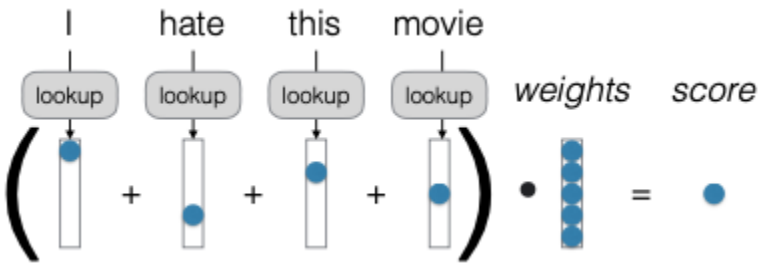
Lecture 2: Biểu diễn từ và Phân loại văn bản

Reminder: Bag of Words (BOW)



Trong bài trước, chúng ta đã học về mô hình Bag of Words. Nhắc lại, mô hình này biểu diễn mỗi từ dưới dạng một vector one-hot. Chúng ta cộng tất cả các vector này lại với nhau, sau đó nhân vector tần suất thu được với một số trọng số (weight) để có được một điểm số. Điểm số này có thể được sử dụng cho phân loại nhị phân hoặc nếu chúng ta muốn thực hiện phân loại đa lớp, chúng ta sẽ có nhiều điểm số cho mỗi lớp. Các đặc trưng F chỉ dựa trên danh tính của các từ và trọng số.

What's Missing in BOW?

Xử lý các từ ghép hoặc từ phức tạp là một thách thức trong mô hình bag of words. Ví dụ, câu "I love this move" có thể được chuyển thành "I loved this movie". Đối với việc xử lý sự tương đồng giữa các từ, câu "I love this movie" có thể được thay thế bằng "I adore this movie". Về việc xử lý các đặc điểm kết hợp, câu "I don't love this movie" có thể được tạo ra từ "I love this movie", và "I don't hate this movie" từ "I hate this movie". Cuối cùng, cấu trúc câu cũng cần được xem xét, ví dụ như "It has an interesting story, but is boring overall".

Tuy nhiên, việc xây dựng một hệ thống dựa trên quy tắc để giải quyết những vấn đề này là không hề đơn giản và tốn nhiều thời gian. Một số giải pháp cho các vấn đề này bao gồm mô hình dựa trên subword hoặc ký tự để xử lý các từ ghép hoặc từ phức tạp. Còn đối với việc xử lý sự tương đồng giữa các từ, chúng ta có thể sử dụng word embeddings. Việc xử lý các đặc điểm kết hợp có thể được thực hiện thông qua mạng nơ-ron. Cuối cùng, cách tiêu chuẩn để xử

lý cấu trúc câu hiện nay là thông qua các mô hình dựa trên chuỗi (Sequence Models), và chúng ta sẽ bắt đầu tìm hiểu về chúng trong vài bài tiếp theo.

Subword Models

Basic Idea

Ý tưởng cơ bản của mô hình subword là chia nhỏ những từ ít phổ biến thành nhiều token subword (từ con). Ví dụ, câu "the companies are expanding" có thể được chia "companies" thành "compan" và "_ies", và "expanding" thành "expand" và "_ing".

Lợi ích:
1. Chia sẻ tham số giữa các biến thể từ và từ ghép.
2. Giảm kích thước tham số, tiết kiệm tài nguyên tính toán và bộ nhớ.

Có khoảng 2 triệu từ trong tiếng Anh, nhưng nhiều mô hình chỉ sử dụng khoảng 60.000 từ. Việc sử dụng các mô hình subword giúp giảm số lượng từ cần thiết mà vẫn giữ được ý nghĩa.

Một vấn đề quan trọng là liệu "company" và "companies" có phải là những từ khác nhau hay không. Nếu chúng ta biết từ "company", chúng ta có thể đoán được nghĩa của "companies". Điều này cho thấy có nhiều sắc thái trong việc xác định từ.

Có một quy luật nổi tiếng gọi là Zipf's law, cho thấy rằng các từ thường gặp sẽ có chuỗi byte ngắn hơn, trong khi các từ ít gặp sẽ có chuỗi byte dài hơn. Điều này dẫn đến việc chúng ta cần xử lý một số lượng từ vô hạn hoặc ít nhất là các chuỗi được phân tách bằng khoảng trắng, và đó là lý do tại sao các đơn vị subword ra đời.

Có một số cách phổ biến để tạo ra các đơn vị subword, nhưng tất cả đều dựa trên việc muốn các chuỗi thường gặp trở thành các đơn vị subword. Một phương pháp thay thế là xử lý từng ký tự hoặc byte trong chuỗi như một đơn vị riêng biệt, nhưng điều này có thể dẫn đến các chuỗi rất dài, gây khó khăn trong việc xử lý và thiếu tính biểu cảm.

Khi chuyển từ mô hình ký tự sang mô hình subword, chúng ta cần cân bằng giữa độ dài và tính biểu cảm. Nếu các subword quá dài, chúng sẽ trở nên hiếm gặp, trong khi nếu quá ngắn, chúng sẽ không đủ biểu cảm. Mô hình subword giúp chúng ta đạt được sự cân bằng này, cho phép mô hình học được các mối quan hệ giữa các từ và token, từ đó cải thiện khả năng xử lý ngôn ngữ tự nhiên.

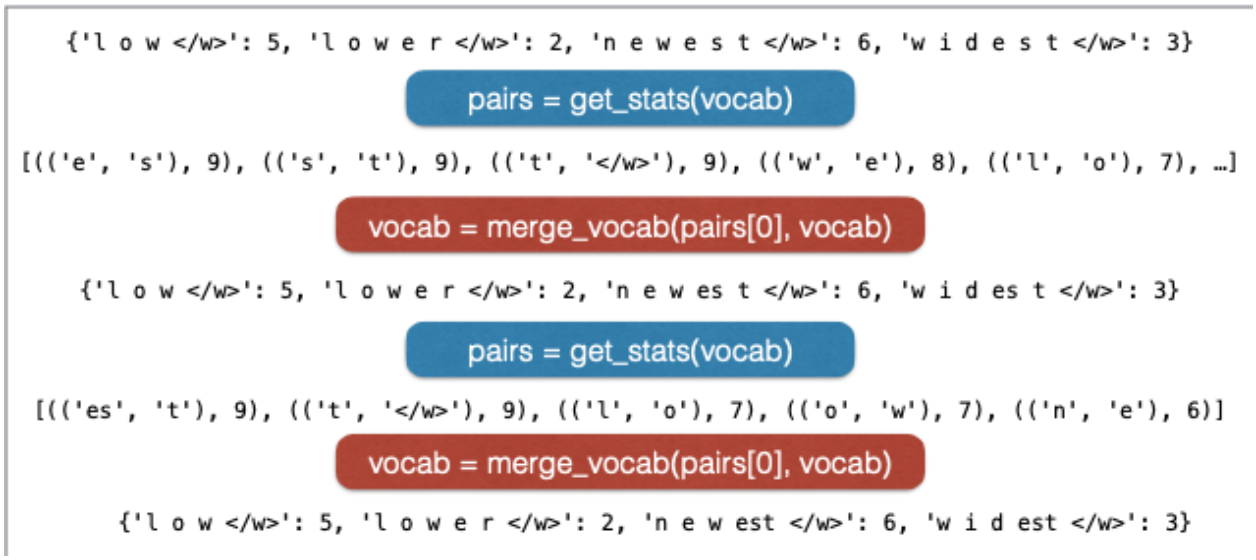
Byte Pair Encoding (Sennrich+ 2015)

Byte Pair Encoding (BPE) là một kỹ thuật đơn giản và hiệu quả được sử dụng để tạo ra các subwords, hay còn gọi là các từ con. Các subwords này có độ dài lớn hơn một ký tự nhưng lại ngắn hơn một token hoàn chỉnh.

Cách thức hoạt động của BPE rất dễ hiểu: đầu tiên, bạn bắt đầu với một tập hợp từ vựng mà bạn muốn xử lý. Mỗi từ trong tập hợp này sẽ được phân tách thành các ký tự riêng lẻ, kèm theo một ký hiệu đặc biệt để đánh dấu sự kết thúc của từ.

Tiếp theo, bạn sẽ thu thập thông tin về các cặp token (các ký tự hoặc nhóm ký tự) xuất hiện cạnh nhau trong tập dữ liệu. Ví dụ, nếu bạn có các từ như "newest" và "wildest", bạn sẽ nhận thấy rằng cặp "e s" xuất hiện nhiều lần. Tương tự, cặp "s t" cũng có thể xuất hiện trong các từ khác.

Sau khi xác định được các cặp token phổ biến nhất, bạn sẽ tiến hành hợp nhất chúng lại với nhau để tạo ra các token mới. Ví dụ, từ cặp "e s", bạn có thể tạo ra token mới là "es". Quá trình này sẽ được lặp đi lặp lại nhiều lần, cho phép bạn tạo ra các token như "EST", một hậu tố phổ biến trong tiếng Anh, từ các cặp token đã được hợp nhất.



Nếu mục tiêu của bạn là xây dựng một từ vựng với 60,000 token, bạn sẽ thực hiện một số lượng hợp nhất tương ứng, cụ thể là 60,000 trừ đi số ký tự ban đầu. Cuối cùng, bạn sẽ có một từ vựng phong phú với 60,000 token, giúp cải thiện khả năng xử lý ngôn ngữ tự nhiên. Phương pháp BPE không chỉ đơn giản mà còn rất hiệu quả trong việc tối ưu hóa việc mã hóa và phân tích ngôn ngữ.
Extra Reading (Optional):
https://medium.com/@hsinhungw/understanding-byte-pair-encoding-fd196ebfe93f
Source code:
https://github.com/neubig/anlp-code/tree/main/02-subword

Unigram Models (Kudo 2018)

Mô hình Unigram (Kudo 2018) là loại mô hình ngôn ngữ đơn giản nhất, trong đó tất cả các từ trong chuỗi được sinh ra độc lập với nhau. Trong bài giảng này, tôi sẽ giải thích chi tiết hơn về chúng sau, nhưng cơ bản là bạn sẽ tạo ra một mô hình mà xác suất của chuỗi được tính bằng tích của xác suất của từng từ trong chuỗi.

