Bài 23 - Neural Attentive Session-Based Recommendation

11 Feb 2020 - phamdinhkhanh

Menu

1. Các dạng mô hình Recommendation

Ngày nay, các thuật toán recommendation ngày càng phát triển đa dạng bởi tính ứng dụng cao trong việc khuyến nghị và matching sản phẩm tới khách hàng trên các nền tảng website/app kinh doanh online. Recommendation có thể coi là vũ khí trong cuộc chơi của các ông lớn trong những ngành có tính đốt tiền cao như ecommerce, airbnb, logistic,…. Do đó đây là bài toán thu hút được rất nhiều sự đầu tư và nghiên cứu từ những trường đại học và từ các tập đoàn công nghệ. Facebook, Amazon, Google, Alibaba,… có thể coi là những nền tảng có hệ thống recommendation phát triển nhất. Recommendation từ những nền tảng này không những đạt độ chính xác cao mà còn đáp ứng nhu cầu đồng thời của một lượng lớn người dùng online. Các kết

Top

quả khuyến nghị từ google, facebook rất tốt là điều không quá ngạc nhiên (chỉ khi nó không tốt mới đáng ngạc nhiêu) bởi đây là những tập đoàn có dữ liệu lớn và có đội ngũ nghiên cứu hàng đầu thế giới.

Tuy nhiên các doanh nghiệp vừa và nhỏ khác cũng hoàn toàn có thể tự xây dựng và phát triển một hệ thống recommendation từ chính dữ liệu về hành vi người dùng và hệ thống cây sản phẩm. Hãy tin vào điều đó vì lý thuyết thuật toán recommendation là không quá phức tạp và tài liệu các thuật toán recommendation là sẵn có.

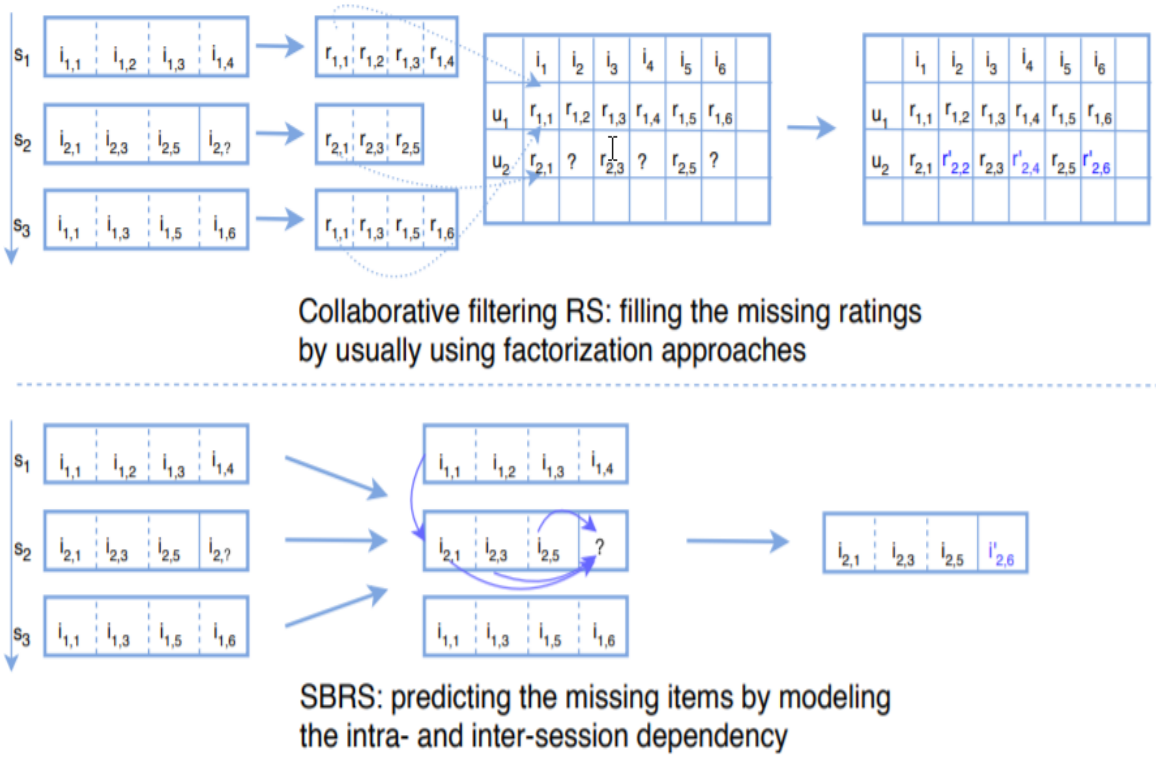
Ở Bài 15 - collaborative và content-based filtering (https://phamdinhkhanh.github.io/2019/11/04/Recommendation_Compound_Part1.html) và Bài 20 - Recommendation Neural Network (https://phamdinhkhanh.github.io/2019/12/26/Sorfmax_Recommendation_Neural_Network.html) tôi đã giới thiệu tới các bạn các phương pháp recommendation nhằm tìm kiếm biểu diễn của một item cụ thể. Những bài toán này phù hợp với dữ liệu không có yếu tố thời gian. Tuy nhiên theo xu hướng nghiên cứu ở thời điểm hiện tại, những lớp mô hình state-of-art được phát triển để phù hợp với dữ liệu dạng session có yếu tố thời gian, rất phổ biến trong các nền tảng website/app hiện đại. Chúng ta cùng tìm hiểu một lớp thuật toán mới là `Session-Based Recommendation` mà tôi sẽ giới thiệu qua bài viết này.

2. Khác biệt giữa Collaborative-Filtering và Session-Based System

Cả 2 thuật toán Collaborative-Filtering và Session-Based System đều dựa trên các lượt tương tác giữa user và item. Tuy nhiên thực tiễn cho thấy Collaborative-Filtering tồn tại những hạn chế đó là hành vi rating của người dùng không được đặt trong ngữ cảnh (context). Tức là mối liên hệ của một lượt rating với các lượt rating liền kề về mặt thời gian không được quan tâm.

Nhưng bối cảnh lại rất có ảnh hưởng tới sở thích của khách hàng. Chẳng hạn như khách hàng trước đó đã từng xem một bộ phim bom tấn. Do bộ phim đó quá hay nên mặc dù bộ phim hiện tại rất xuất sắc nhưng vì ảnh hưởng so sánh làm cho khách hàng vẫn có xu hướng đánh giá thấp bộ phim hiện tại. Mọi người thường nói không nên so sánh nhưng có vẻ đây là một phản chứng :D. Bối cảnh trong trường hợp này đóng vai trò rất lớn nhưng không được đưa vào mô hình Collabrative Filtering. Chính vì hạn chế đó, các mô hình Session-Based System đã khắc phục bằng cách đưa thêm yếu tố bối cảnh vào mô hình nhằm mang lại độ chính xác cao hơn.

Top



Hình 1: So sánh giữa mô hình Collaborative-Filtering và mô hình Session- Based System. Dữ liệu đầu vào là các session của khách hàng 1 (dòng 1 và 3) và khách hàng 2 (dòng 2). Ở nửa phía trên của hình minh họa là mô hình Collaborative Filtering, các lượt rating được sắp xếp từ các session theo đúng điểm rating của user với item. Cuối cùng ta thu được một ma trận tiện ích (utility) của user-item như vị trí thứ 3 từ trái sang. Dựa trên các lượt rating đã biết ta cần dự báo các vị trí chưa biết (dấu ?). Tiếp theo bên dưới hình minh họa là mô hình Session-Based Recommendation System. Theo mô hình này các đầu vào là những items được giữ nguyên vị trí như trong session gốc. Chúng ta cần dự đoán sản phẩm mà tiếp theo khách hàng có khả năng tương tác.

Khái niệm về session có thể được định nghĩa linh hoạt. Đó có thể là toàn bộ chuỗi các sản phẩm mà khách hàng đã view. Trường hợp khác sẽ giới hạn session chỉ bao gồm danh sách các sản phẩm có trong giỏ hàng hoặc cũng có thể là danh sách sản phẩm khách hàng đã thanh toán trong một lần. Thậm chí session cũng có thể mở rộng thành list các event trong một khoảng thời gian ngắn của một khách hàng (có thể thuộc các sessions khác nhau).

Mô hình Collaborative-Filtering gặp hạn chế ở một số tình huống user không đăng nhập, hệ thống sẽ bị giới hạn trong việc thu thập thông tin người dùng và do đó chúng ta không thể biết khách hàng là ai để recommend chính xác các sản phẩm họ có nhu cầu. Trong khi đó mô hình Session Based System lợi thế hơn khi tìm kiếm kiểu (pattern) của chuỗi hành vi mua sắm từ các session và khuyến nghị theo kiểu mà không quan tâm đến định danh user.

Ngoài ra Session-Based System không chia nhỏ các session của một user thành những cặp đánh giá (user, item) không theo thứ tự như Collaborative-Filtering nên bảo toàn được mối liên hệ về bối cảnh của session, một yếu tố có ảnh hưởng lớn đến hành vi của user. Top

2.1. Các dạng của Session-Based System

Trong chương này chúng ta sẽ tìm cách phân loại Session-Based System theo 2 nhánh chính: Tiếp cận theo khía cạnh mô hình (model-based) và tiếp cận phi mô hình (model-free). Trong mỗi cách tiếp cận sẽ có một vài lớp mô hình khác nhau được trình bày cụ thể bên dưới:

2.1.1. Tiếp cận phi mô hình (model-free)

Các lượt khuyến nghị của mô hình tự do chủ yếu được xây dựng dựa trên các kĩ thuật khai phá dữ liệu như thống kê tần suất, tính độ tương quan và không đòi hỏi các thuật toán phức tạp. Có 2 nhánh tiếp cận chủ yếu là Pattern/rule-based RS và Sequential Pattern-based RS cho các session thứ bậc.

