ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

TRẦN TRUNG KIÊN

HỌC ĐẶC TRƯNG KHÔNG GIÁM SÁT BẰNG AUTO-ENCODERS

Chuyên ngành: Khoa Học Máy Tính

Mã số chuyên ngành: 60 48 01

LUẬN VĂN THẠC SỸ: KHOA HỌC MÁY TÍNH

NGƯỜI HƯỚNG DẪN KHOA HỌC:

PGS.TS LÊ HOÀI BẮC

Tp. Hồ Chí Minh, Năm 2014

LỜI CẢM ƠN

Trước tiên, em xin gửi lời tri ân sâu sắc đến Thầy Lê Hoài Bắc. Thầy đã rất tận tâm, nhiệt tình hướng dẫn và chỉ bảo em trong suốt quá trình thực hiện luận văn. Không có sự quan tâm, theo dõi chặt chẽ của Thầy chắc chắn em không thể hoàn thành luận văn này.

Em xin chân thành cảm ơn quý Thầy Cô khoa Công Nghệ Thông Tin - trường đại học Khoa Học Tự Nhiên, những người đã ân cần giảng dạy, xây dựng cho em một nền tảng kiến thức vững chắc.

Con xin cảm ơn ba mẹ đã sinh thành, nuôi dưỡng, và dạy dỗ để con có được thành quả như ngày hôm nay. Ba mẹ luôn là nguồn động viên, nguồn sức mạnh hết sức lớn lao mỗi khi con gặp khó khăn trong cuộc sống.

TP. Hồ Chí Minh, 3/2014

Trần Trung Kiên

MỤC LỤC

DANH MỤC HÌNH ẢNH

DANH MỤC BẢNG

Chương 1

Giới thiệu

Nhờ vào những cải cách trong giao thông và cơ sở hạ tầng viễn thông mà giờ đây toàn cầu hóa đang trở nên gần với chúng ta hơn bao giờ hết. Trong xu hướng đó nhu cầu giao tiếp và thông hiểu giữa những nền văn hóa là không thể thiếu. Tuy nhiên, những nền văn hóa khác nhau thường kèm theo đó là sự khác biệt về ngôn ngữ, là một trong những trở ngại lớn nhất của sự giao tiếp. Một người phải mất rất nhiều thời gian để thành thạo một ngôn ngữ không phải là tiếng mẹ đẻ, và không thể nào học được nhiều ngôn ngữ cùng lúc. Cho nên, việc phát triển một công cụ để giải quyết vấn đề này là tất yếu. Một trong những công cụ như vậy là *Dịch máy*.

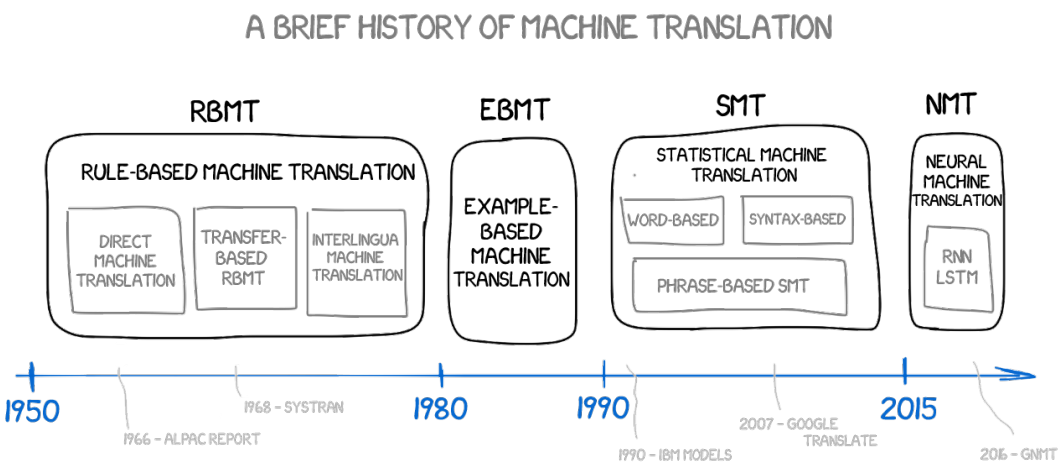
Dịch máy là quá trình chuyển đổi văn bản/tiếng nói từ ngôn ngữ này sang dạng tương ứng của nó trong một ngôn ngữ khác, được thực hiện bởi một chương trình máy tính nhằm mục đích cung cấp bản dịch tốt nhất mà không cần sự trợ giúp của con người.