Để tối ưu hóa, bạn cần chọn một từ vựng sao cho tối đa hóa log likelihood của tập dữ liệu (corpus) với kích thước từ vựng cố định. Ví dụ, nếu bạn có kích thước từ vựng là 60,000, bạn sẽ tìm cách chọn 60,000 từ tốt nhất để tối đa hóa xác suất của tập dữ liệu. Quá trình tối ưu hóa này được thực hiện bằng cách sử dụng thuật toán EM, trong đó bạn dự đoán xác suất của mỗi token xuất hiện, chọn các token phổ biến nhất và loại bỏ những token ít phổ biến hơn. Quá trình này được lặp đi lặp lại cho đến khi bạn còn lại 60,000 token.

Tuy nhiên, có một vấn đề lớn trong mô hình unigram là giả định về tính độc lập trong các mô hình ngôn ngữ, vì điều này cho phép bạn sắp xếp lại thứ tự của các từ trong câu. Mặc dù vậy, thuật toán EM yêu cầu lập trình động, và bạn không thể dễ dàng thực hiện lập trình động nếu không có những giả định này. Cuối cùng, sau khi đã chọn từ vựng và gán xác suất cho mỗi từ, bạn sẽ tìm một phân đoạn của đầu vào sao cho tối đa hóa xác suất unigram.

Tóm lại, đây là ý tưởng chính của mô hình unigram. Tôi sẽ không đi vào chi tiết nhiều vì hầu hết mọi người chỉ sử dụng thuật toán này.

SentencePiece

SentencePiece là một thư viện tối ưu hóa cao, cho phép bạn:
- Huấn luyện và sử dụng các mô hình BPE và Unigram.
- Ví dụ lệnh để huấn luyện mô hình:

```
% spm_train --input=<input> --model_prefix=<model_name> --vocab_size=8000 --character_coverage=1.0 --model_type=<type>
```

- Ví dụ lệnh để mã hóa đầu ra:

```
% spm_encode --model=<model_file> —output_format=piece < input > output
```

Thư viện này rất phổ biến trong việc xây dựng các đơn vị từ con (subword units). Bạn chỉ cần chạy chương trình `sentencepiece train`, chọn kích thước từ vựng (vocab size) và độ bao phủ ký tự (character coverage) để xác định mức độ bao phủ các ký tự trong văn bản đầu vào. Sau đó, bạn sử dụng lệnh `sentencepiece encode` để mã hóa và phân tách đầu ra. Thư viện cũng có các binding cho Python.

Một điều quan trọng cần lưu ý là theo mặc định, SentencePiece sử dụng mô hình Unigram, nhưng nó cũng hỗ trợ BPE. Trong kinh nghiệm của tôi, sự khác biệt giữa hai mô hình này không quá lớn. Điều quan trọng hơn là kích thước từ vựng của bạn. Nếu kích thước từ vựng nhỏ, hiệu suất sẽ cao nhưng khả năng biểu đạt sẽ thấp. Ngược lại, nếu kích thước từ vựng lớn, khả năng biểu đạt sẽ cao nhưng hiệu suất sẽ giảm. Một quy tắc chung là từ 60,000 đến 80,000 từ là hợp lý nếu bạn chỉ làm việc với tiếng Anh. Nếu bạn mở rộng sang các ngôn ngữ khác, bạn sẽ cần một kích thước từ vựng lớn hơn.

Bạn có thể tham khảo thêm tại: https://github.com/google/sentencepiece

Subword Considerations

Các cân nhắc về Subword:

Đa ngôn ngữ:
Các mô hình subword khó sử dụng cho nhiều ngôn ngữ vì chúng có thể phân đoạn quá mức các ngôn ngữ ít phổ biến một cách ngây thơ (Ács 2019). Ví dụ, nếu 50% dữ liệu của bạn là tiếng Anh, 30% là các ngôn ngữ khác viết bằng chữ cái Latin, 10% là tiếng Trung, 5% là các ngôn ngữ viết bằng chữ Cyrillic, và 3% là tiếng Nhật, thì các ngôn ngữ ít phổ biến như tiếng Myanmar có thể bị phân đoạn thành các chuỗi rất dài và không hiệu quả. Một cách để khắc phục điều này là tăng cường tần suất của các ngôn ngữ ít được đại diện hơn trong quá trình tạo dữ liệu.

Tính tùy ý: Một vấn đề khác là tính tùy ý trong việc phân đoạn. Ví dụ, với từ "est", chúng ta có thể phân đoạn thành "es t" hoặc "e st". Điều này có thể ảnh hưởng đến kết quả, đặc biệt khi bạn không có một từ vựng mạnh cho ngôn ngữ mà bạn đang làm việc hoặc khi bạn đang làm việc trong một lĩnh vực mới. Một giải pháp cho vấn đề này là "subword regularization", trong đó mẫu các phân đoạn khác nhau trong thời gian huấn luyện để làm cho mô hình trở nên mạnh mẽ hơn trước sự biến đổi này.

Để giải quyết vấn đề đa ngôn ngữ, bạn có thể tạo ra một phân phối khác, giảm trọng số của tiếng Anh và tăng trọng số cho các ngôn ngữ khác. Điều này giúp bạn có được nhiều dữ liệu hơn từ các ngôn ngữ khác khi tạo ra mô hình.

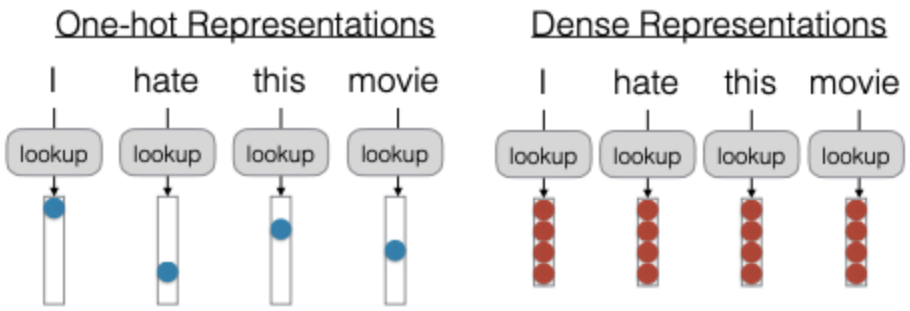
Nếu bạn muốn thêm một ngôn ngữ mới vào một mô hình subword đa ngôn ngữ, bạn có thể sử dụng mô hình unigram, một mô hình xác suất, để kết hợp các xác suất của từ vựng cũ và mới. Một phương pháp khác là huấn luyện trước trên các ngôn ngữ hiện có và sau đó học các embedding cho ngôn ngữ mới, trong khi giữ nguyên phần thân của mô hình.

Việc phân đoạn subword là bước đầu tiên trong việc tạo ra hầu hết các mô hình hiện nay. Các mô hình thường sử dụng phân đoạn ký tự hoặc byte, với phân đoạn byte giúp đơn giản hóa việc xử lý các ký tự Unicode.

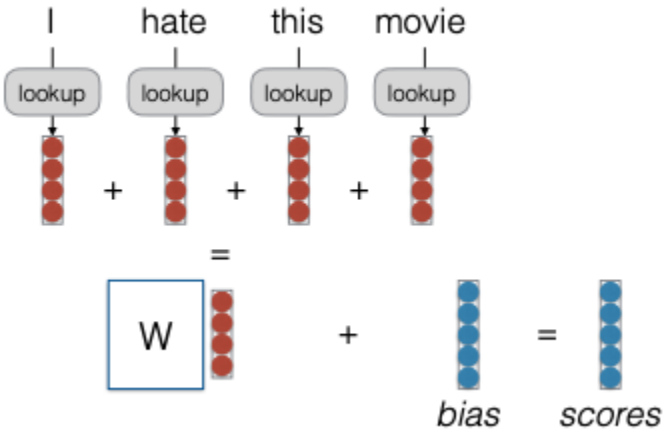
Continuous Word Embeddings

Basic Idea

Ý tưởng cơ bản là trước đây chúng ta đã đại diện cho các từ bằng một vector thưa, với một "1" duy nhất, được gọi là one-hot vector. Continuous word embeddings, hay các embedding từ liên tục, thay vào đó sẽ tra cứu một vector dày đặc, nơi toàn bộ vector có các giá trị liên tục. Điều này mang lại một đại diện dày đặc hơn cho từ ngữ, khác với các biểu diễn thưa như one-hot representations.



Continuous Bag of Words (CBOW)



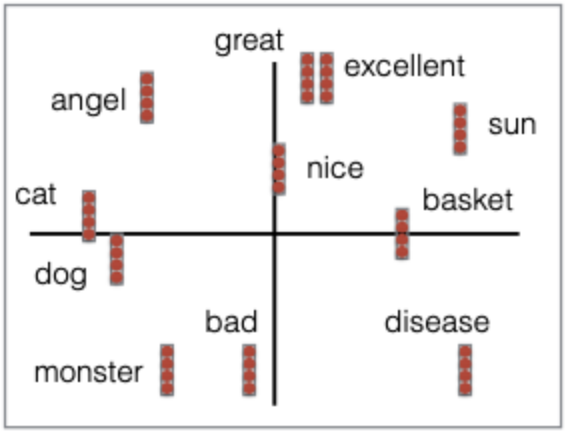
Mô hình CBOW hoạt động bằng cách xem xét các giá trị của từng vector, tức là các embedding của từng từ trong một chuỗi. Điều này cho phép chúng ta tạo ra một vector embedding cho toàn bộ chuỗi. Sau đó, vector này sẽ được nhân với một ma trận trọng số. Cụ thể, các hàng của ma trận trọng số tương ứng với kích thước của embedding liên tục, trong khi các cột của ma trận này sẽ tương ứng với tổng số nhãn mà bạn có.