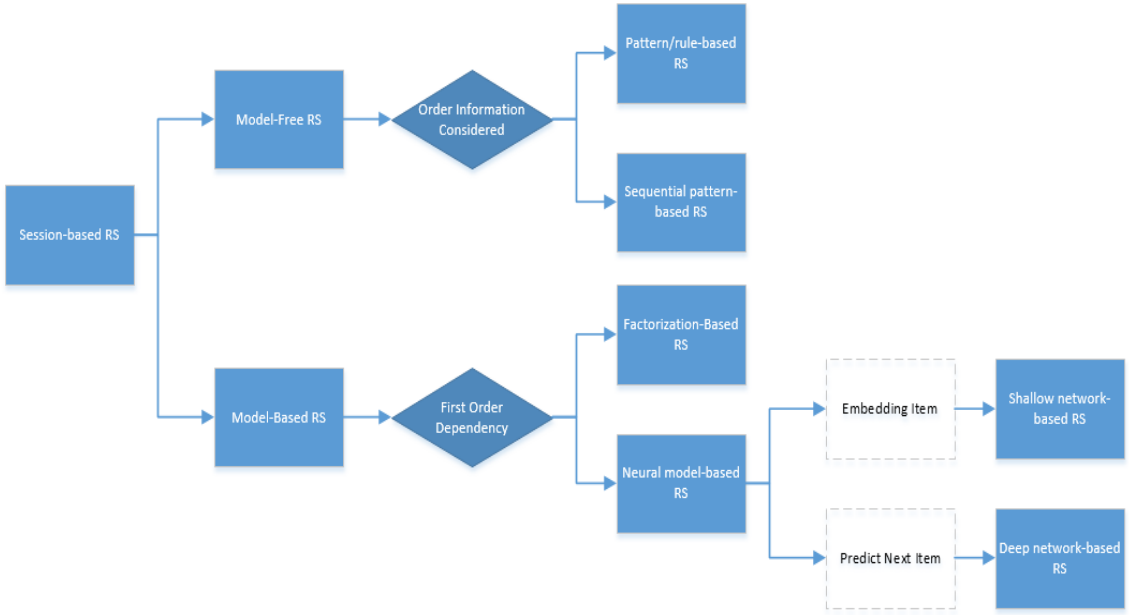
1. **Pattern/Rule-based**: Theo cách tiếp cận Pattern/Rule-based, mô hình sẽ dựa trên giải định hầu hết các lượt mua sắm của khách hàng sẽ có chung những kiểu (pattern) mua sắm. chúng ta sẽ phân tích tần suất kiểu hoặc các qui luật kết hợp và sử dụng các kiểu và qui luật kết hợp này để đưa ra khuyến nghị cho các lượt mua sắm tiếp theo. Chẳng hạn như khách hàng khi mua bánh mì sẽ thường mua kèm sữa nên ta sẽ coi `{bánh mì, sữa}` là một kiểu và áp dụng vào khuyến nghị. Dữ liệu đầu vào của mô hình là dữ liệu không có thứ tự.
2. **Sequential Pattern-based**: Cách tiếp cận chuỗi Pattern-based nhằm giải quyết những dữ liệu mà có tính thứ tự item (dữ liệu dạng chuỗi thời gian). Cũng tương tự như phương pháp `Pattern/Rule-based` , chúng ta cũng phân tích một tập hợp các kiểu mua sắm nhưng dưới dạng chuỗi và sau đó khuyến nghị những items còn lại diễn ra sau đó.

2.1.2. Tiếp cận mô hình (model-based)

Là phương pháp phức tạp hơn, tiếp cận theo mô hình sẽ yêu cầu xây dựng thuật toán dựa trên giải thuyết ngặt đó là dữ liệu có tính thời gian và tính thứ tự chẳng hạn như mô hình Markov Chain. Các cách tiếp cận của phương pháp mô hình cơ sở chủ yếu được phân thành 3 loại chính gồm: mô hình Markov Chain, mô hình phân tích nhân tố (factorization model) và mô hình mạng neural.

1. **Mô hình Markov Chain**: Mô hình hồi qui sự phụ thuộc của bậc 1 qua toàn bộ chuỗi các items bằng việc sử dụng sự dịch chuyển xác suất và sau đó tạo ra các khuyến nghị của các items theo sau đó bằng việc tối ưu hóa các sự phụ thuộc này. Sự khác biệt so với phương pháp chuỗi Pattern-based đó là mô hình dễ dàng lọc ra những pattern và items không thường xuyên và dẫn tới sự mất mát về thông tin. Markov Chain sẽ lấy toàn bộ các đầu vào và do đó làm giảm mất mát thông tin đáng kể.
2. **Mô hình phân tích nhân tố**: Phương pháp này gần giống với Bài 3 - Mô hình Word2Vec (https://phamdinhkhanh.github.io/2019/04/29/ModelWord2Vec.html). Những cách tiếp cận này sẽ phân tích suy biến ma trận đồng xuất hiện hoặc ma trận dịch chuyển item-to-item thành những véc tơ nhân tố ẩn đại diện cho mỗi item và sau đó dự báo các items theo sau đó bằng cách sử dụng những biểu diễn nhân tố ẩn. Cần phân biệt các cách tiếp cận này với các phương pháp phân tích suy biến ma trận trong collaborative-filtering khi chúng sử dụng đầu vào là ma trận tương tác user-item thay vì item-to-item. Các véc tơ nhân tố ẩn của users và items đồng thời cũng được xác định một cách riêng biệt.
3. **Mô hình mạng neural**: Mô hình này tận dụng những lợi thế của mô hình mạng neural để học những mối liên hệ phức tạp giữa những items trong cùng một session. Mô hình có thể là những mô hình học nông (shallow network) chỉ với vài layer như trong bài Bài 20 Top Recommendation Neural Network

(https://phamdinhkhanh.github.io/2019/12/26/Sorfmax_Recommendation_Neural_Network.html) nhằm tìm ra biểu diễn nhúng của các item hoặc mô hình học sâu (deep network) như các lớp mô hình RNN sẽ được giới thiệu ở phần thực hành.



Hình 2: Sơ đồ mô hình Recommendation System theo session based.

2.1.3. So sánh các phương pháp tiếp cận.

Nói chung, phương pháp phi mô mô hình rất đơn giản, dễ hiểu và dễ thực hiện. Bởi vì cả hai cách tiếp cận Pattern/Rule-based and Sequential Pattern-based chỉ dựa trên tần suất, chúng rất dễ lọc ra các items hoặc patterns không xuất hiện thường xuyên nhưng quan trọng. Do đó, các phương pháp phi mô hình phù hợp với khám phá mối quan hệ giữa các item thường xuyên trong bộ dữ liệu đơn giản. Tuy nhiên, chúng có thể dễ dàng thất bại trong việc mô hình hóa sự phụ thuộc phức tạp trong các bộ dữ liệu phức tạp cho các khuyến nghị dựa trên session.

Ngược lại, cách tiếp cận dựa trên mô hình phức tạp hơn nhiều nhưng có sức mạnh đối với các bộ dữ liệu phức tạp. Một mặt, chúng không lọc rõ ràng các item hoặc patterns từ các bộ dữ liệu, nhưng giữ lại thông tin ở mức tối đa. Mặt khác, nhờ sự phức tạp của các cơ chế như mạng học sâu neural networks, các phương pháp dựa trên lớp mô hình này có thể mô hình hóa quan hệ phức tạp và tiềm ẩn trong dữ liệu, dẫn đến các khuyến nghị dựa trên session tốt hơn.

Trong phần thực hành bài này tôi sẽ tiếp cận Session-Based System theo phương pháp mô hình và sử dụng một kiến trúc học sâu để dự báo item tiếp theo trong một chuỗi session. Thuật toán được sử dụng có tên là `Neural Attentive Session-based Recommendation Model` (được viết tắt trong bài là NASM). Một thuận toán kết hợp giữa kiến trúc mạng GRU trong RNN với cơ chế attention. Chúng ta sẽ tìm hiểu về thuật toán này bên dưới.

3. Lý thuyết mô hình GRU

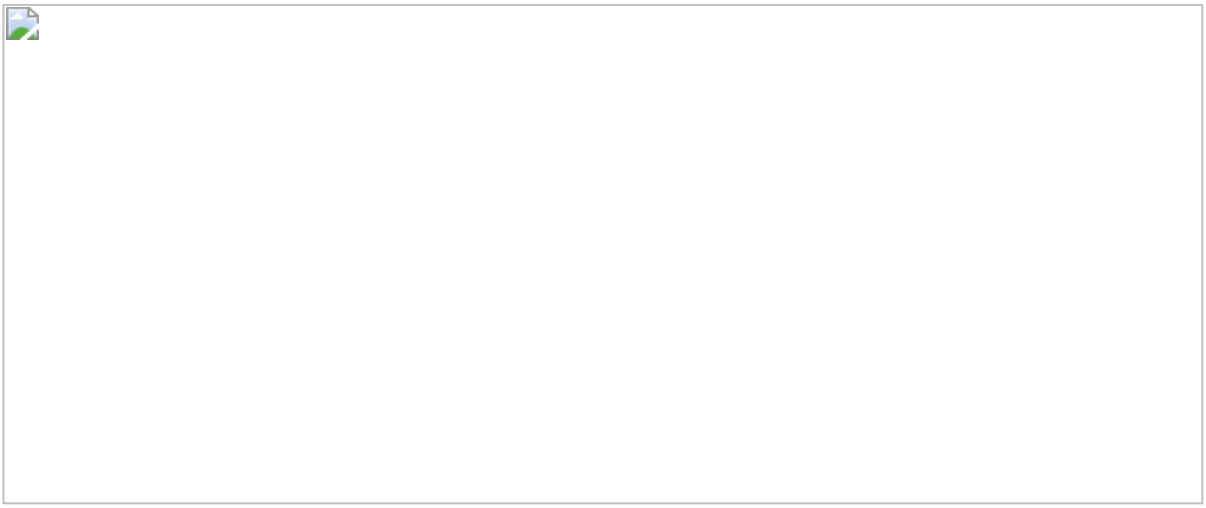
Như chúng ta đã biết, một trong những ưu điểm của mô hình RNN là có khả năng ghi nhớ được sự phụ thuộc chuỗi, do đó rất phù hợp để áp dụng vào mô hình session-based recommendation system. Tuy nhiên hạn chế của RNN đó là sự phụ thuộc dài hạn yếu và thường xảy ra hiện tượng triệt tiêu đạo hàm. GRU là một kiến trúc của RNN cho phép giải quyết được vấn đề triệt tiêu đạo hàm nhờ cơ chế thêm hoặc bớt thông tin thông qua các cổng cập nhật, tái thiết lập. Một lớp mô

Top

hình khác là LSTM cũng giải quyết được vấn đề triệt tiêu đạo hàm. Tuy nhiên các thử nghiệm giữa 2 mô hình LSTM và GRU áp dụng trong kiến trúc mô hình recommendation này đều cho thấy kết quả của GRU là vượt trội hơn.

