary gồm 59 kí tự được sử dụng để mã hóa các kí tự trong bộ truyện. Key của các kí tự là số thứ tự của chúng trong dictionary. Trong sơ đồ thiết kế mạng nơ ron một kí tự được mã hóa bằng một vector đơn vị sao cho phần từ 1 sẽ xuất hiện tại vị trí của key trong bộ từ điển và 0 là các phần tử còn lại.

```
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in enumerate(chars))
print('number of letters: ', len(char_to_int))
print(char_to_int)
```

```
number of letters:  59
{'\n': 0, ' ': 1, '!': 2, '"': 3, '#': 4, '$': 5, '%': 6, "'": 7, '(': 8, ')': 9, '*'
```

Chúng ta nhận thấy rằng mục đích chỉ là dự báo từ tiếp theo do đó cần lọc bỏ những kí tự không quyết định đến nghĩa của 1 từ chẳng hạn như các dấu đặc biệt `#, $, *, @, /` . Như vậy, sẽ cần một bước chuẩn hóa dữ liệu nhằm giảm thiểu nhiễu và số lượng các khả năng có thể ở đầu ra. Điều này sẽ giúp cải thiện chất lượng và độ chính xác trong dự báo của mô hình đáng kể. Việc chuẩn hóa sẽ bao gồm như sau:

1. Chỉ giữ lại các kí tự chữ cái vì chúng có ảnh hưởng đến nội dung của 1 từ.
2. Chỉ giữ lại các dấu câu là `., !, ?` vì chúng thể hiện các loại câu khác nhau và sẽ ảnh hưởng đến từ tiếp theo khi dự báo. Chẳng hạn nếu dấu câu là `?` thì khả năng cao từ tiếp theo sẽ là `yes` hoặc `no` . Dấu câu là `.` thì từ tiếp theo có thể là một đại từ nhân xưng `i, you, we, they, he, she, it` .
3. Giữ lại các dấu `,' '` vì chúng giúp tách các từ và tách các thành phần câu.
4. Chuẩn hóa lại các các chữ số về 1 chữ số duy nhất là 0 vì các con số là ngẫu nhiên và không dự báo được. Chúng ta chỉ có thể dự báo ở vị trí nào có khả năng là số.
5. Các kí tự nằm ngoài số liệt kê trên sẽ đưa vào nhóm `unk` tức unknown.

```
import string
string.ascii_lowercase
# string.digits
# string.punctuation
chars_new = list(string.ascii_lowercase) + ['0', '.', ',', ' ', '!', '?', 'unk']
chars_to_int = dict((v, k) for k, v in enumerate(chars_new))
int_to_chars = dict((k, v) for k, v in enumerate(chars_new))
print('character to int:', chars_to_int)
print('int to character:', int_to_chars)
# def _clean_char(text):
#     return 1
```

Top

```
character to int: {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5, 'g': 6, 'h': 7, 'i
int to character: {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h', 8:
```

```
n_chars = len(raw_text)
n_vocab = len(chars_new)
print('Total characters: ', n_chars)
print('Total Vocab: ', n_vocab)
```

```
Total characters:  163693
Total Vocab:  33
```

Như vậy sau chuẩn hóa văn bản của chúng ta sẽ bao gồm 163693 từ và 33 kí tự. Tiếp theo là một hàm chuyển hóa một câu thành một vector chỉ số các kí tự.

```
def _encode_sen(text):
    text = text.lower()
    sen_vec = []
    for let in text:
        if let in chars_new[:-1]:
            idx = chars_to_int[let]
        else:
            idx = chars_to_int['unk']
        sen_vec.append(idx)
    return sen_vec

x_test = _encode_sen('Alice is a wonderful story. #')
print(x_test)
```

```
[0, 11, 8, 2, 4, 29, 8, 18, 29, 0, 29, 22, 14, 13, 3, 4, 17, 5, 20, 11, 29, 18, 19, 1
```

```
def _decode_sen(vec):
    text = []
    for i in vec:
        let = int_to_chars[i]
        text.append(let)
    text = ''.join(text)
    return text

_decode_sen(x_test)
```

```
'alice is a wonderful story. unk'
```

Để đồng nhất độ dài đầu vào cho mô hình cần tạo ra các chuỗi kí tự (window input) với với độ dài là 100. Mục đích của chúng ta là dự báo kí tự tiếp theo từ 100 kí tự đầu vào. Mỗi một phiên dự báo window input sẽ được tịnh tiến lên 1 kí tự để thu được các kí tự dự báo liên tiếp nhau và từ đó ghép lại thành một câu hoàn chỉnh.