Kết quả là, mỗi vector giờ đây không chỉ đại diện cho một từ trong từ vựng mà còn hy vọng rằng bạn sẽ nhận được các vector mà từ ngữ có thể được biểu diễn một cách chính xác hơn.

What Do Our Vectors Represent?

Các Vectơ của Chúng Biểu Diễn Điều Gì?
Khi xem xét, bạn sẽ hy vọng rằng các vectơ của các từ tương tự, theo một khái niệm nào đó về sự tương tự như cú pháp hay ngữ nghĩa, dù chúng có cùng ngôn ngữ hay không, sẽ gần nhau trong không gian vectơ. Mỗi phần tử của vectơ là một đặc trưng. Ví dụ, mỗi phần tử của vectơ có thể tương ứng với câu hỏi: "Is this an animate object?" hoặc "Is this a positive word?"

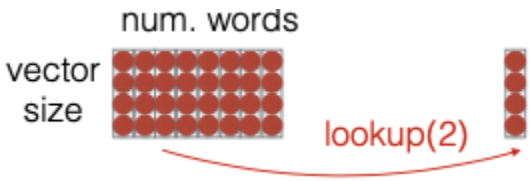
Để minh họa cho khái niệm này, giả sử chúng ta có một không gian vectơ hai chiều, trong đó trục x tương ứng với việc có phải là đối tượng sống hay không, và trục y tương ứng với việc có phải là từ tích cực hay không. Đây là mục tiêu lý tưởng của chúng ta.



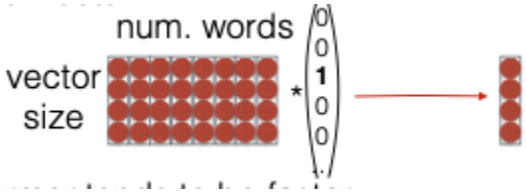
Vậy tại sao chúng ta lại muốn làm điều này? Các phần tử trong không gian vectơ này được học cùng với mô hình. Mục tiêu cuối cùng là sau khi quá trình học diễn ra, các vectơ sẽ có hai đặc tính: thứ nhất, các từ tương tự sẽ gần nhau trong không gian vectơ; thứ hai, các đặc trưng này sẽ có ý nghĩa nào đó, có thể là ý nghĩa có thể hiểu được bởi con người hoặc không.

A Note: “Lookup"

Trong bài này, tôi đã so sánh giữa bag of words và các biểu diễn one-hot với các biểu diễn dày đặc, và tôi đã sử dụng thao tác lookup cho cả hai. Thao tác lookup này thực chất là lấy một vector đơn từ một ma trận lớn các word embeddings. Cách hoạt động của nó là chúng ta có một ma trận lớn và sau đó tìm kiếm cột tương ứng hay là lookup ma trận với index thứ 2 (A[:, i]). Đây là cách mà hầu hết các thư viện học sâu hoặc bất kỳ thư viện nào bạn sử dụng sẽ thực hiện.



Một cách khác để nhìn nhận thao tác này là nhân với một vector one-hot. Bạn có ma trận giống hệt như vậy, nhưng bạn chỉ cần nhân với một vector như 0, 1, 0, 0, và điều đó sẽ cho bạn kết quả giống hệt. Tuy nhiên, các triển khai thực tiễn thường sử dụng phương pháp đầu tiên vì nó nhanh hơn, không cần nhân một ma trận lớn với một vector khổng lồ.



Tuy nhiên, việc hiểu phương pháp thứ hai cũng có lợi. Ví dụ, nếu bạn phát triển một mô hình kỳ lạ dự đoán phân phối xác suất trên các từ thay vì chỉ là các từ, có thể đó là một mô hình ngôn ngữ có ý tưởng về từ tiếp theo sẽ như thế nào. Có thể mô hình của bạn nghĩ rằng từ tiếp theo

có 50% xác suất là “cat”, 30% xác suất là “dog”, và 20% xác suất là “bird”. Bạn có thể lấy vector này và nhân với ma trận để có được một word embedding là sự kết hợp của tất cả các từ đó, điều này có thể thú vị và cho phép bạn thực hiện những điều sáng tạo.

Training a More Complex Model

Reminder: Simple Training of BOW Models

```
for x, y in data:
      # Make a prediction
      features = extract_features(x)
      predicted_y = run_classifier(features)
      # Update the weights if the prediction is wrong
      if predicted_y != y:
            for feature in features:
                  feature_weights[feature] = (
                              feature_weights.get(feature, 0) +
                              y * features[feature]
```

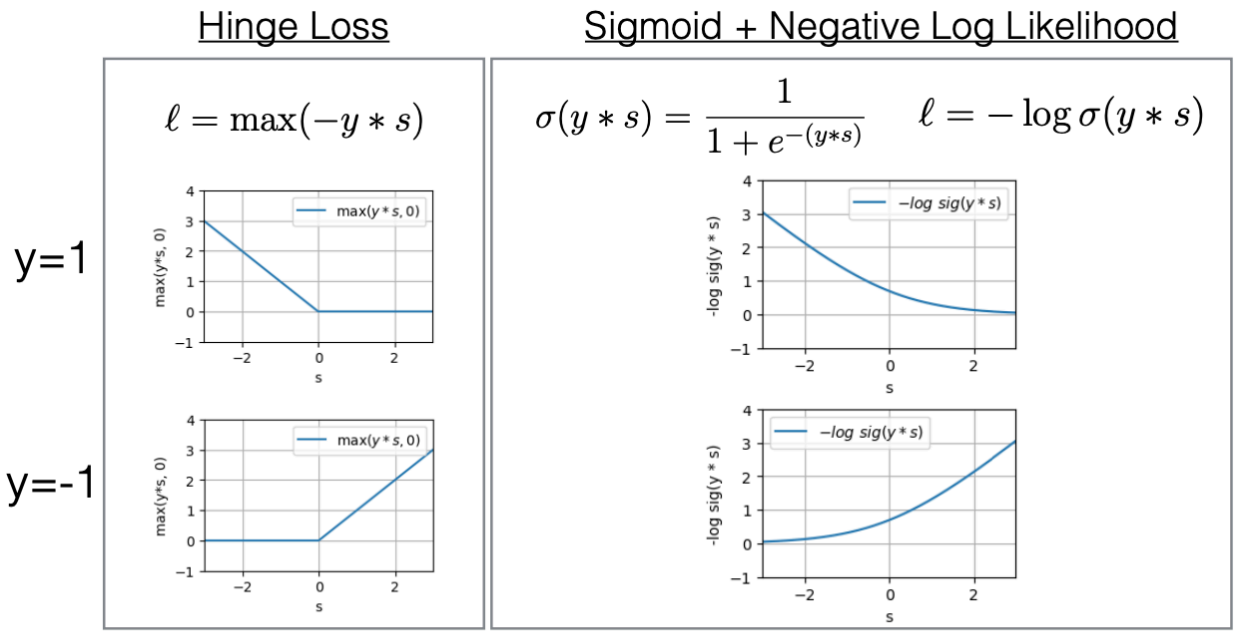
Đến giờ, chúng ta đã đào tạo một mô hình bag of words (BOW) và cách chúng ta đào tạo mô hình BOW là sử dụng thuật toán structured perceptron. Nếu mô hình đưa ra câu trả lời sai, chúng ta sẽ tăng hoặc giảm các embedding dựa trên việc nhãn là tích cực hay tiêu cực. Tôi đã trình bày một ví dụ về thuật toán rất đơn giản này; bạn thậm chí không cần phải viết bất kỳ mã nào như numpy để triển khai thuật toán đó. Đây là cách thực hiện: chúng ta có dữ liệu X, chúng ta trích xuất các đặc trưng, chạy bộ phân loại, có nhãn dự đoán và sau đó tăng hoặc giảm các đặc trưng.
Full example: https://github.com/neubig/anlp-code/tree/main/01-simpleclassifier

How do we Train More Complex Models?

Chúng ta làm thế nào để huấn luyện các mô hình phức tạp hơn? Đầu tiên, chúng ta sử dụng gradient descent. Hầu hết mọi người ở đây đều đã tham gia một khóa học về machine learning nào đó, vì vậy điều này sẽ là một phần ôn tập cho nhiều người. Để thực hiện gradient descent, chúng ta cần viết một hàm mất mát (loss function), tính toán các đạo hàm của hàm mất mát với các tham số, và điều chỉnh các tham số theo hướng giảm thiểu hàm mất mát.

Loss Function

Hàm mất mát (Loss Function) là một giá trị giảm dần khi mô hình cải thiện. Trong bài toán phân loại nhị phân, có hai ví dụ chính thường được sử dụng trong các mô hình NLP. Đầu tiên là "hinge loss" ít phổ biến hơn, và thứ hai là hàm mất mát dựa trên hàm sigmoid kết hợp với "negative log likelihood".



more closely linked to acc probabilistic interpretation, gradients everywhere

Hinge loss hoạt động bằng cách lấy giá trị lớn nhất giữa tích của nhãn (label) Y và điểm số (score) s do mô hình đưa ra với 0. Cụ thể, nếu Y = 1 và điểm số lớn hơn 0, thì mất mát là 0. Điều này có nghĩa là nếu mô hình dự đoán đúng, sẽ không có mất mát. Ngược lại, nếu nhãn Y là âm và mô hình đưa ra điểm số dương, thì mất mát sẽ tăng lên.

Về phần hàm sigmoid kết hợp với negative log likelihood, chúng ta nhân Y với điểm số s và sau đó áp dụng hàm sigmoid, tạo ra một hàm có giá trị nằm trong khoảng từ 0 đến 1. Sau đó, chúng ta lấy log âm của hàm sigmoid này, tạo ra một hàm mất mát L. Sự khác biệt giữa hinge loss và hàm sigmoid là hinge loss có tính sắc nét và cho mất mát bằng 0 khi dự đoán đúng, trong khi hàm sigmoid là mượt mà và không bao giờ cho mất mát bằng 0.