Kiến trúc mô hình GRU

Có rất nhiều kiến trúc khác nhau của mạng RNN như LSTM, GRU. Nhìn chung kiến trúc của chúng đều là một chu kì tuần hoàn gồm các layers nối tiếp nhau. Sự khác biệt giữa các mạng được thể hiện qua kiến trúc chi tiết trong từng layer, bạn đọc có thể xem thêm Bài 2 - Lý thuyết về mạng LSTM (https://phamdinhkhanh.github.io/2019/04/22/Ly_thuyet_ve_mang_LSTM.html). Ở bài này tôi sẽ áp dụng mô hình GRU vào xây dựng một hệ thống recommendation system. GRU là một trong những kiến trúc giải quyết được tốt vấn đề triệt tiêu gradient (vanishing gradient problem) nhờ các `cổng cập nhật` (update gate) và `cổng tái thiết lập` (reset gate). Hai cổng này giúp điều chỉnh thông tin để quyết định xem thông tin nào được truyền qua chúng. Chính vì thế chúng có thể giữ được thông tin qua một số lượng quãng thời gian rất dài mà không mất đi thông tin quan trọng và xóa đi những thông tin không liên quan. Cụ thể về những cổng này sẽ được diễn giải như sau:



Hình 3: Sơ đồ một GRU layer

Trong hình vẽ trên:

- Dấu $+$ kí hiệu cho phép cộng véc tơ.
- Dấu σ kí hiệu cho hàm sigmoid có giá trị như sau: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Dấu $\odot$ kí hiệu cho tích hadamard của các véc tơ. $\mathbf{u} \odot \mathbf{v} = (u_1v_1, u_2v_2, \ldots, u_nv_n)$

Tại mỗi time step sẽ nhận đầu vào là các hidden véc tơ và chính là đầu ra của time step trước kết hợp với véc tơ nhúng của input tại time step đó. Sau khi đi qua một loạt các cổng bên trong layer GRU sẽ trả về kết quả là một hidden véc tơ ở đầu ra. Cụ thể trong layer GRU bao gồm những bước nào sẽ diễn giải bên dưới.

Các bước của GRU layer:

- **Bước 1**: Cổng cập nhật

Đầu tiên GRU layer sẽ quyết định thông tin nào sẽ bị loại bỏ khỏi bộ nhớ bằng `cổng cập nhật`. Hàm σ có giá trị càng lớn tiệm cận 1 thì lượng thông tin cũ được lưu giữ càng lớn và ngược lại. Đầu vào của `cổng cập nhật` là các véc tơ hidden $\mathbf{h}_{t-1}$ và véc tơ nhúng $\mathbf{x}_t$.

$$\mathbf{z}_t = \sigma(\mathbf{W}_z[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

Top

$\mathbf{h}_{t-1}$ là kết quả ouput của layer GRU thứ $t-1$ nên lưu giữ được các thông tin của toàn bộ $t-1$ bước thời gian trước. Thông tin này được kết nối với thông tin ở bước thời gian hiện tại $\mathbf{x}_t$. `Cổng cập nhật` sẽ giúp mô hình xác định được bao nhiêu thông tin trong quá khứ sẽ được chuyển tiếp tới tương lai. Cơ chế cổng cập nhật thực sự rất mạnh, chỉ cần copy toàn bộ lại thông tin trong quá khứ chúng ta sẽ loại bỏ được rủi ro triệt tiêu gradient.

- **Bước 2**: Cổng tái thiết lập

Trái ngược với cổng cập nhật, cổng tái thiết lập sẽ quyết định bao nhiêu thông tin trong quá khứ sẽ bị lãng quên. Chúng ta cũng sử dụng hàm σ để tính toán thông tin bị lãng quên:

$$\mathbf{r}_t = \sigma(\mathbf{W}_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

- **Bước 3**: Nội dung nhớ hiện tại

Sau khi tính toán xong các `cổng cập nhật` và `cổng tái thiết lập` , thông tin mới sẽ được tính toán lại bằng cách kết hợp thông tin từ cổng tái thiết lập và đầu vào tại time step t như sau:

$$\mathbf{h'}_t = \tanh(\mathbf{W}[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t])$$

Gía trị tổng hợp quá khứ $\mathbf{h}_{t-1}$ đã được loại bỏ những thông tin không quan trọng bằng cách nhân với véc tơ tái thiết lập $\mathbf{r}_t$. Việc cập nhật này là cực kì quan trọng, giả sử trong một tác vụ phân loại cảm xúc bình luận có một câu bình luận như sau:

```
"Đây là một bộ phim rất hay."
```

Để biết được nội dung của bình luận trên là tích cực hay tiêu cực ta chỉ cần quan tâm đến từ cuối cùng. Như vậy tại bước thời gian t, ta chỉ cần điều chỉnh véc tơ $\mathbf{r}_t$ về gần 0 để cho các giá trị trước đó bị lãng quên thì giá trị mới sau cập nhật trọng tâm vào giá trị hiện tại $\mathbf{x}_t$.

- **Bước 4**: Trí nhớ cuối cùng tại time step hiện tại

Đây là bước cuối cùng để tính toán ra output chính là véc tơ hidden $\mathbf{h}_t$. Véc tơ hiện tại sẽ tổng hợp thông tin giữa nội dung nhớ hiện tại $\mathbf{h'}_t$ với các thông tin được lưu giữ tại các bước trước đó $\mathbf{h}_{t-1}$ thông qua trọng số hiệu chỉnh là véc tơ cổng cập nhật $\mathbf{z}_t$:

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \mathbf{h'}_t$$

Có thể coi các hệ số $\mathbf{z}_t, (1-\mathbf{z}_t)$ như là các hệ số đánh đổi giữa giá trị quá khứ và giá trị hiện tại. Như vậy muốn lưu giữ nhiều thông tin quá khứ hơn thì thiết lập $\mathbf{z}_t$ gần với 1 hơn và muốn giữ lại nhiều thông tin hiện tại hơn thì thiết lập $\mathbf{z}_t$ gần với 0 hơn.

3.1. Global encoder

Global encoder là quá trình encode các thông tin đầu vào chính là các hidden véc tơ tại mỗi time step. Kết quả sau cùng thu được là một global véc tơ được thiết lập bằng chính hidden véc tơ tại time step cuối cùng như sau:

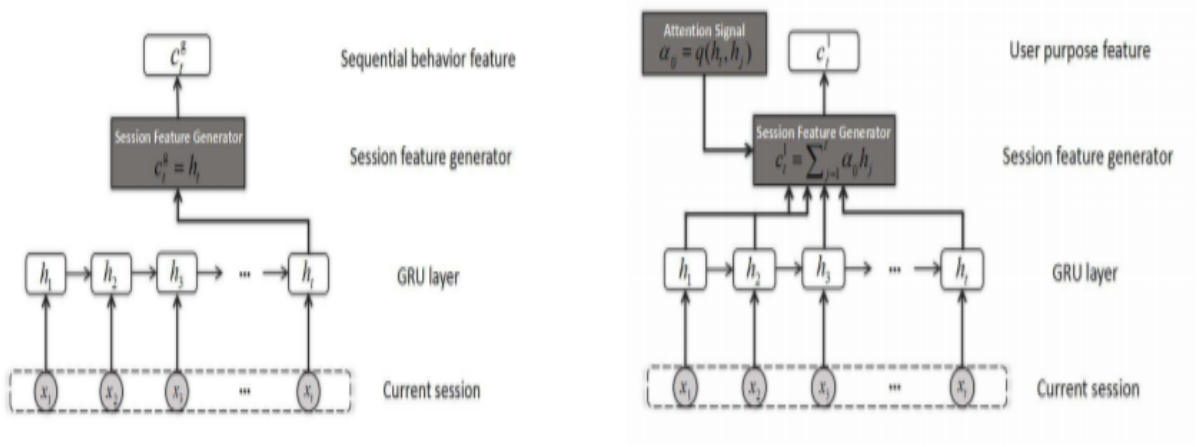
$$\mathbf{c}_t^g = \mathbf{h}_t$$

global véc tơ sẽ có tác dụng tổng hợp lại toàn bộ các thông tin tại các time step trước đó của mô hình. Do đó nó đại diện cho nội dung của toàn bộ input ở pharse encoder. Tuy nhiên $\mathbf{h}_t$ không thể hiện được mối liên hệ cục bộ giữa các inputs tại những time step khác nhau. Mối liên hệ này sẽ được thể hiện thông qua một cơ chế attention được giải thích bên dưới.

3.2. Local encoder

Top

Để ghi nhận được hành vi của người dùng tại các session hiện tại, chúng ta dựa vào một cơ chế attention cho phép lựa chọn các kết hợp tùy ý từ đầu vào tại các time step thông qua tổ hợp tuyến tính của chúng. Về kĩ thuật attention các bạn có thể xem thêm tại Bài 4 - Attention is all you need (https://phamdinhkhanh.github.io/2019/06/18/AttentionLayer.html).



(a) Sơ đồ graphic của global encoder trong NARM, hidden state cuối cùng được mô tả như feature của chuỗi hành vi của user $\mathbf{c}_t^g = \mathbf{h}_t$

(b) Sơ đồ graphic của local encoder trong NARM, ở đây tổng có trọng số của toàn bộ các hidden véc tơ được mô tả như feature ghi nhận hành vi của user $\mathbf{c}_t^l = \sum_{j=1}^t \alpha_{tj}\mathbf{h}_t$

Hình 4: Cơ chế global encoder và local encoder trong mô hình NARM.

Để tính toán được local véc tơ ta cần tính các hệ số attention α_{tj} đại diện cho mức độ phân phối attention của đầu vào tại time step j tới đầu ra tại time step t.