Dịch máy có một quá trình lịch sử lâu dài. Từ thế kỷ XVII, đã có những ý tưởng về việc cơ giới hóa quá trình dịch thuật. Tuy nhiên, đến thế kỷ XX, những nghiên cứu về dịch máy mới thật sự bắt đầu. Vào những năm 1930, Georges Artsrouni người Pháp và Petr Troyanskii người Nga đã nộp bằng sáng chế cho công trình có tên "máy dịch" của riêng họ. Trong số hai người, công trình của Troyanskii có ý nghĩa hơn. Nó đề xuất không chỉ một phương pháp cho bộ từ điển tự động, mà còn là lược đồ cho việc mã hóa các vai trò ngữ pháp song ngữ và một phác thảo về cách phân tích và tổng hợp có thể hoạt động. Tuy nhiên, những ý tưởng của Troyanskii đã không được biết đến cho đến cuối những năm 1950. Trước đó, máy tính đã được phát minh.

Những nỗ lực xây dựng hệ thống dịch máy bắt đầu ngay sau khi máy tính ra đời.

Có thể nói, chiến tranh và sự thù địch giữa các quốc gia là động lực lớn nhất cho dịch máy thời bấy giờ. Trong Thế chiến thứ II, máy tính đã được quân đội Anh sử dụng trong việc giải mã các thông điệp được mã hóa của quân Đức. Việc làm này có thể coi là một dạng ẩn dụ của dịch máy khi người ta cố gắng dịch từ tiếng Đức được mã hóa sang tiếng Anh. Trong thời kỳ chiến tranh lạnh, vào tháng 7/1949, Warren Weaver, người được xem là nhà tiên phong trong lĩnh vực dịch máy, đã viết một bản ghi nhớ đưa ra các đề xuất khác nhau của ông trong lĩnh vực này. Những đề xuất đó dựa trên thành công của máy phá mã, sự phát triển của lý thuyết thông tin bởi Claude Shannon và suy đoán về các nguyên tắc phổ quát cơ bản của ngôn ngữ. Trong vòng một năm, một vài nghiên cứu về dịch máy đã bắt đầu tại nhiều trường đại học của Mỹ. Vào ngày 7/1/1954, tại trụ sở chính của IBM ở New York, thử nghiệm Georgetown-IBM được tiến hành. Máy tính IBM 701 đã tự động dịch 49 câu tiếng Nga sang tiếng Anh lần đầu tiên trong lịch sử chỉ sử dụng 250 từ vựng và sáu luật ngữ pháp [**?**]. Thí nghiệm này được xem như là một thành công và mở ra kỉ nguyên cho những nghiên cứu với kinh phí lớn về dịch máy ở Hoa Kỳ. Ở Liên Xô những thí nghiệm tương tự cũng được thực hiện không lâu sau đó.

Trong một thập kỷ tiếp theo, nhiều nhóm nghiên cứu về dịch máy được thành lập. Một số nhóm chấp nhận phương pháp thử và sai, thường dựa trên thống kê với mục tiêu là một hệ thống dịch máy có thể hoạt động ngay lập tức, tiêu biểu như: nhóm nghiên cứu tại đại học Washington (và sau này là IBM) với hệ thống dịch Nga-Anh cho Không quân Hoa Kỳ, những nghiên cứu tại viện Cơ học Chính xác ở Liên Xô và Phòng thí nghiệm Vật lý Quốc gia ở Anh. Trong khi một số khác hướng đến giải pháp lâu dài với hướng tiếp cận lý thuyết bao gồm cả những vấn đề liên quan đến ngôn ngữ cơ bản như nhóm nghiên cứu tại Trung tâm nghiên cứu lý thuyết tại MIT, Đại học Havard và Đơn vị nghiên cứu ngôn ngữ Đại học Cambridge. Những nghiên cứu trong giai đoạn này có tầm quan trọng và ảnh hưởng lâu dài không chỉ cho Dịch máy mà còn cho nhiều ngành khác như Ngôn ngữ học tính toán, Trí tuệ nhân tạo - cụ thể là việc phát triển các từ điển tự động và kỹ thuật phân tích cú pháp. Nhiều nhóm nghiên cứu đã đóng góp đáng kể cho việc phát triển lý thuyết ngôn ngữ. Tuy nhiên, mục tiêu cơ bản của dịch máy là xây dựng hệ thống có khả năng tạo ra bản dịch tốt lại không đạt được dẫn đến một kết quả là vào năm 1966 bản báo cáo từ Ủy ban tư vấn xử lý ngôn ngữ tự động (Automatic Language Processing Advisory) của Hoa Kỳ, tuyên bố