Hinge loss có mối liên hệ chặt chẽ với độ chính xác (accuracy) vì khi mô hình dự đoán đúng tất cả, chúng ta sẽ không cập nhật mô hình nữa do không có mất mát. Ngược lại, hàm sigmoid có hai lợi thế chính: gradient không bao giờ bằng 0 và tăng lên khi điểm số xấu đi, giúp tối ưu hóa mô hình dễ dàng hơn. Thêm vào đó, hàm sigmoid có thể được hiểu như một xác suất, điều này hữu ích cho các mô hình downstream hoặc khi cần dự đoán độ tin cậy từ mô hình.

Calculating Derivatives

Chúng ta sẽ tính toán đạo hàm của tham số dựa trên hàm mất mát. Để minh họa, tôi sẽ sử dụng ví dụ từ mô hình Bag of Words (BOW) và hàm mất mát hinge.

$$\frac{\partial \max(0, -y * \sum_i^{|\mathcal{V}|} w_i \text{freq}(v_i, x))}{\partial w_i} =$$

$$\begin{cases} -y \cdot \text{freq}(v_i, x) & \text{if } -y \cdot \sum_i^{|\mathcal{V}|} w_i \text{freq}(v_i, x) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Hàm mất mát hinge, như tôi đã nói, là giá trị lớn nhất của điểm số (score) nhân với y và 0. Trong mô hình Bag of Words, score là tần suất của từ vựng v_i trong đầu vào x nhân với trọng số w_i. Đây là một hàm đơn giản mà ta có thể tính đạo hàm bằng tay. Kết quả sẽ là: nếu y * giá trị này lớn hơn 0, tức là nếu giá trị lớn nhất chọn giá trị này thay vì giá trị khác, thì đạo hàm sẽ là - y *freq. Ngược lại, đạo hàm sẽ bằng 0.

Optimizing Gradients

Tối ưu hóa bằng phương pháp gradient ngẫu nhiên (stochastic gradient descent) là một trong những thuật toán tối ưu hóa cơ bản nhất cho các mô hình. Chúng ta sẽ tính toán độ dốc (gradient) của hàm mất mát (loss function) đối với các tham số, và ký hiệu độ dốc này là g_t. Cụ thể, chúng ta sẽ lấy giá trị trước đó của tham số, ký hiệu là W (hoặc Theta), và sau đó trừ đi tích của tốc độ học (learning rate) với độ dốc.

$$g_t = \underbrace{\nabla_{\theta_{t-1}} \ell(\theta_{t-1})}_{\text{Gradient of Loss}}$$

$$\theta_t = \theta_{t-1} - \underbrace{\eta}_{\text{Learning Rate}} g_t$$

What is this Algorithms?

```
feature_weights = {}
for x, y in data:
      # Make a prediction
      features = extract_features(x)
      predicted_y = run_classifier(features)
      # Update the weights if the prediction is wrong
      if predicted_y != y:
            for feature in features:
                  feature_weights[feature] = (
                              feature_weights.get(feature, 0) +
```

```
                                y * features[feature])
```

Đoạn code này mô tả một thuật toán học máy đơn giản sử dụng Hinge Loss làm hàm mất mát và Stochastic Gradient Descent (SGD) với tốc độ học là 1. Thuật toán này hoạt động như sau:

Đầu tiên, nó khởi tạo một từ điển `feature_weights` để lưu trữ trọng số của các đặc trưng. Đối với mỗi cặp dữ liệu (x, y), thuật toán sẽ thực hiện dự đoán bằng cách trích xuất các đặc trưng từ x và chạy bộ phân loại để có được `predicted_y`. Nếu dự đoán không đúng (tức là `predicted_y` khác y), thuật toán sẽ cập nhật trọng số của các đặc trưng theo công thức:

```python
feature_weights[feature] = (
 feature_weights.get(feature, 0) +
 y * features[feature] )
```

Hàm mất mát được sử dụng ở đây là Hinge Loss, và tốc độ học là 1. Điều này có nghĩa là nếu dự đoán đúng, thuật toán sẽ không thay đổi trọng số, còn nếu sai, nó sẽ điều chỉnh trọng số theo hướng của y nhân với tần suất của các đặc trưng.

Mặc dù thuật toán này rất đơn giản và chỉ hoạt động hiệu quả với các mô hình như bag of words hoặc các mô hình dựa trên đặc trưng đơn giản, nhưng nó mở ra nhiều khả năng mới cho việc tối ưu hóa các mô hình. Như đã đề cập, thuật toán này tương đương với phương pháp Stochastic Gradient Descent mà chúng ta thường sử dụng trong các mô hình phức tạp hơn.

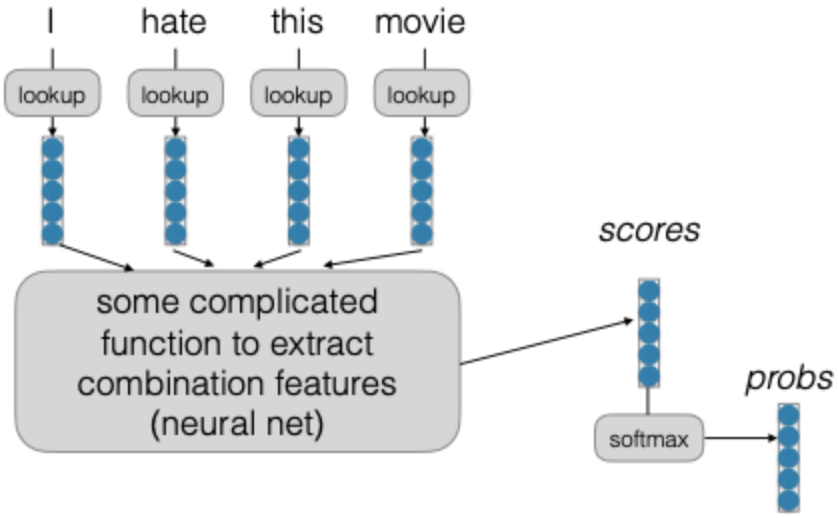
Combination Features

Combination Features

Trong bài trước, chúng ta đã gặp vấn đề với các đặc trưng kết hợp. Ví dụ, "don't hate" và "don't love" không chỉ đơn giản là “hate” cộng với “don’t” và “love” cộng với “don’t”. Sự kết hợp của hai từ này thực sự rất quan trọng. Để minh họa, chúng ta có thể xem xét các câu như "I don’t love this movie" có thể được coi là "bad", trong khi câu "There’s nothing I don’t love about this movie" lại được đánh giá là "very good". Để giải quyết vấn đề này, chúng ta sẽ sử dụng mạng nơ-ron (neural networks).

Basic Idea of Neural Networks (for NLP Prediction Tasks)

Ý tưởng cơ bản của Mạng nơ-ron (Neural Networks) trong các nhiệm vụ dự đoán ngôn ngữ tự nhiên (NLP) là sử dụng các vector để đại diện cho các đặc trưng (features) của dữ liệu. Mỗi vector có thể chứa các đặc trưng như "is this an animate object?" (đây có phải là một đối tượng sống không?) hay "is this a positive word?" (đây có phải là một từ tích cực không?). Chúng ta tổng hợp các đặc trưng này và sử dụng chúng để đưa ra dự đoán. Tuy nhiên, trong giai đoạn đầu, chúng ta chỉ sử dụng sức mạnh biểu diễn của một mô hình tuyến tính mà không có sự kết hợp giữa các đặc trưng, dẫn đến việc giảm chiều dữ liệu.



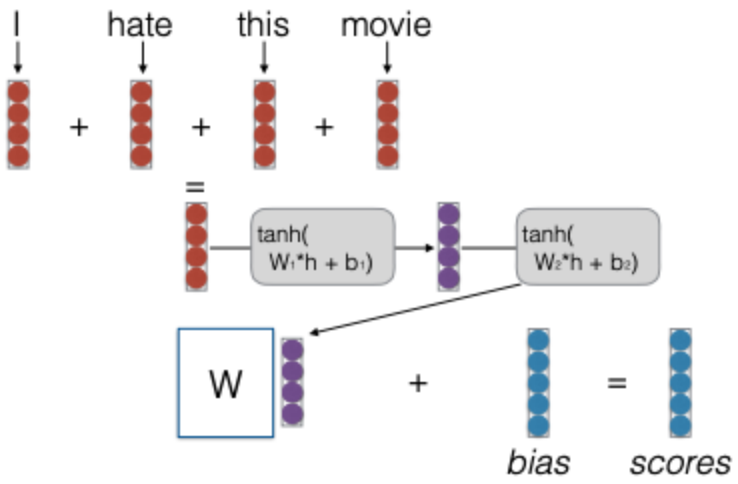
Để khắc phục vấn đề này, chúng ta chuyển sang sử dụng mạng nơ-ron. Cách thực hiện là chúng ta có một lookup của các dense embeddings. Chúng ta tạo ra một hàm phức tạp để trích xuất các đặc trưng kết hợp (combination features) và sau đó sử dụng chúng để tính toán các điểm số (scores). Vấn đề với các đặc trưng Continuous bag of words là chúng ta đã trích xuất các đặc trưng nhưng lại sử dụng trực tiếp các đặc trưng dày đặc (dense features) để đưa ra dự đoán mà không cho phép bất kỳ sự tương tác nào giữa các đặc trưng.

Mạng nơ-ron giúp chúng ta khắc phục điều này bằng cách đầu tiên trích xuất các đặc trưng từ mỗi embedding từ (word embedding), sau đó chạy chúng qua các biến đổi tuyến tính và phi tuyến (linear and nonlinear transforms) để trích xuất thêm các đặc trưng. Cuối cùng, chúng ta chạy các đặc trưng này qua nhiều lớp và sử dụng các đặc trưng kết quả để đưa ra dự đoán.

Deep CBOW

Mô hình Deep Continuous Bag of Words (Deep CBOW) là một phương pháp thú vị trong việc xử lý ngôn ngữ tự nhiên. Mô hình này cho phép chúng ta học các kết hợp giữa các đặc trưng của từ, giúp hiểu rõ hơn về ngữ nghĩa của chúng trong ngữ cảnh.