Khác với cơ chế tính trọng số α_{tj} ở attention layer trong Bài 4 (https://phamdinhkhanh.github.io/2019/06/18/AttentionLayer.html) một chút. Chúng ta sẽ không tính α_{tj} trực tiếp từ $\text{score}(\mathbf{h}_t, \mathbf{h}_j)$. Ở đây ta sử dụng một cơ chế phức tạp hơn, bao gồm các bước:

- **Bước 1:** Giảm chiều dữ liệu của $\mathbf{h}_t, \mathbf{h}_j$ bằng một phép chiếu linear projection lên không gian latent bằng cách nhân với các ma trận $\mathbf{A}_1$ và $\mathbf{A}_2$. Khi đó thu được véc tơ latent tổng hợp như sau:

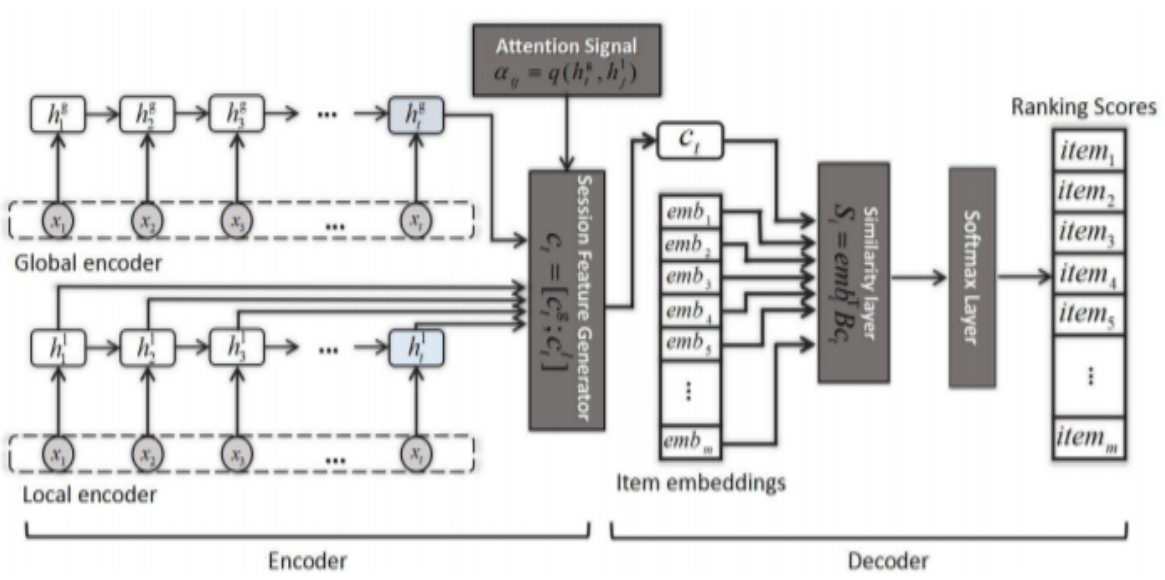
$$\mathbf{l}_{tj} = \mathbf{A}_1\mathbf{h}_t + \mathbf{A}_2\mathbf{h}_j$$

- **Bước 2:** Tính véc tơ phân phối xác suất cho véc tơ latent $\mathbf{l}_{tj}$ bằng cách truyền vào hàm sigmoid. Hệ số α_{tj} bằng tổng tổ hợp tuyến tính của sigmoid với các hệ số của véc tơ $\mathbf{v}$.

$$\alpha_{tj} = \mathbf{v}^\intercal\sigma(\mathbf{l}_{tj})$$

3.3. Kiến trúc mô hình NARM

Top



Hình 5: Sơ đồ mô hình NARM, ở đây session của feature $\mathbf{c}_t$ được biểu diễn bởi véc tơ concatenate $\mathbf{c}_t = [c_t^g; c_t^l]$.

Đối với nhiệm vụ của session-based recommedation, global encoder sẽ thống kê toàn bộ chuỗi hành vi, trong khi local encoder có thể lựa chọn phù hợp một số items quan trọng trong session hiện tại để nắm bắt mục đích chính của user. Chúng ta phỏng đoán rằng biểu diễn của một chuỗi hành vi có thể cung cấp những thông tin hữu ích trong session hiện tại. Do đó, chúng ta sử dụng biểu diễn của chuỗi hành vi và của hidden states liền trước để tính toán attention weight cho mỗi item click. Sau đó một mở rộng tự nhiên kết hợp các chuỗi đặc trưng hành vi và đặc trưng mục đích của user bằng cách concatenate chúng để tạo thành một biểu diễn mở rộng cho mỗi time step.

Như thể hiện trong hình 5, chúng ta có thể nhìn thấy véc tơ tổng hợp $\mathbf{h}_t^g$ được kết hợp vào $\mathbf{c}_t$ để cung cấp một biểu diễn chuỗi hành vi cho mô hình NARM. Do đó $\mathbf{h}_t^g$ được coi là global encoder. Sau đó chúng được sử dụng để tính toán attention weight với những hidden states trước đó. Bằng cơ chế encoding hỗn hợp, cả chuỗi hành vi và mục đích chính của user trong session hiện tại có thể được mô hình vào một biểu diễn thống nhất $\mathbf{c}_t$, nó là kết hợp của các véc tơ $\mathbf{c}_t^g$ và $\mathbf{c}_t^l$ như sau:

$$\mathbf{c}_t = [\mathbf{c}_t^g; \mathbf{c}_t^l] = [\mathbf{h}_t^g; \sum_{j=1}^{t} \alpha_{tj} \mathbf{h}_t^l]$$

Hình 5 cũng đưa ra một sơ đồ mô phỏng cơ chế decoding trong mô hình NARM. Tổng quát hóa, một RNN chuẩn tối ưu hóa fully-connected layer để giải mã. Nhưng sử dụng fully-connected layer có nghĩa rằng số lượng các tham số cần phải được học trong layer này là $H * N$ ở đây H là số chiều của biểu diễn session và N là số item được đề xuất cho dự báo.

Chính vì thế, chúng ta phải dự trữ một không gian lớn để lưu trữ các tham số này. Mặc dù có một vài cách tiếp cận để giảm tham số như sử dụng hierachical softmax layer và negative sampling ngẫu nhiên, chúng không phải là lựa chọn phù hợp nhất cho mô hình của chúng ta.

Một cách thay thế được đề xuất là bi-linear decoding scheme giúp không chỉ giảm số lượng các tham số mà còn cải thiện kết quả cho mô hình NARM. Đặc biệt, một hàm số đo độ tương đương bi-linear giữa các biểu diễn của session hiện tại và mỗi đề cử item được sử dụng để tính toán điểm tương đương Si,

Top

$$S_i = \text{emb}_i^\top \mathbf{B}\mathbf{c}_t$$

Trong đó $\mathbf{B} \subset \mathbb{R}^{D \times H}$, D là chiều của mỗi item embedding. Sau đó điểm tương đương của mỗi item được đưa vào một softmax layer để đạt được xác suất mà item sẽ xảy ra tiếp theo. Bằng sử dụng bi-linear decoder, chúng ta sẽ giảm số lượng các tham số từ $N \times H$ xuống còn $D \times H$, ở đây D nhỏ hơn đáng kể so với N. Ngoài ra, kết quả kiểm nghiệm cho thấy sử dụng bi-linear decoder có thể cải thiện biểu diễn của mô hình NARM.

Để học tham số biểu diễn của mô hình, chúng ta không tối ưu hóa thủ tục huấn luyện được đề xuất, ở đây mô hình được huấn luyện trong một session-parallel, sequence-to-sequence manner. Thay vào đó, nhằm mục đích làm phù hợp attention vào local encoder, NARM tiến hành mỗi chuỗi $[x_1, x_2, \ldots, x_{t-1}, x_t]$ một cách tách biệt. Mô hình của chúng ta có thể được huấn luyện sử dụng một mini-batch gradient chuẩn trên hàm mất mát cross-entropy:

$$L(p, q) = -\sum_{i=1}^{m} p_i \log(q_i)$$

Với p là phân phối xác suất ground truth và q phân phối xác suất được dự báo. Cuối cùng chúng ta sử dụng phương pháp lan truyền ngược (Back propagation through time) để cập nhật các biểu diễn cho các hidden state ứng với mỗi item.

4. Huấn luyện mô hình

4.1. Dataset

yoochoose (https://2015.recsyschallenge.com/) là tập dữ liệu được cung cấp từ cuộc thi RecSys Challange 2015. Dữ liệu bao gồm tập hợp các thông tin về các click event trong từng session tương ứng. Trong các events sẽ bao gồm buy event giúp xác định các sự kiện mua hàng. Mục tiêu là dự báo khách hàng có khả năng mua hàng sắp tới không và nếu mua hàng thì mua sản phẩm nào?

Bộ dữ liệu bao gồm 3 files chính:

Các files huấn luyện:

1 . **yoochoose-clicks.dat**: Dữ liệu về các click events. Mỗi một dòng bao gồm các trường sau đây:

- Session ID: Id của session. Trong một session có thể có 1 hoặc nhiều lượt clicks.
- Timestamp: Thời điểm diễn ra click.
- Item ID: Id dùng để xác định item.
- Category: Nhóm category của item.

2 . **yoochoose-buys.dat**: Dữ liệu về buy events. Mỗi một dòng bao gồm các trường sau đây:

- Session ID: Id của session. Trong một session có thể có 1 hoặc nhiều lượt mua sắm.
- Timestamp: Thời điểm diễn ra hành vi mua sắm.
- Item ID: Id của item.
- Price: Gía của sản phẩm.
- Quantity: Số lượng sản phẩm được mua.

Do các trong các session sẽ một số session có các event buying nên số lượng các session trong file yoochoose-buys.dat nhiều hơn so với yoochoose-clicks.dat và mọi session trong file yoochoose-buys.dat sẽ chứa trong file yoochoose-clicks.dat. Một session có thể rất ngắn (vài phút) hoặc rất dài (vài giờ) và theo đó số lượng event cũng tương ứng ít hoặc nhiều tùy vào hoạt động của user.

File kiểm tra:

3 . **yoochoose-test.dat**: Cấu trúc tương tự như file yoochoose-clicks.dat của huấn luyện. Bao gồm các trường: Session ID, Timestamp, Item ID, Category. Mục tiêu của chúng ta là cần dự báo xem trong các session của tập test có event buying hay không và nếu có thì list các Item ID được mua là gì?

4.1.1. Khởi tạo dữ liệu

Bắt đầu từ bước này bạn đọc có thể mở file NARSyscom2015.ipynb (https://colab.research.google.com/drive/17Y3FLu_CzWc9ZMoVCSZDm4D6wVB4-jMb?usp=sharing) để thực hành code trực tiếp trên google colab.