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 1):
    # Lấy ra 100 kí tự liền trước
    seq_in = raw_text[i:i + seq_length]
    # Lấy ra kí tự liền sau 100 kí tự đó
    seq_out = raw_text[i + seq_length]
    dataX.append(_encode_sen(seq_in))
    dataY.append(_encode_sen(seq_out)[0])
    n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
```

Top

```
Total Patterns:  163593
```

Để có thể đưa vào mô hình LSTM, đầu vào `X` cần được chuẩn hóa thành một ma trận 3 chiều `samples, time steps, features` . Trong đó:

1. samples: Số lượng mẫu đầu vào (tức số lượng cửa sổ window 100 length).
2. time steps: Độ dài của cửa sổ window chính là số lượng các vòng lặp khi được trải phẳng ở hình 2 cấu trúc trải phẳng mạng nơ ron. Trong mô hình này time steps = 100.
3. features: Số lượng chiều được mã hóa của đầu vào. Trong mô hình LSTM, mỗi một từ hoặc kí tự (tùy theo chúng ta làm việc với level nào) thường được mã hóa theo 2 cách thông thường sau đây:
    - mã hóa theo one-hot encoding để một kí tự (ở bài thực hành này là kí tự) được biểu diễn bởi một véc tơ one-hot.
    - mã hóa theo giá trị véc tơ được lấy từ mô hình word embedding pretrain trước đó. Có thể là word2vec, fasttext, ELMo, BERT,…. Số lượng các chiều theo level kí tự thường ít hơn so với level từ. Trong bài này để đơn giản ta không sử dụng một lớp embedding ở đầu để nhúng từ thành véc tớ mà sử dụng trực tiếp giá trị đầu vào là index của kí tự. Do đó số features = 1.

```
# reshape X to be [samples, time steps, features]
X_train = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X_train = X_train / float(n_vocab)
# one hot encode the output variable
y_train = np_utils.to_categorical(dataY)
print('X [samples, time steps, features] shape: ', X_train.shape)
print('Y shape: ', y_train.shape)
```

```
X [samples, time steps, features] shape:  (163593, 100, 1)
Y shape:  (163593, 33)
```

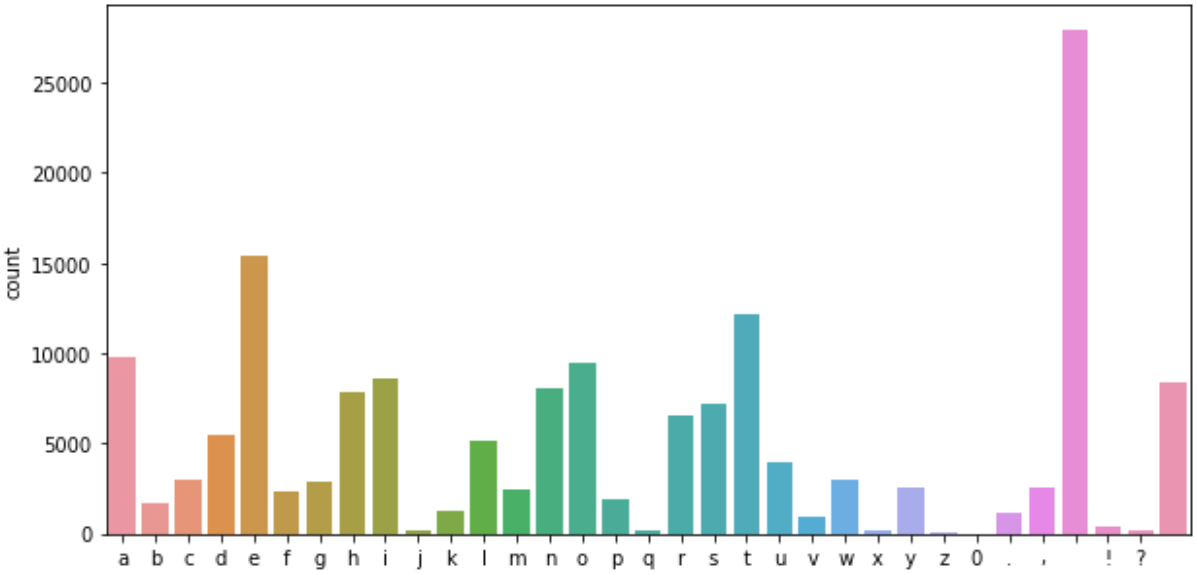
```
print(type(X_train))
print(type(y_train))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

Thống kê số lượng các kí tự theo nhóm.

```
import seaborn as sn
import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize = (10, 5))
sn.countplot(np.array(dataY))
plt.xticks(np.arange(32),np.array(chars_new))
```