Cụ thể, trong mô hình này, chúng ta có thể xác định các đặc trưng (features) của từ. Ví dụ, một nút trong lớp thứ hai của mô hình có thể cho biết rằng “feature 1 AND feature 5 are active”, tức là cả hai đặc trưng này đều đang hoạt động. Điều này có thể giúp chúng ta nhận diện các cụm từ mang ý nghĩa tiêu cực, như “not” AND “hate”.



Khi chúng ta áp dụng mô hình này, nó cho phép chúng ta học được các kết hợp đặc trưng. Chẳng hạn, feature một có thể đại diện cho các từ mang nghĩa tiêu cực như "hate" (ghét) và "despise" (khinh thường). Trong trường hợp này, giá trị của feature một sẽ cao, ví dụ như 8.0 cho "hate" và 7.2 cho "despise". Đồng thời, chúng ta cũng có các từ phủ định như "don't" (không) hoặc "not" (không), với giá trị cao cho "don't" là khoảng 2.5.

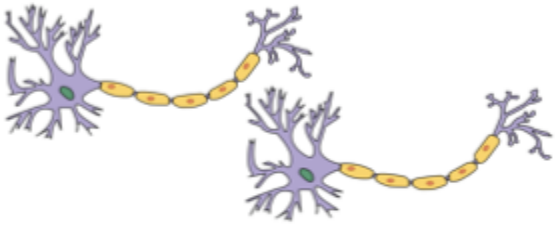
Các từ nhúng (word embeddings) trong mô hình này sẽ tương ứng với các đặc trưng của từ. Sau đó, chúng ta sẽ trích xuất các kết hợp đặc trưng trong lớp thứ hai. Điều này cho phép mô hình nhận diện các cụm từ như "don't hate" (đừng ghét) hoặc "not despise" (không khinh thường).

Mô hình Deep CBOW được đề xuất vào năm 2015 và đã cho thấy nhiều kết quả ấn tượng. Ngay cả một mô hình đơn giản như vậy cũng có thể hoạt động tốt trong các tác vụ phân loại văn bản và các nhiệm vụ khác, nhờ vào khả năng chia sẻ và trích xuất các đặc trưng của từ. Điều này giúp cải thiện khả năng hiểu ngữ nghĩa của các câu và cụm từ trong văn bản.

What is a Neural Network? Computation Graphs

“Neural” Nets

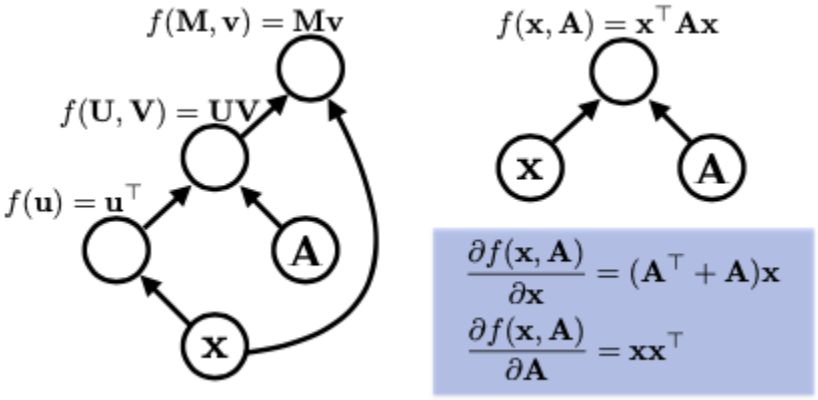
Mạng nơ-ron (Neural Networks) được phát triển dựa trên động lực ban đầu là các nơ-ron trong não bộ, nơi mà mỗi synapse (khớp thần kinh) của nơ-ron tiếp nhận tín hiệu điện và khi đủ tín hiệu, nó sẽ kích hoạt. Hiện nay, khái niệm về mạng nơ-ron hay mô hình học sâu (deep learning models) chủ yếu là các đồ thị tính toán (computation graphs).



Trong một đồ thị tính toán, mỗi nút (node) tương ứng với một giá trị như tensor, ma trận (matrix), vector hoặc scalar. Một nút có một cạnh (edge) đến từ nút khác sẽ là một hàm của nút gốc của cạnh đó. Mỗi nút biết cách tính giá trị của nó và giá trị của đạo hàm (derivative) liên quan đến mỗi tham số (argument) của nó. Các hàm có thể là nullary, unary, binary, hoặc n-ary, nhưng thường là unary hoặc binary. Đồ thị tính toán là có hướng (directed) và không chu trình (acyclic).

Khi thực hiện các phép toán phức tạp, như tính toán hàm mất mát (loss function) cho một mô hình, việc sử dụng đồ thị tính toán giúp đơn giản hóa quá trình này. Ví dụ, biểu thức $x^T Ax$ có thể được biểu diễn bằng nhiều cách khác nhau trong đồ thị, nhưng cách biểu diễn khác nhau có thể ảnh hưởng đến hiệu suất. Đồ thị lớn hơn sẽ tiêu tốn nhiều bộ nhớ và thường chậm hơn. Do đó, việc tối ưu hóa các phép toán là rất quan trọng, đặc biệt khi làm việc với các mô hình phức tạp như attention.

Ví dụ: biểu thức $x^T Ax$ với Graph bên dưới:

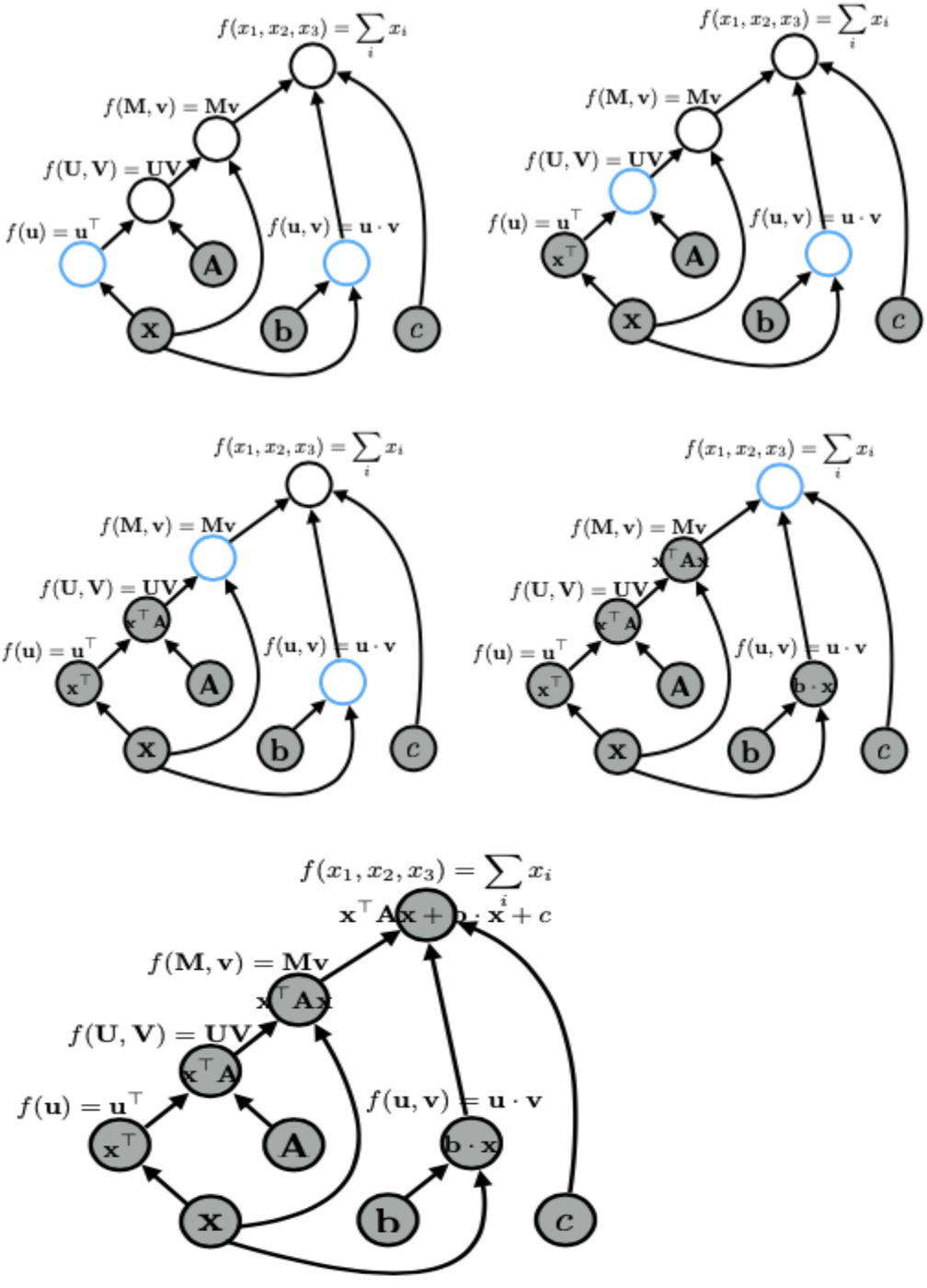


Cuối cùng, trong một đồ thị tính toán của mạng nơ-ron, tên biến chỉ là nhãn cho các nút, và mặc dù bạn chỉ khai báo một biến, nhưng có thể có nhiều phép toán diễn ra ẩn sau, điều này cũng ảnh hưởng đến bộ nhớ và thời gian tính toán. Việc hiểu rõ điều này sẽ giúp bạn tối ưu hóa các triển khai của mình hiệu quả hơn.

Algorithms (1)

Trong quá trình triển khai mạng nơ-ron, có một số thuật toán quan trọng. Đầu tiên là graph construction (xây dựng đồ thị), trong đó bạn sẽ khai báo tất cả các biến và cấu trúc đồ thị. Thuật toán thứ hai là forward propagation (lan truyền thuận), trong đó bạn tính toán giá trị của một nút dựa trên các đầu vào của nó theo thứ tự topo.