Bên dưới chúng ta sẽ tiến hành đọc dữ liệu từ các file trong tập dataset `yoochoose` và thực hiện một số lọc bỏ các session có độ dài ngắn và các item có tần suất xuất hiện thấp.

```
import os

path = 'your_folder_path'
os.chdir(path)
os.listdir()
```

Unzip file `yoochoose-data.7z`

```
from pyunpack import Archive

os.mkdir('yoochoose-data')
Archive('yoochoose-data.7z').extractall('yoochoose-data')
```

Đọc dữ liệu yoochoo-clicks.dat dataset

```
 import os
 import pandas as pd
 from datetime import datetime
 import time

 filepath = 'yoochoose-data/yoochoose-clicks.dat'
 dataset = pd.read_csv(filepath, names = ['SessionId', 'DateTime', 'It
 dataset['DateTime'] = dataset['DateTime'].apply(lambda x: datetime.st
 dataset['Timestamp'] = dataset['DateTime'].apply(lambda x: x.strftime
 dataset = dataset.sort_values(by = ['SessionId', 'DateTime'], ascendi
```

Bên dưới ta sẽ lọc bỏ các itemId có ít hơn 5 lượt xuất hiện và lọc bỏ các session có ít hơn 2 itemId.

```
# Thống kê số lượt xuất hiện và lọc ra các ItemId có trên 5 lượt xuất h
df_item_count = dataset[['ItemId', 'SessionId']].groupby('ItemId').coun
df_item_count.columns = ['CountItemId']
df_item_count_5 = df_item_count[df_item_count['CountItemId'] < 5]
# Lọc khỏi dataset những ItemId có ít hơn 5 lượt xuất hiện
dataset = dataset[~dataset['ItemId'].isin(list(df_item_count_5.index))
```

Top

4.1.2. Phân chia dữ liệu train/test

Từ tập dữ liệu train ta sẽ tách ra 7 ngày cuối cùng làm dữ liệu test. Đối với dữ liệu còn lại để việc huấn luyện được nhanh hơn thì chúng ta sẽ chỉ dữ lại 1/4 số lượng các session cho huấn luyện.

Khi đó dữ liệu test sẽ có cấu trúc tương tự như train, mỗi session bao gồm các itemIds được sắp xếp theo thứ tự thời gian. Từ list các itemId liền trước ta cần dự báo itemId tiếp theo có khả năng được click. Các dữ liệu trên test được xem như là session mới hoàn toàn và được sử dụng để kiểm tra mức độ dự báo chuẩn xác của mô hình được huấn luyện từ tập dữ liệu train.

Bên dưới ta sẽ thực hành phân chia train/test cho mô hình.

```
from datetime import timedelta
# Phân chia tập train/test sao cho tập test là 7 ngày gần đây nhất và
maxdate = dataset['DateTime'].max()
mindate7 = maxdate - timedelta(days = 7)
test = dataset[dataset['DateTime'] >= mindate7]
dataset = dataset[dataset['DateTime'] <= mindate7]
```

Lấy ra ngẫu nhiên 1/4 số lượng các session cho huấn luyện.

```
import numpy as np

# list các sessionIds
sessIds = list(dataset['SessionId'].unique())
# Lấy ngẫu nhiên 1/4 số lượng các session
n_filter = int(len(sessIds)/4)
np.random.shuffle(sessIds)
sessIdsFilter = sessIds[:n_filter]
# Lựa chọn các 1/4 session làm dataset train (dữ liệu này bao gồm cả
# Set index là sessionId để filter nhanh hơn
dataset.set_index('SessionId', inplace=True)
dataset_filter = dataset[dataset.index.isin(sessIdsFilter)]
dataset_filter = dataset_filter.reset_index()
```

Khởi tạo chuỗi các itemId sắp xếp theo thời gian trên mỗi session. Mỗi một itemId sẽ tương ứng với giá trị thời gian của nó. Cấu trúc của các `train_sess, test_sess` có dạng:

```
['sessionId': {'itemId1':'Timestamp1', ..., 'itemId_n':'Timestamp_n'}]
```

```
# Lấy ra dictionary có dạng {SessionId:{ItemId1:Timestamp1, ItemId2:Ti
train_sess = dataset_filter[['SessionId', 'ItemId', 'Timestamp']].grou
test_sess = test[['SessionId', 'ItemId', 'Timestamp']].groupby('Sessio
```

4.1.3. Preprocessing data

Ở bước này ta sẽ khởi tạo input và output cho mô hình.

Input của mô hình là list các itemIds trong quá khứ và Output là itemId ở vị trí hiện tại. Qúa trình khởi tạo các Input và Output trên một session được thực hiện tịnh tiến từ vị trí itemID đầu tiên cho đến cuối cùng.

Ví dụ: Chúng ta có `sessionId = [itemId_1, itemId_2, ..., itemId_n]` Như vậy sau các cặp (input, output) sẽ là:

Top

- cặp thứ 1: `([itemId_1], [itemId_2])`

- cặp thứ 2: `([itemId_1, itemId_2], [itemId_3])`

  ...

- cặp thứ n-1: `([itemId_1, itemId_2,..., itemId_(n-1)], [itemId_n])`

```
sessDict = {214834865: '1396808691.295000', 214706441: '1396808691.42
def _preprocess_sess_dict(sessDict):
    sessDictTime = dict([(v, k) for (k, v) in sessDict.items()])
    sessSort = sorted(sessDictTime.items(), reverse = False)
    times = [item[0] for item in sessSort]
    itemIds = [item[1] for item in sessSort]
    inp_seq = []
    labels = []
    inp_time = []

    for i in range(len(sessSort)):
        if i >= 1:
            inp_seq += [itemIds[:i]]
            labels += [itemIds[i]]
            inp_time += [times[i]]
    return inp_seq, inp_time, labels, itemIds

inp_seq, inp_time, labels, itemIds = _preprocess_sess_dict(sessDict)
print('input sequences: ', inp_seq)
print('input times: ', inp_time)
print('targets: ', labels)
print('sequence: ', itemIds)
```

Khởi tạo chuỗi input và output cho toàn bộ các session

```
def _preprocess_data(data_sess):
    inp_seqs = []
    inp_times = []
    labels = []
    sequences = []
    sessIds = list(data_sess.index)
    for sessId in sessIds:
        sessDict = data_sess.loc[sessId]
        inp_seq, inp_time, label, sequence = _preprocess_sess_dict(se
        inp_seqs += inp_seq
        inp_times += inp_time
        labels += label
        sequences += sequence
    return inp_seqs, inp_times, labels, sequences

train_inp_seqs, train_inp_dates, train_labs, train_sequences = _prepr
test_inp_seqs, test_inp_dates, test_labs, test_sequences = _preproces

train = (train_inp_seqs, train_labs)
test = (test_inp_seqs, test_labs)

print('Done.')
```

Top

Do kích thước dữ liệu là khá lớn nên để thuận lợi cho những lượt train sau ta nên lưu dữ liệu của train và test vào một folder là `yoochoose-data-4`.

```
import pickle
import os

def _save_file(filename, obj):
  with open(filename, 'wb') as fn:
    pickle.dump(obj, fn)

# Tạo folder yoochoose-data-4 để lưu dữ liệu train/test nếu chưa tồn
if not os.path.exists('yoochoose-data/yoochoose-data-4'):
  os.mkdir('yoochoose-data/yoochoose-data-4')

# Lưu train/test
_save_file('yoochoose-data/yoochoose-data-4/train.pkl', train)
_save_file('yoochoose-data/yoochoose-data-4/test.pkl', test)
```

Ở những lượt huấn luyện sau ta chỉ cần load lại những dữ liệu train/test đã lưu tại folder `yoochoose-data-4`

```
import pickle

def _load_file(filename):
  with open(filename, 'rb') as fn:
    data = pickle.load(fn)
  return data

# Load dữ liệu train/test từ folder
train = _load_file('yoochoose-data/yoochoose-data-4/train.pkl')
test = _load_file('yoochoose-data/yoochoose-data-4/test.pkl')
```

4.1.4. Khởi tạo dictionary

Chúng ta cần sử dụng dictionary để nhúng itemId của mỗi một sản phẩm thành một index sao cho mỗi chỉ số này là duy nhất đối với mỗi itemId. Từ index ta có thể tạo ra được các véc tơ one-hot dễ dàng làm đầu vào cho huấn luyện mô hình ở bước embedding.

Top

```
# Các token default
PAD_token = 0  # token padding cho câu ngắn

class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.item2index = {}
        self.item2count = {}
        self.index2item = {PAD_token: "PAD"}
        self.num_items = 1  # số lượng mặc định ban đầu là 1 ứng với l

    def addSenquence(self, data):
        for sequence in data:
          for item in sequence:
              self.addItem(item)

    # Thêm một item vào hệ thống
    def addItem(self, item):
        if item not in self.item2index:
            self.item2index[item] = self.num_items
            self.item2count[item] = 1
            self.index2item[self.num_items] = item
            self.num_items += 1
        else:
            self.item2count[item] += 1

    # Loại các item dưới ngưỡng xuất hiện min_count
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_items = []

        for k, v in self.item2count.items():
            if v >= min_count:
                keep_items.append(k)

        print('keep_items {} / {} = {:.4f}'.format(
            len(keep_items), len(self.item2index), len(keep_items) /
        ))

        # Khởi tạo lại từ điển
        self.item2index = {}
        self.item2count = {}
        self.index2item = {PAD_token: "PAD"}
        self.num_items = 1

        # Thêm các items vào từ điển
        for item in keep_items:
            self.addItem(item)

    # Hàm convert sequence về chuỗi các indices
    def _seqItem2seqIndex(self, x):
        return [voc.item2index[item] if item in voc.item2index else 0
```

Top

Lấy toàn bộ list các itemIds trong các session.

```
from itertools import chain
seq_targets = [train[1]] + [test[1]]
sessionIds = list(chain.from_iterable(seq_targets))
sessionIds = set(sessionIds)
print('Number of sessionIds: ', len(sessionIds))
```

Khởi tạo vocabullary cho bộ dữ liệu.

```
voc = Voc('DictItemId')
voc.addSenquence(seq_targets)

# Convert thử nghiệm một sequence itemIds
print('sequence of itemIds: ', train[0][7])
print('converted indices: ', voc._seqItem2seqIndex(train[0][7]))
```

Tiếp theo ta sẽ chuyển dữ liệu train, test từ item sang indices của item

```
train_x_index = [voc._seqItem2seqIndex(seq) for seq in train[0]]
test_x_index = [voc._seqItem2seqIndex(seq) for seq in test[0]]
train_y_index = voc._seqItem2seqIndex(train[1])
test_y_index = voc._seqItem2seqIndex(test[1])
train_index = (train_x_index, train_y_index)
test_index = (test_x_index, test_y_index)
```

4.1.5. Phân chia tập train/test/validation

Từ 2 tập dữ liệu `train_index` và `test_index` ban đầu ta sẽ phân chia thành 3 tập train/test và validation như sau:

- Mỗi tập dữ liệu bao gồm 2 phần: input bao gồm chuỗi các itemId liên tiếp, output là itemId tiếp theo trong được khách hàng click.
- Dữ liệu validation được rút ra từ dữ liệu train set theo tỷ lệ được qui định tại valid_portion. Phần còn lại của train set được dữ làm tập train set mới.
- Dữ liệu test được tính toàn trực tiếp từ tập test set.
- Đối với dữ liệu input, chuỗi dữ liệu sẽ được truncate về độ dài maxlen nếu nó vượt qua maxlen.