Top



8.2. Mô hình LSTM

Xây dựng một kiến trúc model gồm một layer LSTM kết nối tới 1 layer Dropout và kết nối tới Dense layer ở cuối. Với mục đích là để chúng ta hiểu về lý thuyết LSTM nên tôi chọn mô hình có cấu trúc đơn giản nhất.

```
model = Sequential()
model.add(LSTM(256, input_shape = (X_train.shape[1], X_train.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1], activation = 'softmax'))
model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accu
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_5 (LSTM)                (None, 256)               264192
_________________________________________________________________
dropout_5 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_5 (Dense)              (None, 33)                8481
=================================================================
Total params: 272,673
Trainable params: 272,673
Non-trainable params: 0
_________________________________________________________________
```

```
filepath = 'weights-improvement-{epoch:02d}-{loss:.4f}.hdf5'
checkpoint = ModelCheckpoint(filepath, monitor = 'val_acc', verbose = 1, save_best_on
callback_list = [checkpoint]
```

```
model.fit(X_train, y_train, epochs = 5, batch_size = 128, validation_split=0.33, call
```

Top

```
Train on 109607 samples, validate on 53986 samples
Epoch 1/5
109607/109607 [==============================] - 408s 4ms/step - loss: 2.8991 - acc:

Epoch 00001: val_acc improved from -inf to 0.18686, saving model to weights-improvem
Epoch 2/5
109607/109607 [==============================] - 406s 4ms/step - loss: 2.7600 - acc:

Epoch 00002: val_acc improved from 0.18686 to 0.21489, saving model to weights-impro
Epoch 3/5
109607/109607 [==============================] - 404s 4ms/step - loss: 2.6864 - acc:

Epoch 00003: val_acc improved from 0.21489 to 0.21520, saving model to weights-impro
Epoch 4/5
109607/109607 [==============================] - 405s 4ms/step - loss: 2.6287 - acc:

Epoch 00004: val_acc improved from 0.21520 to 0.23499, saving model to weights-impro
Epoch 5/5
109607/109607 [==============================] - 403s 4ms/step - loss: 2.5696 - acc:

Epoch 00005: val_acc improved from 0.23499 to 0.24291, saving model to weights-impro




<keras.callbacks.History at 0x7fa09d77fd30>
```

Dự báo kết quả từ tiếp theo từ một tập hợp kí tự đầu vào.

```
import numpy as np
base_word = 'Alice was beginning to get very tired of sitting by her sister on the b

def _predict_let(text, len_sen = 1):
    text_for = []
    for i in range(len_sen):
        x_input = np.array(_encode_sen(text)[-100:])/float(n_vocab)
        if x_input.shape[0] < 100:
            x_input = np.concatenate((np.zeros(100-x_input.shape[0]), x_input), axis
        x_input = np.expand_dims(np.expand_dims(x_input, -1), 0)
        y_prob = model.predict(x_input)
        y_let = int_to_chars[np.argmax(y_prob, axis = 1)[0]]
        text = text + y_let
    return text[len_sen:]

_predict_let(base_word, 100)
```

```
'nd the  and the  and the  and the  and the  and the  and the  and the  and t'
```

8.3. Kiến trúc BiLSTM.

Bên dưới là kiến trúc mạng LSTM 2 chiều có khả năng đọc đầu vào theo chiều từ trái qua phải và từ phải qua trái.

Top

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Dropout
from keras.layers import LSTM, Input, Bidirectional
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras.metrics import categorical_accuracy

#import spacy, and spacy english model
# spacy is used to work on text
import spacy
nlp = spacy.load('en')

#import other libraries
import numpy as np
import random
import sys
import os
import time
import codecs
import collections
from six.moves import cPickle

#define parameters used in the tutorial
data_dir = '../input'# data directory containing raw texts
# save_dir = 'save' # directory to store trained NN models
file_list = os.listdir('../input')
vocab_file = os.path.join("words_vocab.pkl")
sequences_step = 1 #step to create sequences
```

```
def create_wordlist(doc):
    wl = []
    for word in doc:
        if word.text not in ("\n","\n\n",'\u2009','\xa0'):
            wl.append(word.text.lower())
    return wl

wordlist = []

for file_name in file_list:
    input_file = os.path.join(data_dir, file_name)
    #read data
    with codecs.open(input_file, "r") as f:
        data = f.read()

    #create sentences
    doc = nlp(data)
    wl = create_wordlist(doc)
    wordlist = wordlist + wl
```

Top

```
batch_size = 32 # minibatch size
num_epochs = 50 # number of epochs

callbacks=[EarlyStopping(patience=4, monitor='val_loss'),
           ModelCheckpoint(filepath=save_dir + "/" + 'my_model_gen_sentences.{epoch:
                           monitor='val_loss', verbose=0, mode='auto', period=2)]
#fit the model
history = md.fit(X, y,
                 batch_size=batch_size,
                 shuffle=True,
                 epochs=num_epochs,
                 callbacks=callbacks,
                 validation_split=0.1)