Cụ thể, bạn bắt đầu với tất cả các nút mà bạn đã cung cấp làm đầu vào, sau đó tìm bất kỳ nút nào trong đồ thị mà tất cả các nút con của nó đã được tính toán. Trong trường hợp này, đó sẽ là hai nút này. Cuối cùng, bạn có thể tính toán giá trị của tất cả các nút đã thỏa mãn, theo thứ tự tùy ý hoặc thậm chí song song, cho đến khi bạn đạt đến nút cuối cùng.



Algorithms (2)

Quá trình Back-propagation (Lan truyền ngược):

Trong quá trình này, chúng ta sẽ xử lý các ví dụ theo thứ tự topo ngược. Đầu tiên, chúng ta tính toán các đạo hàm của các tham số liên quan đến giá trị cuối cùng, thường là “loss function” (hàm mất mát) mà chúng ta muốn tối thiểu hóa.

Cập nhật tham số:
Sau khi tính toán các đạo hàm, chúng ta sẽ cập nhật các tham số bằng cách di chuyển chúng theo hướng của đạo hàm này. Công thức cập nhật tham số được thể hiện như sau: $W \mathrel{-}= \alpha \cdot dL/dW$, trong đó W là các tham số, α là tốc độ học (learning rate), và dl/dW là đạo hàm của hàm mất mát theo các tham số.

Khi thực hiện Back-propagation, chúng ta bắt đầu từ giá trị cuối cùng (thường là hàm mất mát) và lùi lại theo thứ tự topo để tính toán các đạo hàm của tất cả các tham số.

Mặc dù quy trình này có vẻ đơn giản, nhưng nó rất quan trọng khi triển khai các mô hình xử lý ngôn ngữ tự nhiên (NLP), đặc biệt là những mô hình tiêu tốn nhiều bộ nhớ. Nếu bạn vô tình tính toán cùng một giá trị hai lần hoặc tạo ra một đồ thị thao tác với các tensor lớn và tạo ra các trạng thái trung gian lớn, điều này có thể làm tiêu tốn bộ nhớ và gây ra các vấn đề nghiêm trọng. Vì vậy, việc hiểu rõ quy trình lan truyền ngược là rất cần thiết.

Concrete Implementation Examples

Neural Network Frameworks

Hiện nay, có nhiều framework cho mạng nơ-ron, nhưng trong lĩnh vực Xử lý Ngôn ngữ Tự nhiên (NLP), chủ yếu có hai framework chính, và Pytorch là framework phổ biến nhất. Cả hai framework này đều được phát triển bởi các công ty lớn và có nhiều hỗ trợ kỹ thuật, điều này rất quan trọng khi bạn chọn framework nào vì nó sẽ được hỗ trợ tốt. Pytorch được sử dụng rộng rãi trong nghiên cứu NLP, trong khi Jax cũng được dùng trong một số dự án.

Pytorch ưu tiên thực thi động, có nghĩa là bạn sẽ tạo ra một đồ thị tính toán và thực thi nó mỗi khi xử lý một đầu vào. Ngược lại, Jax yêu cầu bạn định nghĩa đồ thị tính toán trước và sau đó thực thi nó nhiều lần. Điều này có nghĩa là bạn chỉ cần xây dựng đồ thị một lần, sau đó có thể biên dịch và sử dụng lại. Pytorch hỗ trợ cả việc định nghĩa và biên dịch, trong khi Jax thiên về các tính năng động hơn. Sự khác biệt giữa hai khung này là Pytorch mang lại nhiều linh hoạt hơn, trong khi Jax cung cấp khả năng tối ưu hóa tốt hơn và tốc độ làm việc nhanh hơn.

Một điểm thú vị về Jax là nó rất giống với numpy, vì nó sử dụng cấu trúc tương tự và dựa nhiều vào tensor. Nhờ đó, bạn có thể dễ dàng chia một tensor ra hai GPU, điều này rất hữu ích khi huấn luyện một mô hình lớn. Mặc dù việc này có thể thực hiện dễ dàng hơn trong Jax, nhưng cũng hoàn toàn khả thi trong Pytorch.

Cuối cùng, Pytorch có hệ sinh thái phong phú nhất, vì vậy, như đã nói, Pytorch là lựa chọn tốt nhất, nhưng bạn có thể xem xét sử dụng Jax nếu bạn thích thử nghiệm với những điều mới.

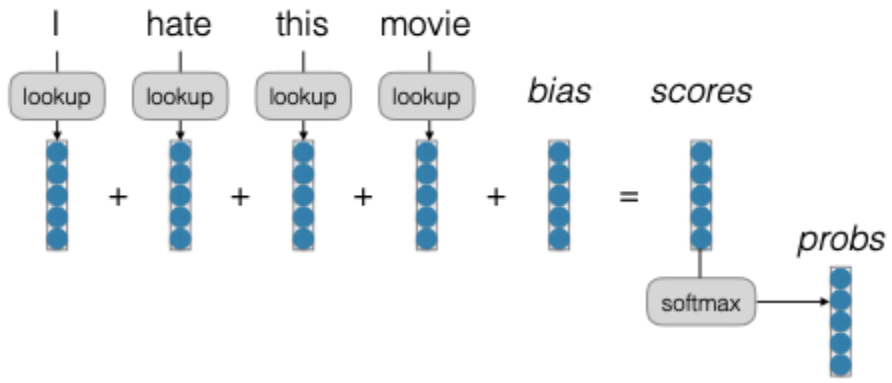
Basic Process in Neural Network Frameworks

Quá trình cơ bản trong các framework mạng nơ-ron bao gồm việc tạo ra một mô hình. Đối với mỗi ví dụ, bạn cần tạo một đồ thị đại diện cho phép tính mà bạn muốn thực hiện và tính toán kết quả của phép tính đó. Nếu đang trong quá trình huấn luyện, hãy thực hiện lan truyền ngược và cập nhật.

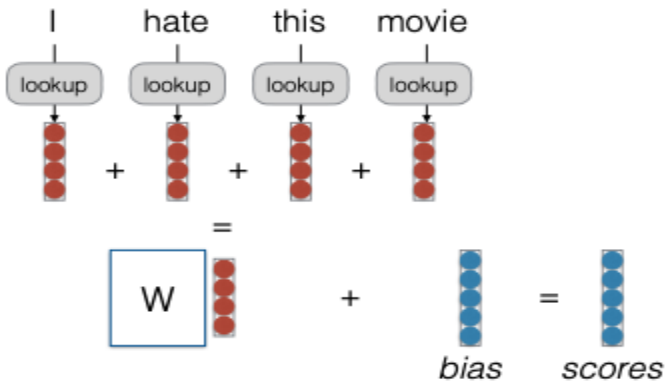
Trong bài giảng hôm nay, chúng ta đã thảo luận về các mô hình như Bag of Words (BOW), Continuous Bag of Words (CBOW) và Deep CBOW. Những mô hình này được tạo ra với mục đích đơn giản hóa, không phải để tối ưu tốc độ hay hiệu suất triển khai. Các ví dụ này được xây dựng dựa trên PyTorch, nơi bạn có thể tạo ra mô hình Bag of Words, mô hình Continuous Bag of Words và mô hình Deep Continuous Bag of Words.

Tất cả các mô hình này được triển khai trong file model.py. Điều quan trọng nhất là nơi bạn định nghĩa forward pass. Ví dụ, đây là nơi bạn thực hiện word embedding, tổng hợp tất cả các embedding và thêm bias. Sau đó, bạn sẽ trả về score. Mô hình Continuous Bag of Words sẽ tổng hợp một số embedding, lấy score và chạy qua một lớp tuyến tính. Mô hình Deep Continuous Bag of Words cũng thêm một vài lớp biến đổi tuyến tính.

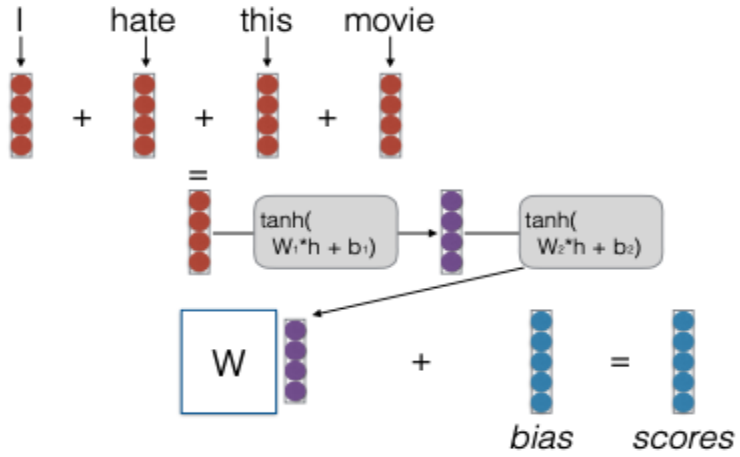
Bag of Words



Continuous Bag of Words



Deep CBOW



Source code: https://github.com/neubig/anlp-code/tree/main/02-textclass

A Few More Important Concepts

A Better Optimization: Adam

Trong NLP, có rất nhiều thuật toán tối ưu mà bạn có thể sử dụng, nhưng hầu hết các nghiên cứu đều sử dụng một biến thể nào đó của Adam. Cách hoạt động của Adam là tối ưu hóa mô hình bằng cách xem xét trung bình lăn (rolling average) của gradient và động lượng (momentum). Cụ thể, chúng ta có gradient và động lượng, và cách mà Adam hoạt động là thêm một phần của gradient vào thành phần động lượng. Điều này giúp động lượng chuyển tiếp một cách mượt mà hơn so với phương pháp gradient tiêu chuẩn, nơi mà mỗi tham số có thể được cập nhật rất khác nhau ở mỗi bước thời gian.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad \text{Momentum}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t \qquad \text{Rolling average of Gradient}$$

Một điểm quan trọng khác là Adam theo dõi phương sai của gradient. Một số tham số có thể có phương sai rất lớn trong gradient và dao động mạnh, trong khi những tham số khác có thể có phương sai nhỏ hơn và không dao động nhiều. Chúng ta cần đảm bảo rằng tất cả các tham số đều được cập nhật một cách thích hợp. Ví dụ, nếu chúng ta có một ma trận embedding từ lớn, với các từ thường xuyên ở một bên và các từ ít thường xuyên ở bên kia, chúng ta muốn đảm bảo rằng tất cả đều nhận được đủ cập nhật. Các tham số với phương sai nhỏ sẽ được cập nhật ít hơn, trong khi các tham số với phương sai lớn sẽ nhận được nhiều cập nhật hơn.