Top

```
def load_data(root='', valid_portion=0.1, maxlen=19, sort_by_len=Fals
    """Load dataset từ root
    root: folder dữ liệu train, trong trường hợp train_set, test_set
    valid_portion: tỷ lệ phân chia dữ liệu validation/train
    maxlen: độ dài lớn nhất của sequence
    sort_by_len: có sort theo chiều dài các session trước khi chia ha
    train_set: training dataset
    test_set:  test dataset
    """

    # Load the dataset
    if train_set is None and test_set is None:
        path_train_data = os.path.join(root, 'train.pkl')
        path_test_data = os.path.join(root, 'test.pkl')
        with open(path_train_data, 'rb') as f1:
            train_set = pickle.load(f1)

        with open(path_test_data, 'rb') as f2:
            test_set = pickle.load(f2)

    if maxlen:
        new_train_set_x = []
        new_train_set_y = []
        # Lọc dữ liệu sequence đến maxlen
        for x, y in zip(train_set[0], train_set[1]):
            if len(x) < maxlen:
                new_train_set_x.append(x)
                new_train_set_y.append(y)
            else:
                new_train_set_x.append(x[:maxlen])
                new_train_set_y.append(y)
        train_set = (new_train_set_x, new_train_set_y)
        del new_train_set_x, new_train_set_y

        new_test_set_x = []
        new_test_set_y = []
        for xx, yy in zip(test_set[0], test_set[1]):
            if len(xx) < maxlen:
                new_test_set_x.append(xx)
                new_test_set_y.append(yy)
            else:
                new_test_set_x.append(xx[:maxlen])
                new_test_set_y.append(yy)
        test_set = (new_test_set_x, new_test_set_y)
        del new_test_set_x, new_test_set_y

    # phân chia tập train thành train và validation
    train_set_x, train_set_y = train_set
    n_samples = len(train_set_x)
    sidx = np.arange(n_samples, dtype='int32')
    np.random.shuffle(sidx)
    n_train = int(np.round(n_samples * (1. - valid_portion)))
    valid_set_x = [train_set_x[s] for s in sidx[n_train:]]
    valid_set_y = [train_set_y[s] for s in sidx[n_train:]]
    train_set_x = [train_set_x[s] for s in sidx[:n_train]]
    train_set_y = [train_set_y[s] for s in sidx[:n_train]]

```

Top

```
        (test_set_x, test_set_y) = test_set

        # Trả về indices thứ tự độ dài của mỗi phần tử trong seq
        def len_argsort(seq):
            return sorted(range(len(seq)), key=lambda x: len(seq[x]))

        # Sắp xếp session theo độ dài tăng dần
        if sort_by_len:
            sorted_index = len_argsort(test_set_x)
            test_set_x = [test_set_x[i] for i in sorted_index]
            test_set_y = [test_set_y[i] for i in sorted_index]

            sorted_index = len_argsort(valid_set_x)
            valid_set_x = [valid_set_x[i] for i in sorted_index]
            valid_set_y = [valid_set_y[i] for i in sorted_index]

        train = (train_set_x, train_set_y)
        valid = (valid_set_x, valid_set_y)
        test = (test_set_x, test_set_y)
        return train, valid, test
```

4.2. Data Loader

Để streaming dữ liệu theo batch thì ta cần sử dụng class `DataLoader` của pytorch. Class này có chức năng tương tự như ImageDataGenerator trong tensorflow và keras. Nó sẽ cho phép ta sử dụng generator để sinh dữ liệu cho từng batch huấn luyện. Do đó ta sẽ có thể huấn luyện được những mô hình từ dữ liệu có kích thước lớn hơn RAM gấp rất nhiều lần. Data Loader trong pytorch sẽ sử dụng dữ liệu là các class `Dataset` của pytorch như bên dưới.

4.2.1. RecSysDataset

`Dataset` sẽ có dữ liệu hoàn toàn giống với tập train/test và validation mà ta đã phân chia ở bước trên. Chúng được tạo ra đơn thuần là để thích hợp với định dạng data được sử dụng trong quá trình khởi tạo `DataLoader` .

```
    from torch.utils.data import Dataset

    class RecSysDataset(Dataset):
        """define the pytorch Dataset class for yoochoose and diginetica
        """
        def __init__(self, data):
            self.data = data
            print('-'*50)
            print('Dataset info:')
            print('Number of sessions: {}'.format(len(data[0])))
            print('-'*50)

        def __getitem__(self, index):
            session_items = self.data[0][index]
            target_item = self.data[1][index]
            return session_items, target_item

        def __len__(self):
            return len(self.data[0])
```

Top

4.2.2. Hàm phụ trợ

Ngoài ra để tùy biến các dữ liệu từ `Dataset` , ta có thể truyền vào `DataLoader` thông qua tham số `collate_fn` một hàm số có tác dụng biến đổi dữ liệu.

Đối với mô hình này ta cũng sẽ sử dụng hàm `collate_fn()` như bên dưới để biến đổi data (chính là các batch) theo các bước như sau:

- Sắp xếp độ dài input sequence theo độ dài list từ cao xuống thấp. Việc này sẽ giúp cho quá trình huấn luyện nhanh hơn.
- Padding thêm 0 vào dữ liệu để độ dài list bằng nhau và bằng với độ dài của input sequence lớn nhất trong batch.
- transpose dữ liệu từ `batch_size x maxlen --> maxlen x batch_size`
- Cuối cùng trả về kết quả ngoài list các chuỗi sessions, list các labels còn trả thêm list các lens ghi nhận độ dài của các sesssions.

```
import torch

def collate_fn(data):
    """
    Hàm số này sẽ được sử dụng để pad session về max length
    Args:
      data: batch truyền vào
    return:
      batch data đã được pad length có shape maxlen x batch_size
    """
    # Sort batch theo độ dài của input_sequence từ cao xuống thấp
    data.sort(key=lambda x: len(x[0]), reverse=True)
    lens = [len(sess) for sess, label in data]
    labels = []
    # Padding batch size
    padded_sesss = torch.zeros(len(data), max(lens)).long()
    for i, (sess, label) in enumerate(data):
        padded_sesss[i,:lens[i]] = torch.LongTensor(sess)
        labels.append(label)

    # Transpose dữ liệu từ batch_size x maxlen --> maxlen x batch_siz
    padded_sesss = padded_sesss.transpose(0,1)
    return padded_sesss, torch.tensor(labels).long(), lens
```

4.3. Metric

Có 2 metrics chính để đo lường hiệu quả của mô hình recommendation đó là:

- `Recall@k` : Metric của dữ liệu bao gồm tỷ lệ xuất hiện ground truth của itemId trong top k sản phẩm được suggest có xác suất lớn nhất. Tỷ lệ này cho biết khả năng khách hàng sẽ click vào một sản phẩm được suggest từ mô hình với xác suất là bao nhiêu. Do đó giá trị của nó càng lớn thì mô hình sẽ có độ chuẩn xác càng cao. Nếu `recall@20 = 10%` có nghĩa là nếu áp dụng mô hình để suggest ra 20 sản phẩm cho khách hàng thì khả năng họ có click vào sản phẩm là 10%.

Top

- `mrr@k` : Chúng ta sẽ quan tâm trong trường hợp ground truth của itemId nằm trong top k sản phẩm được suggest thì thứ tự là bao nhiêu? Nếu vị trí của nó càng nhỏ thì độ chính xác của mô hình càng cao. `mrr@k` sẽ là nghịch đảo vị trí của ground truth trong trường hợp nó nằm trong top k sản phẩm được suggest. Do đó `mrr@k` càng lớn thì mô hình càng chất lượng.

Để tính toán các metrics trên sẽ dựa trên ground truth và top k sản phẩm được suggest từ mô hình như bên dưới:

Top

```
import torch

def get_recall(indices, targets):
    """
    Tính toán chỉ số recall cho một tập hợp predictions và targets
    Args:
        indices (Bxk): torch.LongTensor. top-k indices được dự báo từ
        targets (B): torch.LongTensor. actual target indices.
    Returns:
        recall (float): the recall score
    """
    # copy targets k lần để trở thành kích thước Bxk
    targets = targets.view(-1, 1).expand_as(indices)
    # so sánh targets với indices để tìm ra vị trí mà khách hàng sẽ h
    hits = (targets == indices).to(device)
    hits = hits.double()
    if targets.size(0) == 0:
        return 0
    # Đếm số hit
    n_hits = torch.sum(hits)
    recall = n_hits / targets.size(0)
    return recall


def get_mrr(indices, targets):
    """
    Tính toán chỉ số MRR cho một tập hợp predictions và targets
    Args:
        indices (Bxk): torch.LongTensor. top-k indices được dự báo từ
        targets (B): torch.LongTensor. actual target indices.
    Returns:
        recall (float): the MRR score
    """
    tmp = targets.view(-1, 1)
    targets = tmp.expand_as(indices)
    hits = (targets == indices).to(device)
    hits = hits.double()
    if hits.sum() == 0:
      return 0
    argsort = []
    for i in np.arange(hits.shape[0]):
      index_col = torch.where(hits[i, :] == 1)[0]+1
      if index_col.shape[0] != 0:
        argsort.append(index_col.double())
    inv_argsort = [1/item for item in argsort]
    mrr = sum(inv_argsort)/hits.shape[0]
    return mrr


def evaluate(logits, targets, k=20):
    """
    Đánh giá model sử dụng Recall@K, MRR@K scores.
    Args:
        logits (B,C): torch.LongTensor. giá trị predicted logit cho i
        targets (B): torch.LongTensor. actual target indices.
    Returns:
        recall (float): the recall score
```

Top

```
            mrr (float): the mrr score
        """
        # Tìm ra indices của topk lớn nhất các giá trị dự báo.
        _, indices = torch.topk(logits, k, -1)
        recall = get_recall(indices, targets)
        mrr = get_mrr(indices, targets)
        return recall, mrr
```

Hàm `evaluate()` sẽ tính toán đồng thời cả 2 chỉ số là `recall@k` và `mrr@k`. Tham số của hàm `evaluate()` bao gồm:

- logits: Kích thước `BxC` trong đó `B` là `batch_size` và `C` là số `class`. Mỗi dòng của logits là phân phối xác suất dự báo cho itemId tiếp theo.
- targets: Kích thước `B` trong đó `B` là `batch_size`. Đây là index của itemId mà khách hàng đã click và chính là ground truth của mô hình.