Hình 1.1: Lịch sử tóm tắt của dịch máy, nguồn ảnh: Ilya Pestov trong blog A history of machine translation from the Cold War to deep learning

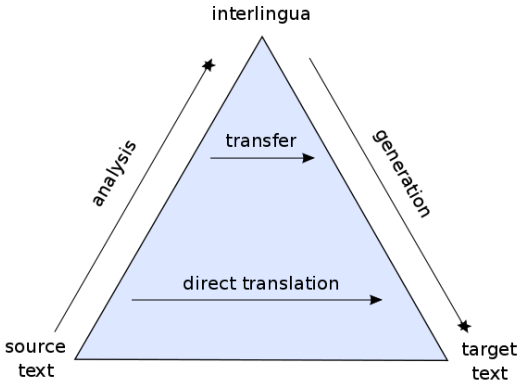
rằng dịch máy là đắt tiền, không chính xác và không mang lại kết quả hứa hẹn [**?**]. Thay vào đó, họ đề nghị tập trung vào phát triển các từ điển, điều này đã loại bỏ các nhà nghiên cứu Mỹ ra khỏi cuộc đua trong gần một thập kỷ.

Các phương pháp Dịch máy

Từ đó đến nay, đã có nhiều hướng tiếp cập đã được sử dụng trong dịch máy với mục tiêu tạo ra bản dịch có độ chính xác cao và giảm thiểu công sức của con người. Trong những năm đầu tiên, để tạo ra bản dịch tốt, các phương pháp thời bấy giờ đều hỏi hỏi những lý thuyết tinh vi về ngôn ngữ học. Hầu hết những hệ thống dịch máy trước những năm 1980 đều là *dịch máy dựa trên luật (Rule-based machine translation - RBMT)*. Những hệ thống này thường bao gồm:

- Một từ điển song ngữ (ví dụ từ điển Anh - Đức)
- Một tập các luật ngữ pháp (ví dụ trong tiếng Đức, từ kết thúc bằng -heit, -keit, -ung là những từ mang giống cái)

Có ba cách tiếp cận khác nhau theo phương pháp dịch máy dựa trên luật. Bao gồm phương pháp dịch máy trực tiếp, dịch máy chuyển giao và dịch máy ngôn ngữ phổ



Hình 1.2: Kim tự tháp của Bernard Vauquois thể hiện ba phương pháp dịch máy dựa luật theo độ sâu của đại diện trung gian. Bắt đầu từ dịch máy trực tiếp đến dịch máy chuyển dịch và trên cùng là dịch máy ngôn ngữ phổ quát (Nguồn: http://en.wikipedia.org/wiki/Machine_translation)

quát. Mặc dù cả ba đều thuộc về RBMT, tuy nhiên chúng khác nhau về độ sâu của đại diện trung gian. Sự khác biệt này được thể hiện qua kim tự tháp Vauquois, minh họa trên hình 1.2

Dịch máy trực tiếp (Direct machine translation - DMT): Đây là phương pháp đơn giản nhất của dịch máy. DMT không dùng bất cứ dạng đại diện nào của ngôn ngữ nguồn, nó chia câu thành các từ, dịch chúng bằng một từ điển song ngữ. Sau đó, dựa trên các luật mà những nhà ngôn ngữ học đã xây dựng, nó chỉnh sửa để bản dịch trở nên đúng cú pháp và ít nhiều đúng về mặt phát âm.

Dịch máy ngôn ngữ phổ quát (Interlingual machine translation - IMT): Trong phương pháp này, câu nguồn được chuyển thành biểu diễn trung gian và biểu diễn này được thống nhất cho tất cả ngôn ngữ trên thế giới (interlingua). Tiếp theo, dạng đại diện này sẽ được chuyển đổi sang bất kỳ ngôn ngữ đích nào. Một trong những ưu điểm chính của hệ thống này là tính mở rộng của nó khi số lượng ngôn ngữ cần dịch tăng lên. Mặc dù trên lý thuyết, phương pháp này trông rất hoàn hảo. Nhưng trong thực tế, thật khó để tạo được một ngôn ngữ phổ quát như vậy.