Adam cũng bao gồm các thuật toán để điều chỉnh độ lệch trong giai đoạn đầu của quá trình huấn luyện, vì các thành phần động lượng và phương sai có thể chưa được hiệu chỉnh tốt. Điều này giúp ngăn chặn việc các giá trị này dao động quá mức trong giai đoạn đầu.

$$\widehat{m}_t = m_t / \left(1 - (\beta_1)^t\right) \qquad \widehat{v}_t = v_t / \left(1 - (\beta_2)^t\right)$$

Ngoài ra, trong các mô hình Transformer, việc điều chỉnh tốc độ học (learning rate) là rất quan trọng. Adam có tham số tốc độ học, và chúng ta thường bắt đầu với một giá trị thấp, sau đó tăng lên và cuối cùng giảm xuống. Điều này là cần thiết vì các mô hình Transformer rất nhạy cảm với tốc độ học cao ngay từ đầu. Nếu tốc độ học quá cao, mô hình có thể trở nên không ổn định. Tuy nhiên, chúng ta cần tăng tốc độ học để mô hình học một cách hiệu quả và giảm tốc độ học vào cuối quá trình để ngăn chặn mô hình dao động và đạt được độ chính xác tốt trên một lượng lớn dữ liệu.

Cập nhật cuối cùng: $\theta_t = \theta_{t-1} - \eta/\sqrt{\hat{v}_t + \varepsilon} \cdot \hat{m}_t$

Giải thích thêm cho Thuật toán Adam (Tuỳ chọn)

What Is the Adam Optimization Algorithm?

Thuật toán tối ưu Adam là một phương pháp điều chỉnh tốc độ học, được thiết kế để cải thiện tốc độ huấn luyện trong các mạng nơ-ron sâu và đạt được hội tụ nhanh chóng. Adam được giới thiệu trong bài báo “Adam: A Method for Stochastic Optimization.”

Trước khi tìm hiểu về Adam, chúng ta cần nắm vững thuật toán gradient descent tiêu chuẩn, là nền tảng cho Adam. Gradient descent tiêu chuẩn được biểu diễn bằng công thức sau:

$$\theta = \theta - \alpha \cdot g_t$$

θ = Tham số mô hình, α = Tốc độ học, g□ = Đạo hàm của hàm chi phí theo các tham số.

Cập nhật này thay đổi các tham số θ theo hướng âm của đạo hàm để giảm thiểu hàm chi phí. Tốc độ học α xác định kích thước bước nhảy.

Trong thuật toán gradient descent tiêu chuẩn, tốc độ học α là cố định, điều này có nghĩa là chúng ta cần bắt đầu với một tốc độ học cao và điều chỉnh α theo từng bước hoặc theo một lịch trình học. Tốc độ học thấp sẽ dẫn đến hội tụ rất chậm, trong khi tốc độ học quá cao có thể bỏ lỡ điểm cực tiểu.

Adam giải quyết vấn đề này bằng cách điều chỉnh tốc độ học α cho từng tham số θ, cho phép hội tụ nhanh hơn so với gradient descent tiêu chuẩn với tốc độ học toàn cục cố định.

How Does Adam Optimization Work?

Chiến lược điều chỉnh tốc độ học tập thích ứng có thể được hiểu qua ví dụ về Adam, một người cha dạy hai đứa trẻ, Chris và Sam, cách đạp xe. Chris sợ tốc độ, trong khi Sam lại đạp rất nhanh. Nếu Adam đẩy cả hai xe với cùng một tốc độ, Chris sẽ không theo kịp, còn Sam có thể gặp tai nạn.

Adam quan sát tốc độ và gia tốc của từng đứa trẻ để điều chỉnh cách đẩy xe. Khi thấy Chris đạp chậm, Adam sẽ đẩy mạnh hơn để giúp Chris tăng tốc. Ngược lại, khi thấy Sam đạp nhanh, Adam sẽ đẩy nhẹ hơn để làm chậm Sam lại. Nhờ việc điều chỉnh này, cả hai đều học đạp xe một cách an toàn và hiệu quả.

Tương tự, thuật toán Adam trong tối ưu hóa điều chỉnh tốc độ học tập cho từng tham số dựa trên lịch sử gradient, giúp mạng nơ-ron học tập hiệu quả hơn.

Theory Behind Adam

Bây giờ, khi đã có cái nhìn tổng quan, chúng ta có thể đi sâu vào các chi tiết toán học phức tạp về cách mà Adam hoạt động. Tuy nhiên, trước khi làm điều đó, chúng ta cần hiểu hai khái niệm quan trọng về các thuật toán tối ưu hóa đã xuất hiện trước Adam, vì chúng kết hợp lại để tạo thành thuật toán Adam.

1. Momentum

Kỹ thuật Momentum giúp tăng tốc quá trình huấn luyện bằng cách gia tăng độ lớn của gradient theo hướng đúng bằng cách thêm một phần của gradient trước đó vào gradient hiện tại. Cụ thể, nếu gradient đã liên tục chỉ về cùng một hướng, thì thuật toán Momentum sẽ tích lũy và tăng tốc độ tối ưu hóa theo hướng đó.

Hãy tưởng tượng gradient descent giống như việc lăn một quả bóng xuống đồi. Thông thường, quả bóng sẽ di chuyển từng bước cố định với cùng một tốc độ học (learning rate). Tuy nhiên, với Momentum, nếu các bước trước đó đã đi theo cùng một hướng, chúng ta có thể tăng tốc cho bước hiện tại, giúp giảm số bước cần thiết.

$$v_t = \gamma * v_{t-1} + \eta * g_t$$

$$\theta = \theta - v_t$$

Về mặt toán học, chúng ta sẽ điều chỉnh cập nhật tham số θ bằng cách trừ đi cập nhật từ momentum. Vector momentum v tại thời điểm t (v□) là hàm của vector momentum trước đó (v□-₁). Tham số hyperparameter γ là hệ số giảm động lượng, trong khi η là tốc độ học dùng để điều chỉnh bước đi ngược lại với gradient.

2. Root Mean Square Propagation (RMSProp)

Momentum và RMSProp là hai phương pháp tối ưu hóa khác nhau trong học sâu. Momentum giúp tăng tốc độ học bằng cách gia tăng tốc độ khi các gradient trước đó có cùng hướng. Ngược lại, RMSProp điều chỉnh tốc độ học một cách thích ứng dựa trên độ dốc của bề mặt lỗi cho từng tham số. Cụ thể, các tham số có gradient cao sẽ có bước cập nhật nhỏ hơn, trong khi các tham số có gradient thấp sẽ cho phép bước cập nhật lớn hơn.

$$E[g_t^2] = \gamma * E[g_{t-1}^2] + (1 - \gamma) * g_t^2$$

$$\theta = \theta - \eta / \sqrt{E[g_t^2] + \varepsilon} * g_t$$

Đầu tiên, chúng ta có một trung bình di động có trọng số của các gradient bình phương, điều này thực chất là phương sai của các gradient. Ở đây, chúng ta thấy rằng tốc độ học trong cập nhật θ được chia cho căn bậc hai của trung bình di động của các gradient bình phương. Điều này có nghĩa là, khi phương sai của các gradient cao, chúng ta sẽ giảm tốc độ học vì chúng ta muốn thận trọng hơn. Ngược lại, khi phương sai của các gradient thấp, chúng ta sẽ tăng tốc độ học, từ đó tiến nhanh hơn về phía tối ưu.

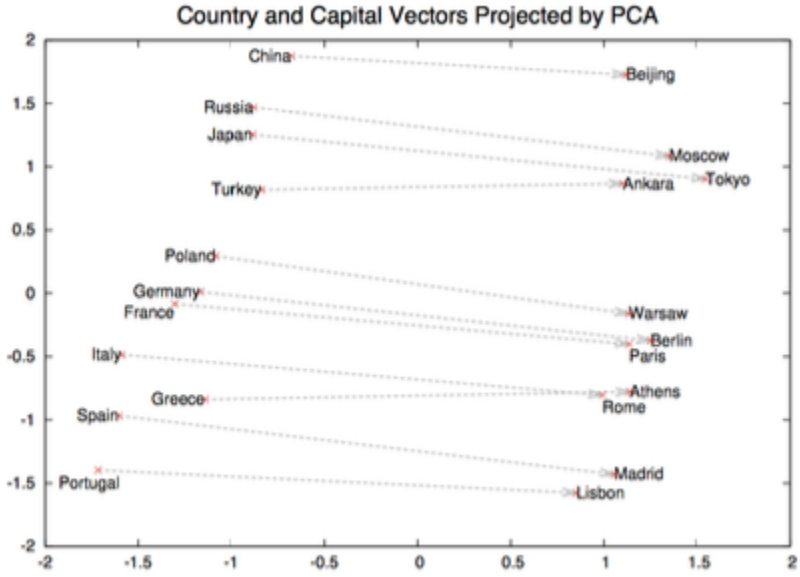
Bây giờ, chúng ta đã hiểu về Momentum và RMSProp, và chúng ta có thể thấy cách hoạt động của Adam. Phiên bản đơn giản của Adam kết hợp hai phương pháp này thông qua các siêu tham số β_1 và β_2.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\theta = \theta - \alpha * m_t / \sqrt{v_t + \varepsilon}$$