Kiểm tra hàm `evaluate()`.

```
import torch
import numpy as np
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

logits = torch.tensor([[0.1, 0.2, 0.7],
                       [0.4, 0.1, 0.5],
                       [0.1, 0.2, 0.7]]).to(device)

targets = torch.tensor([1, 2, 2]).to(device)

evaluate(logits = logits, targets = targets, k = 2)
```

4.4. Model NARM

4.4.1. Các layer của NARM

Embedding Layer: Là layer nhúng giúp mã hóa các item index thành những véc tơ với chiều xác định bằng `embedding_dim`. Embedding Layer sẽ nhận đầu vào là một sequence của các item index. Layer này sẽ thực hiện 2 biến đổi chính:

- Đầu tiên mỗi một item index sẽ được mã hóa về véc tơ one-hot (là véc tơ đơn vị có giá trị tại vị trí index = 1 và các vị trí khác bằng 0). Như vậy kích thước của véc tơ one-hot sẽ bằng kích thước của từ điển. Lấy ví dụ: Nếu từ điển có 10000 từ với index nhận giá trị từ 1-10000, khi đó một từ có index = 2 sẽ được mã hóa thành véc tơ [0, 1, 0, …, 0]. Tức vị trí số 2 của véc tơ bằng 1 và các vị trí còn lại bằng 0.
- Sau khi mã hóa xong toàn bộ các từ trong câu thì input của chuỗi item index sẽ trở thành ma trận có kích thước là `max_len x vocabulary_size`. Khi đó để giảm chiều của dữ liệu ta sẽ sử dụng một phép chiếu linear projection để giảm chiều từ `vocabulary_size` về `embedding_dim`. Như vậy sau cùng thu được ma trận kích thước `max_len x embedding_dim`. Lưu ý: ta giả định là không quan tâm đến `batch_size` nên kích thước ma trận ở trên chưa bao gồm `batch_size`.

Top

Dropout: Layer này có tác dụng tắt đi ngẫu nhiên một số lượng kết nối liên kết giữa 2 layers để giảm thiểu overfitting cho mô hình.

GRU: Layer GRU là trọng tâm của mô hình, thực hiện quá trình dự báo chuỗi. Mô hình bao gồm nhiều time step và mỗi một time step sẽ trả ra phân phối xác suất đại diện cho từ ở vị trí đó. Kết quả trả ra của layer GRU goòm 2 thành phần: `(output, hidden)`.

- Trong đó `output` sẽ chứa toàn bộ các hidden véc tơ tại toàn bộ các time step t tại layer GRU cuối cùng (chúng ta có thể chồng nhiều layers GRU nối tiếp nhau thông qua khai báo ở tham số `num_layers` và chúng đều đưa ra cùng một kết quả cuối cùng mà không phụ thuộc vào `num_layers`, kết quả trả ra sẽ được tính từ layer cuối). `output` sẽ có kích thước là `max_len x batch_size x (n_directions*hidden_size)`. Sở dĩ có thêm `n_direction` là vì mô hình được thực hiện theo một chiều từ trái qua phải hoặc 2 chiều từ trái qua phải và từ phải qua trái. Trường hợp 2 chiều thì tại mỗi time step sẽ có 2 hidden véc tơ đại diện cho 2 chiều.
- Tham số thứ 2 trả ra từ layer GRU là `hidden` chính là list các véc tơ hidden tại time step cuối cùng của toàn bộ các layer GRU. Như vậy `hidden` sẽ có kích thước là `(num_layers*n_directions) x batch_size x hidden_size`. Để thu được hidden state của layer GRU cuối cùng thì ta sẽ trích suất véc tơ cuối cùng (chính là công thức `ht = hidden[-1]` bên dưới).

Quá trình tính attention

Top

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_seque

class NARM(nn.Module):
    def __init__(self, hidden_size, n_items, embedding_dim, n_layers=
        super(NARM, self).__init__()
        self.hidden_size = hidden_size
        self.n_items = n_items
        self.embedding_dim = embedding_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(self.n_items, self.embedding_di
        # Initialize GRU; the input_size and hidden_size params are b
        # set bidirectional = True for bidirectional
        # https://pytorch.org/docs/stable/nn.html?highlight=gru#torch
        self.gru = nn.GRU(input_size = hidden_size, # number of expec
                          hidden_size = hidden_size, # number of expe
                          num_layers = n_layers, # number of GRU laye
                          dropout=(0 if n_layers == 1 else dropout), 
                          bidirectional=True # one or two directions.
                         )
        self.emb_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(self.embedding_dim, self.hidden_size, self.
        self.a_1 = nn.Linear(self.hidden_size, self.hidden_size, bias
        self.a_2 = nn.Linear(self.hidden_size, self.hidden_size, bias
        self.v_t = nn.Linear(self.hidden_size, 1, bias=False)
        self.ct_dropout = nn.Dropout(0.5)
        self.b = nn.Linear(self.embedding_dim, 2 * self.hidden_size, 
        self.sf = nn.Softmax()
        self.device = torch.device('cuda' if torch.cuda.is_available(

    def forward(self, input_seq, input_lengths, hidden=None):
        """
        input_seq: Batch input_sequence. Shape: max_len x batch_size
        input_lengths: Batch input lengths. Shape: batch_size
        """
        # Step 1: Convert sequence indexes to embeddings
        # shape: (max_length , batch_size , hidden_size)
        embedded = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module. Padding zero
        # shape: (max_length , batch_size , hidden_size)
        packed = pack_padded_sequence(embedded, input_lengths)

        # Step 2: Forward packed through GRU
        # outputs is output of final GRU layer
        # hidden is concatenate of all hidden states corresponding wi
        # outputs shape: (max_length , batch_size , hidden_size x num
        # hidden shape: (n_layers x num_directions , batch_size , hid
        outputs, hidden = self.gru(packed, hidden)
        # Unpack padding. Revert of pack_padded_sequence
        # outputs shape: (max_length , batch_size , hidden_size x num
        outputs, length = pad_packed_sequence(outputs)

        # Step 3: Global Encoder & Local Encoder
        # num_directions = 1 -->
        # outputs shape:(max_length , batch_size , hidden_size)
```

Top

```
58            # hidden shape: (n_layers , batch_size , hidden_size)
59            # lấy hidden state tại time step cuối cùng
60            ht = hidden[-1]
61            # reshape outputs
62            outputs = outputs.permute(1, 0, 2) # [batch_size, max_length,
63            c_global = ht
64            # Flatten outputs thành shape: [batch_size, max_length, hidde
65            gru_output_flatten = outputs.contiguous().view(-1, self.hidde
66            # Thực hiện một phép chiếu linear projection để tạo các laten
67            q1 = self.a_1(gru_output_flatten).view(outputs.size())
68            # Thực hiện một phép chiếu linear projection để tạo các laten
69            q2 = self.a_2(ht)
70            # Ma trận mask đánh dấu vị trí khác 0 trên padding sequence.
71            mask = torch.where(input_seq.permute(1, 0) > 0, torch.tensor(
72            # Điều chỉnh shape
73            q2_expand = q2.unsqueeze(1).expand_as(q1) # shape [batch_size,
74            q2_masked = mask.unsqueeze(2).expand_as(q1) * q2_expand # bat
75            # Tính trọng số alpha đo lường similarity giữa các hidden sta
76            alpha = self.v_t(torch.sigmoid(q1 + q2_masked).view(-1, self.
77            alpha_exp = alpha.unsqueeze(2).expand_as(outputs) # batch_siz
78            # Tính linear combinition của các hidden state
79            c_local = torch.sum(alpha_exp * outputs, 1) # (batch_size x h
80
81            # Véc tơ combinition tổng hợp
82            c_t = torch.cat([c_local, c_global], 1) # batch_size x (2*hid
83            c_t = self.ct_dropout(c_t)
84            # Tính scores
85
86            # Step 4: Decoder
87            # embedding cho toàn bộ các item
88            item_indices = torch.arange(self.n_items).to(device) # 1 x n_
89            item_embs = self.embedding(item_indices) # n_items x embeddin
90            # reduce dimension by bi-linear projection
91            B = self.b(item_embs).permute(1, 0) # (2*hidden_size) x n_ite
92            scores = torch.matmul(c_t, B) # batch_size x n_items
93            # scores = self.sf(scores)
94            return scores
```

4.4.2. Kiểm tra model NARM

Top

```
# Thử nghiệm model bằng cách giả lập 1 input và thực hiện quá trình f
from torch import nn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
hidden_size = 3
n_layers = 7
# embedding = nn.Embedding(11000, hidden_size)
input_variable = torch.tensor([[  66,  369,   66, 1272],
                               [ 567,  183,   28,  616],
                               [ 392, 1558, 1143,  175],
                               [ 394,   31,   31, 5558],
                               [   0,    0,    0,    0]]).to(device)

lengths =  torch.tensor([5, 5, 5, 5]).to(device)
print('input_seq: \n', input_variable)
print('input_lengths: \n', lengths)
model_test = NARM(hidden_size = hidden_size, n_items  = 100000, embed
print('model phrase: \n', model_test)
scores = model_test.forward(input_seq = input_variable, input_lengths
print('probability distribution: ', scores.shape)
```

4.5. Validation

Hàm validation sẽ sử dụng để tính toán `recall@k` và `mrr@k` trên tập dữ liệu validation.

Đầu tiên ta sẽ sử dụng `model` để dự báo xác suất output cho từng batch. `valid_loader` là một iterator được sử dụng để khởi tạo batch. Sử dụng vòng lặp để để duyệt qua toàn bộ batch của `valid_loader` và lưu các giá trị `recall@k` và `mrr@k` tính được ở mỗi batch vào list. Cuối cùng lấy trung bình của toàn bộ các chỉ số này để thu được `recall@k` và `mrr@k` đại diện trên tập validation.