Dịch máy chuyển giao (Transfer-based machine translation - TMT): dịch máy chuyển giao tương tự như dịch máy ngôn ngữ đại diện ở chỗ, nó cũng tạo ra bản dịch từ biểu diễn trung gian mô phỏng ý nghĩa của câu gốc. Tuy nhiên, không giống như IMT, TMT phụ thuộc một phần vào cặp ngôn ngữ mà nó tham gia vào quá trình dịch. Trên cơ sở sự khác biệt về cấu trúc của ngôn ngữ nguồn và ngôn ngữ đích, một

hệ thống TMT có thể được chia thành ba giai đoạn: i) Phân tích, ii) Chuyển giao, iii) Tạo ra bản dịch. Trong giai đoạn đầu tiên, trình phân tích cú pháp ở ngôn ngữ nguồn được sử dụng để tạo ra biểu diễn cú pháp của câu nguồn. Trong giai đoạn tiếp theo, kết quả của phân tích cú pháp được chuyển đổi thành biểu diễn tương đương trong ngôn ngữ đích. Trong giai đoạn cuối cùng, một bộ phân tích hình thái của ngôn ngữ đích được sử dụng để tạo ra các bản dịch cuối cùng.

Mặc dù đã có một số hệ thống RBMT được đưa vào sử dụng như PROMPT [**?**] và Systrans [**?**]. Tuy nhiên, bản dịch của hướng tiếp cận này có chất lượng thấp so với nhu cầu của con người và không sử dụng được trừ một số trường hợp đặc biệt. Ngoài ra chúng còn có một số nhược điểm lớn như:

- Các loại từ điển chất lượng tốt có sẵn là không nhiều và việc xây dựng những bộ từ điển mới là rất tốn kém.
- Hầu hết những luật ngôn ngữ được tạo ra bằng tay bởi các nhà ngôn ngữ học. Việc này gây khó khăn và tốn kém khi hệ thống trở nên lớn hơn.
- Các hệ thống RBMT gặp khó khăn trong việc giải quyết những vấn đề như thành ngữ hay sự nhập nhằng về ngữ nghĩa của các từ.

Từ những năm 1980, dịch máy dựa trên *Ngữ liệu* (Corpus-based machine translation) được đề xuất. Điểm khác biệt lớn nhất và cũng là quan trọng nhất của hướng tiếp cận này so với RBMT là thay vì sử dụng các bộ từ điển song ngữ, nó dùng những tập câu tương đương trong hai ngôn ngữ làm nền tảng cho việc dịch thuật. Tập những câu tương đương này được gọi là ngữ liệu. So với từ điển, việc thu thập ngữ liệu đơn giản hơn rất nhiều. Ví dụ như ta có thể tìm thấy nhiều phiên bản trong các ngôn ngữ khác nhau của những văn bản hành chính hay các trang web đa ngôn ngữ. Trước khi dịch máy nơ-ron ra đời, dịch máy dựa trên ngữ liệu bao gồm hai phương pháp: dịch máy dựa trên ví dụ và dịch máy thống kê.

Dịch máy dựa trên ví dụ (Example-based Machine Translation - EBMT):

Dịch máy thống kê (Statistical machine translation - SMT): ý tưởng của phương pháp này là thay vì định nghĩa những từ điển và các luật ngữ pháp một cách thủ công, SMT dùng mô hình thống kê để học các từ điển và các luật ngữ pháp này từ ngữ liệu. Những ý tưởng đầu tiên của SMT được giới thiệu đầu tiên bởi Waren Weaver vào năm

1949 bap gồm việc áp dụng lý thuyết thông tin của Claude Shannon vào dịch máy. SMT được giới thiệu lại vào cuối những năm 1980 và đầu những năm 1990 tại trung tâm nghiên cứu Thomas J. Watson của IBM. SMT là phương pháp được nghiên cứu rộng rãi nhất thời bấy giờ và thậm chí đến hiện tại, nó vẫn là một trong những phương pháp được nghiên cứu nhiều nhất về dịch máy.