Cập nhật θ trong Adam tương tự như cập nhật trong RMSProp, nhưng được tăng cường với Momentum, giúp chúng ta có được những lợi ích tốt nhất từ cả hai phương pháp.
Nguồn: https://builtin.com/machine-learning/adam-optimization

Visualization of Embeddings

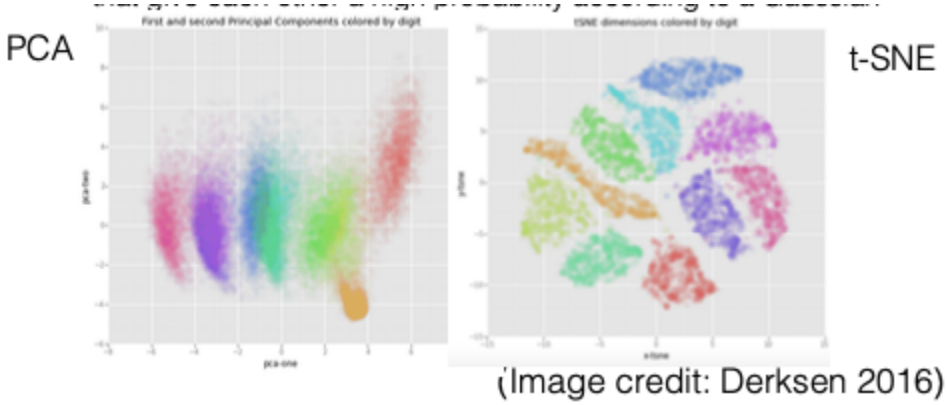
Việc trực quan hóa các vector embedding thường liên quan đến việc giảm kích thước của các vector có nhiều chiều xuống còn 2 hoặc 3 chiều để dễ dàng hình dung. Ví dụ, trong nghiên cứu của Mikolov et al. 2013, các vector embedding từ thường có kích thước lớn, có thể lên tới 512 hoặc 1024 chiều.



Một ví dụ điển hình là sử dụng phương pháp phân tích thành phần chính (PCA), một kỹ thuật giảm chiều tuyến tính. Đây là một trong những nghiên cứu đầu tiên về vector word embedding, cho thấy rằng khi thực hiện giảm chiều tuyến tính, chúng ta có thể phát hiện ra những điều thú vị, chẳng hạn như có thể vẽ một vector gần như cùng hướng giữa các quốc gia và thủ đô của chúng. Tuy nhiên, PCA không phải lúc nào cũng mang lại khả năng trực quan hóa tốt nhất. Trong một số trường hợp, nó có thể không cung cấp trực quan rõ ràng. Do đó, lý do tôi giới thiệu các phép chiếu phi tuyến tính tiếp theo là vì phương pháp này thường mang lại kết quả tốt hơn trong việc trực quan hóa các vector embedding.

Non-linear Projection

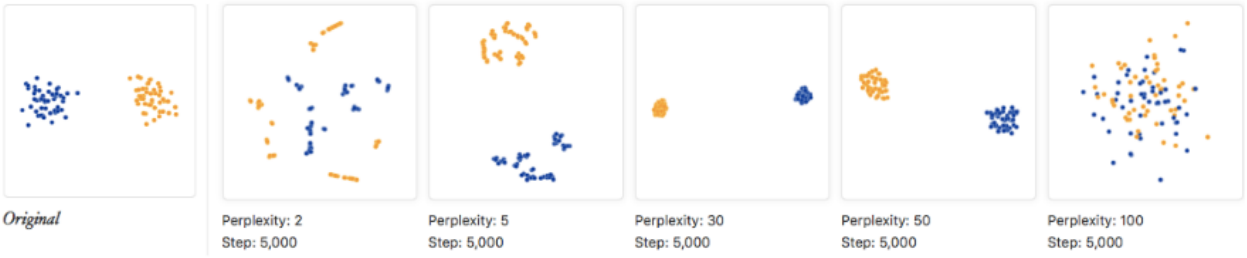
Phép chiếu phi tuyến nhóm các đối tượng gần nhau trong không gian nhiều chiều, ví dụ như SNE/t-SNE (van der Maaten và Hinton 2008) nhóm các đối tượng có xác suất cao với nhau theo phân phối Gaussian. Phương pháp t-SNE là một cách tiêu chuẩn để thực hiện điều này. Nó cố gắng nhóm các đối tượng gần nhau trong không gian nhiều chiều sao cho chúng cũng gần nhau trong không gian ít chiều, nhưng không bị giới hạn bởi tính tuyến tính.



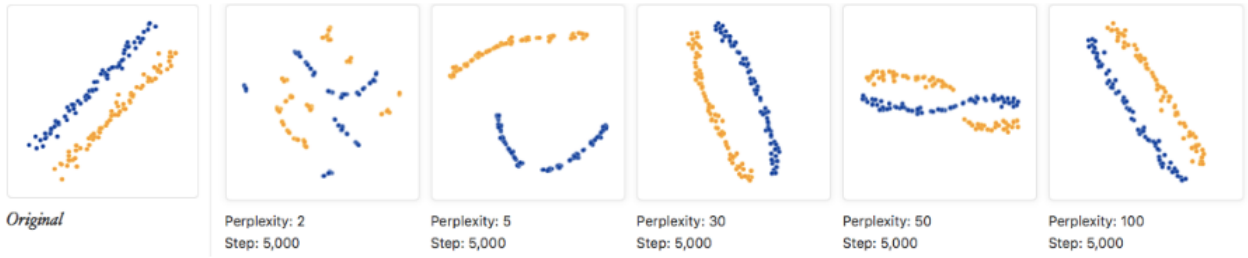
Một ví dụ về việc nhóm các chữ số từ tập dữ liệu MNIST cho thấy rằng khi sử dụng PCA, kết quả là những cụm chồng chéo lên nhau. Ngược lại, khi sử dụng t-SNE, kết quả là các cụm rõ ràng tương ứng với các chữ số thực tế, mặc dù t-SNE là một phương pháp học không giám sát và không huấn luyện mô hình để gán nhãn, các nhãn chỉ được sử dụng để tô màu.

t-SNE Visualization can be Misleading! (Wattenberg et al. 2016)

Tuy nhiên, một vấn đề với t-SNE là các cài đặt của nó rất quan trọng. Các tham số như perplexity tổng thể và số bước thực hiện có thể ảnh hưởng lớn đến kết quả. Có những ví dụ cho thấy nếu thay đổi các tham số này, bạn có thể nhận được những kết quả rất khác nhau. Nếu bạn chạy t-SNE trên dữ liệu gốc, kết quả có thể thay đổi đáng kể dựa trên các tham số hyperparameter mà bạn điều chỉnh.



Ngoài ra, với PCA, bất kể bạn chạy như thế nào, bạn vẫn sẽ nhận được đầu ra tuyến tính. Trong khi đó, t-SNE có thể cho ra những kết quả bất ngờ, thậm chí là những cấu trúc phức tạp như DNA. Do đó, bạn cần cẩn thận vì t-SNE không nhất thiết sẽ cho bạn những mối tương quan tuyến tính rõ ràng.



Giải thích thêm cho Thuật toán t-SNE (Tuỳ chọn)

t-SNE vs PCA

t-SNE và PCA đều là các kỹ thuật giảm chiều dữ liệu, nhưng chúng hoạt động theo những cơ chế khác nhau và phù hợp với các loại dữ liệu khác nhau. PCA (Phân tích thành phần chính) là một kỹ thuật tuyến tính, hiệu quả nhất với dữ liệu có cấu trúc tuyến tính. Nó tìm kiếm các thành phần chính bằng cách chiếu dữ liệu xuống các chiều thấp hơn, tối thiểu hóa phương sai và bảo tồn khoảng cách lớn giữa các cặp điểm. Ngược lại, t-SNE là một kỹ thuật phi tuyến, tập trung vào việc bảo tồn sự tương đồng giữa các điểm dữ liệu trong không gian chiều thấp. t-SNE chú trọng vào việc giữ khoảng cách nhỏ giữa các cặp điểm, trong khi PCA tập trung vào việc duy trì khoảng cách lớn để tối đa hóa phương sai. Tóm lại, PCA bảo tồn phương sai trong dữ liệu, trong khi t-SNE bảo tồn mối quan hệ giữa các điểm dữ liệu, làm cho nó trở thành một thuật toán tốt để trực quan hóa dữ liệu phức tạp có chiều cao.

How t-SNE work?

Thuật toán t-SNE tìm kiếm độ tương đồng giữa các cặp điểm trong không gian có chiều cao và chiều thấp. Quá trình này diễn ra qua ba bước chính. Đầu tiên, t-SNE tính toán độ tương đồng cặp giữa tất cả các điểm dữ liệu trong không gian nhiều chiều bằng cách sử dụng Gaussian kernel, trong đó các điểm gần nhau có xác suất cao hơn để được chọn làm hàng xóm so với các điểm xa.

Tiếp theo, thuật toán cố gắng ánh xạ các điểm dữ liệu từ không gian nhiều chiều sang không gian ít chiều hơn, đồng thời bảo tồn các độ tương đồng cặp. Điều này được thực hiện bằng cách tối thiểu hóa độ phân kỳ giữa phân phối xác suất của dữ liệu gốc và dữ liệu đã giảm chiều. Thuật toán sử dụng phương pháp giảm dần gradient để tối ưu hóa quá trình này, giúp tạo ra một nhúng ở không gian ít chiều ổn định.

Quá trình tối ưu hóa cho phép tạo ra các cụm và cụm con của các điểm dữ liệu tương tự trong không gian ít chiều, từ đó giúp trực quan hóa cấu trúc và mối quan hệ trong dữ liệu nhiều chiều. Nguồn: https://www.datacamp.com/tutorial/introduction-t-sne

Resources

https://phontron.com/class/anlp2024/lectures/#representing-words-jan-18
https://builtin.com/machine-learning/adam-optimization

https://www.datacamp.com/tutorial/introduction-t-sne