```
def validate(valid_loader, model):
    model.eval()
    recalls = []
    mrrs = []
    with torch.no_grad():
        for seq, target, lens in valid_loader:
            seq = seq.to(device)
            target = target.to(device)
            outputs = model(seq, lens)
            logits = F.softmax(outputs, dim = 1)
            recall, mrr = evaluate(logits, target, k = args['topk'])
            recalls.append(recall)
            mrrs.append(mrr)

    mean_recall = torch.mean(torch.stack(recalls))
    mean_mrr = torch.mean(torch.stack(mrrs))
    return mean_recall, mean_mrr
```

4.6. Training model

Top

Hàm `main()` giúp huấn luyện mô hình trên toàn bộ epochs và trả về kết quả là mô hình sau huấn luyện. Bên cạnh đó, mô hình sau huấn luyện sẽ được lưu vào file `latest_checkpoint.pth.tar` để có thể tái sử dụng về sau.

Hàm `trainForEpoch()` thực hiện huấn luyện mô hình trên mỗi một epoch. Dựa vào Data Loader, chúng ta có thể dễ dàng duyệt qua toàn bộ dataset và training trên từng batch.

Top

```
import os
import time
import random
import argparse
import pickle
import numpy as np
from tqdm import tqdm
from os.path import join

import torch
from torch import nn
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
from torch.autograd import Variable
from torch.backends import cudnn

args = {
    'dataset_path':'../input/yoochoose/yoochoose-clicks.dat',
    'batch_size': 256,
    'hidden_size': 100,
    'embed_dim': 50,
    'epoch': 2,
    'lr':0.001,
    'lr_dc':0.1,
    'lr_dc_step':80,
    'test':None,
    'topk':20,
    'valid_portion':0.1
}

here = os.path.dirname(os.getcwd())
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu'

def main():
    print('Loading data...')
    train_data, valid_data, test_data = load_data(train_set=train_in
    train_data = RecSysDataset(train_data)
    valid_data = RecSysDataset(valid_data)
    test_data = RecSysDataset(test_data)
    train_loader = DataLoader(train_data, batch_size = args['batch_s
    valid_loader = DataLoader(valid_data, batch_size = args['batch_s
    test_loader = DataLoader(test_data, batch_size = args['batch_siz
    print('Complete load data!')
    n_items = voc.num_items
    model = NARM(hidden_size = args['hidden_size'], n_items = n_item
    print('complete load model!')

    if args['test'] == 'store_true':
        ckpt = torch.load('latest_checkpoint.pth.tar')
        model.load_state_dict(ckpt['state_dict'])
        recall, mrr = validate(test_loader, model)
        print("Test: Recall@{}: {:.4f}, MRR@{}: {:.4f}".format(args[
        return model

    optimizer = optim.Adam(model.parameters(), args['lr'])
```

Top

```
        criterion = nn.CrossEntropyLoss()
        scheduler = StepLR(optimizer, step_size = args['lr_dc_step'], ga
        print('start training!')
        for epoch in tqdm(range(args['epoch'])):
            # train for one epoch
            trainForEpoch(train_loader, model, optimizer, epoch, args['e
            scheduler.step(epoch = epoch)
            recall, mrr = validate(valid_loader, model)
            print('Epoch {} validation: Recall@{}: {:.4f}, MRR@{}: {:.4f
            # store best loss and save a model checkpoint
            ckpt_dict = {
                'epoch': epoch + 1,
                'state_dict': model.state_dict(),
                'optimizer': optimizer.state_dict()
            }
            # Save model checkpoint into 'latest_checkpoint.pth.tar'
            torch.save(ckpt_dict, 'latest_checkpoint.pth.tar')
        return model


    def trainForEpoch(train_loader, model, optimizer, epoch, num_epochs,
        model.train()

        sum_epoch_loss = 0

        start = time.time()
        for i, (seq, target, lens) in enumerate(train_loader):
            seq = seq.to(device)
            target = target.to(device)

            optimizer.zero_grad()
            outputs = model(seq, lens)
            loss = criterion(outputs, target)
            loss.backward()
            optimizer.step()

            loss_val = loss.item()
            sum_epoch_loss += loss_val

            iter_num = epoch * len(train_loader) + i + 1

            if i % log_aggr == 0:
                print('[TRAIN] epoch %d/%d  observation %d/%d batch loss
                    % (epoch + 1, num_epochs, i, len(train_loader), loss
                      len(seq) / (time.time() - start)))

            start = time.time()

    model = main()
```

Để nhanh gọn thì tôi chỉ training model trên 2 epoch có sử dụng GPU, thời gian huấn luyện hết tổng cộng 15 phút. Các bạn có thể điều chỉnh tăng epochs để thu được kết quả tốt hơn. Top

Chỉ với 2 epochs nhưng kết quả của `Recall@20` là 43.03%, chỉ số này chứng tỏ khoảng 43.03% khả năng một khi mô hình khuyến nghị ra 20 kết quả top đầu thì khách hàng sẽ có click vào một trong các sản phẩm đó. Tuy nhiên điều này là lý tưởng dựa trên thực tế item mà khách hàng click xuất hiện trong top 20 được dự báo là 43.03%. Khi áp dụng vào recommendation kết quả có thể thấp hơn nhiều do việc click còn phụ thuộc vào vị trí của item có khả năng click có nằm ở top đầu hay không.

Bên cạnh đó `MRR@20` tăng dần qua từng epochs cho thấy vị trí của itemId được dự báo ngày càng sát với top đầu.

4.7. Dự báo

Trước khi thực hiện dự báo thì ta sẽ cần phải load mô hình từ `PATH` đã lưu trước đó. Có 2 kiểu save mô hình chính trên pytorch đó là save toàn bộ mô hình và save các checkpoints. Nếu save toàn bộ mô hình sẽ buộc phải xác định các classes và cấu trúc thư mục lưu trữ mô hình nên thường dẫn tới xảy ra lỗi khi thay đổi project. Do đó save theo checkpoint thường được khuyến nghị, đối với bài toán này chúng ta cũng save mô hình theo checkpoint. Để load lại mô hình cũ ta thực hiện như sau:

```
import torch
import torch.optim as optim

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
PATH = 'latest_checkpoint.pth.tar'
model = NARM(hidden_size = args['hidden_size'], n_items = 37682, embe
optimizer = optim.Adam(params = model.parameters(), lr=0.001)

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
optimizer.load_state_dict(checkpoint['optimizer'])
epoch = checkpoint['epoch']

model.eval()
```

Lưu ý: Sau khi load xong model ta luôn phải chạy hàm `model.eval()` để kích hoạt các dropout và batchnormalization layer. Nếu không kết quả dự báo có thể dẫn tới sai lệch. Tiếp theo chúng ta sẽ sử dụng mô hình để dự báo cho một trường hợp cụ thể.

```
# Lựa chọn ngẫu nhiên một session trên test
import numpy as np
i = np.random.randint(0, len(test_index[0]))
x = [test_index[0][i]]
y = [test_index[1][i]]
print('item indexes sequence input: ', x)
print('item index next output: ', y)
```

Tiếp theo ta sẽ khởi tạo data loader và đưa vào dự báo

Top

```
# Step 1: Khởi tạo test_loader để biến đổi dữ liệu session đưa vào mô
test_data = RecSysDataset([x, y])
test_loader = DataLoader(test_data, batch_size = args['batch_size'],

# Step 2: Dự báo các indice tiếp theo mà khách hàng có khả năng click
def _preddict(loader, model):
    model.eval()
    recalls = []
    mrrs = []
    j = 1
    with torch.no_grad():
      for seq, target, lens in loader:
        seq = seq.to(device)
        target = target.to(device)
        outputs = model(seq, lens)
        logits = F.softmax(outputs, dim = 1)
        _, indices = torch.topk(logits, 20, -1)
        print('Is next clicked item in top 20 suggestions: ', (target
        print('Top 20 next item indices suggested: ')
    return indices

_preddict(test_loader, model)
```

Kết quả cho thấy mô hình dự báo là chính xác. Index của item mà thực tế khách hàng dự báo nằm trong list các sản phẩm được suggest.

5. Tổng kết

Như vậy ta đã hoàn thành quá trình huấn luyện và dự báo mô hình recommendation theo phương pháp `Neural Attentive Session-based Recommendation`. Đây là một mô hình thuộc lớp các mô hình deep neural network có hiệu quả dự báo tốt hơn so với một số phương pháp state-of-art khác như `Session-Based Recommendations With Reccurent Neural Networks`.

Đồng thời qua bài viết tôi cũng đã giới thiệu đến các bạn phương pháp phân loại mô hình recommendation bao gồm: lớp thuật toán phi mô hình và lớp thuật toán mô hình kèm theo các thuật toán phổ biến trong mỗi lớp. Ưu nhược điểm của mỗi thuật toán và trường hợp nên áp dụng.

Hi vọng rằng chúng ta có thể tự xây dựng được một mô hình recommendation cho doanh nghiệp của mình.

6. Tài liệu

1. Neural Attentive Session-based Recommendation - Jing Li, Pengjie Ren, el. Shangdong University, China (https://arxiv.org/pdf/1711.04725.pdf)
2. A Survey on Session-based Recommender Systems - ShouJing Wang, LongBing Cao, Yan Wang, University of Technology Sydney, Australia (https://arxiv.org/pdf/1902.04864.pdf)
3. Session-Based Recommendations With Reccurent Neural Networks - Balazs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, Domonkos Tikk, Hungary (https://arxiv.org/pdf/1511.06939v4.pdf)

Top

4. Neural Attentive Session Based Recommendation PyTorch - Wang Shuo, github (https://github.com/Wang-Shuo/Neural-Attentive-Session-Based-Recommendation-PyTorch)
5. Neural Attentive Recommendation Machine - lijingsdu, github (https://github.com/lijingsdu/sessionRec_NARM)
6. Bài 20 - Recommendation Neural Network - khanh’s blog (https://phamdinhkhanh.github.io/2019/12/26/Sorfmax_Recommendation_Neural_Network.html)
7. Bài 15 - collaborative và content-based filtering, khanh’s blog (https://phamdinhkhanh.github.io/2019/11/04/Recommendation_Compound_Part1.html)
8. Bài 3 - Mô hình Word2Vec, khanh’s blog (https://phamdinhkhanh.github.io/2019/04/29/ModelWord2Vec.html)

Top