#save the model
md.save(save_dir + "/" + 'my_model_generate_sentences.h5')batch_size = 32 # minibatc
num_epochs = 50 # number of epochs

callbacks=[EarlyStopping(patience=4, monitor='val_loss'),
           ModelCheckpoint(filepath=save_dir + "/" + 'my_model_gen_sentences.{epoch:
                           monitor='val_loss', verbose=0, mode='auto', period=2)]
#fit the model
history = md.fit(X, y,
                 batch_size=batch_size,
                 shuffle=True,
                 epochs=num_epochs,
                 callbacks=callbacks,
                 validation_split=0.1)

#save the model
md.save(save_dir + "/" + 'my_model_generate_sentences.h5')batch_size = 32 # minibatc
num_epochs = 50 # number of epochs

callbacks=[EarlyStopping(patience=4, monitor='val_loss'),
           ModelCheckpoint(filepath=save_dir + "/" + 'my_model_gen_sentences.{epoch:
                           monitor='val_loss', verbose=0, mode='auto', period=2)]
#fit the model
history = md.fit(X, y,
                 batch_size=batch_size,
                 shuffle=True,
                 epochs=num_epochs,
                 callbacks=callbacks,
                 validation_split=0.1)

#save the model
md.save(save_dir + "/" + 'my_model_generate_sentences.h5')len(wordlist[3]
```

```
# count the number of words
word_counts = collections.Counter(wordlist)

# Mapping from index to word : that's the vocabulary
vocabulary_inv = [x[0] for x in word_counts.most_common()]
vocabulary_inv = list(sorted(vocabulary_inv))

# Mapping from word to index
vocab = {x: i for i, x in enumerate(vocabulary_inv)}
words = [x[0] for x in word_counts.most_common()]

#size of the vocabulary
vocab_size = len(words)
print("vocab size: ", vocab_size)

#save the words and vocabulary
with open(os.path.join(vocab_file), 'wb') as f:
    cPickle.dump((words, vocab, vocabulary_inv), f)
```

Top

```
#create sequences
sequences = []
next_words = []
for i in range(0, len(wordlist) - seq_length, sequences_step):
    sequences.append(wordlist[i: i + seq_length])
    next_words.append(wordlist[i + seq_length])

print('nb sequences:', len(sequences))
```

```
X = np.zeros((len(sequences), seq_length, vocab_size), dtype=np.bool)
y = np.zeros((len(sequences), vocab_size), dtype=np.bool)
for i, sentence in enumerate(sequences):
    for t, word in enumerate(sentence):
        X[i, t, vocab[word]] = 1
    y[i, vocab[next_words[i]]] = 1
```

```
print('X shape: ', X.shape)
print('y shape: ', y.shape)
```

```
 def bidirectional_lstm_model(seq_length, vocab_size):
     print('Build LSTM model.')
     model = Sequential()
     model.add(Bidirectional(LSTM(rnn_size, activation="relu"),input_shape=(seq_lengt
     model.add(Dropout(0.6))
     model.add(Dense(vocab_size))
     model.add(Activation('softmax'))

     optimizer = Adam(lr=learning_rate)
     callbacks=[EarlyStopping(patience=2, monitor='val_loss')]
     model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=[cat
     print("model built!")
     return model

 rnn_size = 256 # size of RNN
 seq_length = 100 # sequence length
 learning_rate = 0.001 #learning rate

 md = bidirectional_lstm_model(seq_length, vocab_size)
 md.summary()
```

```
batch_size = 32 # minibatch size
num_epochs = 50 # number of epochs

callbacks=[EarlyStopping(patience=4, monitor='val_loss'),
           ModelCheckpoint(filepath="/" + 'my_model_gen_sentences.{epoch:02d}-{val_l
                           monitor='val_loss', verbose=0, mode='auto', period=2)]
#fit the model
history = md.fit(X, y,
                 batch_size=batch_size,
                 shuffle=True,
                 epochs=num_epochs,
                 callbacks=callbacks,
                 validation_split=0.1)

#save the model
md.save("/" + 'my_model_generate_sentences.h5')
```

9. Tài liệu

Top

1. Understanding - LSTMs - Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs)

2. Leanring Long - Term Dependencies with Gradient Descent is Difficult (http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf)
3. (*Gated Recurrent Unit - GRU*) (https://arxiv.org/pdf/1406.1078v3.pdf)
4. so sánh biến thể LSTM (https://arxiv.org/pdf/1503.04069.pdf)
5. kiến trúc cổng sâu RNN (*Depth Gated RNN*) (https://arxiv.org/pdf/1508.03790v2.pdf)

Top