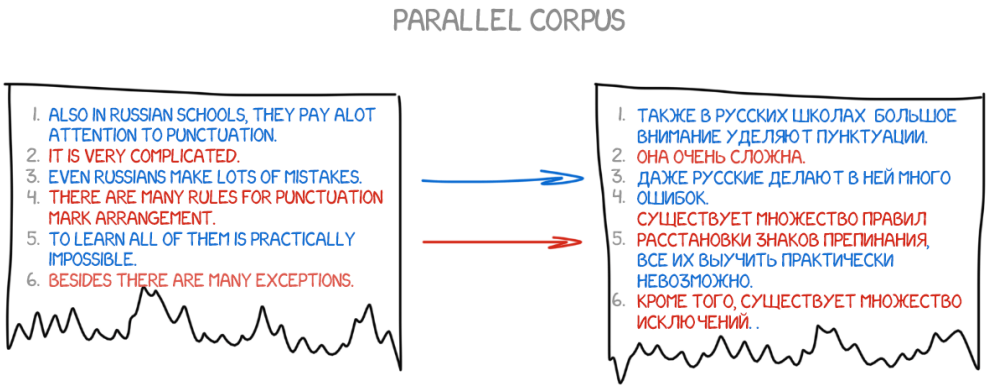
Để hiểu rõ hơn về dịch máy thống kê, xét một ví dụ: ta cần dịch một câu f trong tiếng Pháp sang dạng tiếng Anh e của nó. Có nhiều bản dịch có thể có cuả f trong tiếng Anh, việc cần làm là chọn e sao cho nó là bản dịch "tốt nhất" của f. Chúng ta có thể mô hình hóa quá trình này bằng một xác suất có điều kiện $p(e|f)$ với e là những bản dịch có thể có với câu cho trước f. Một cách hợp lý để chọn bản dịch "tốt nhất" là chọn e sao cho nó tối đa xác suất có điều kiện $p(e|f)$. Cách tiếp cận quen thuộc là sử dụng định lý Bayes để viết lại $p(e|f)$:

$$p(e|f) = \frac{p(f|e)p(e)}{p(f)} \tag{1.1}$$

Bởi vì f là cố định, tối đa hóa $p(e|f)$ tương đương với tìm e sao cho tối đa hóa $p(f|e)p(e)$. Để làm được điều này, chúng ta dựa vào một tập ngữ liệu là những câu song ngữ Anh - Pháp để suy ra các mô hình $p(f|e)$ và $p(e)$ và sử dụng những mô hình đó để tìm một bản dịch cụ thể $\tilde{e}$ sao cho:

$$\tilde{e} = \arg\max_{e \in e^*} p(e|f) = \arg\max_{e \in e^*} p(f|e)p(e) \tag{1.2}$$

Ở đây, $p(f|e)$ được gọi là *mô hình dịch* (translation model) và $p(e)$ được gọi là *mô hình ngôn ngữ* (language model). Mô hình dịch $p(f|e)$ thể hiện khả năng câu e là một bản dịch của câu f. Những mô hình dịch ban đầu dựa trên từ (word-based) như các mô hình IBM 1-5 (IBM Models 1-5). Những năm 2000, những mô hình dịch dựa trên cụm từ (phrase based) xuất hiện giúp cải thiện khả năng dịch của SMT. Trong khi đó, mô hình ngôn ngữ $p(e)$ thể hiện độ trơn tru của câu e. Ví dụ $p(\text{"tôi đi học"}) > p(\text{"học tôi đi"})$ vì rõ ràng "tôi đi học" là có lý hơn "học tôi đi". Các mô hình ngôn ngữ cho SMT thường được ước lượng bằng các mô hình n-gram được làm mịn, cách làm này cũng là một nhược điểm của SMT. Mô hình ngôn ngữ là một chủ đề quan trọng và sẽ được chúng tôi đề cập lại một lần nữa trong chương Kiến thức nền tảng.



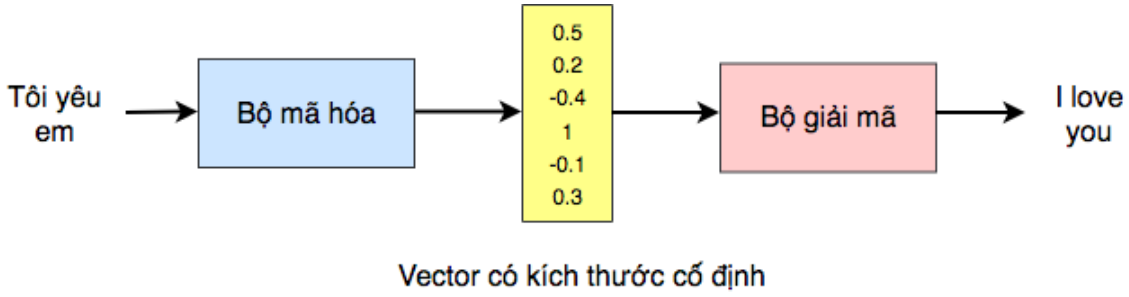
Hình 1.3: Ví dụ về tập các câu song song trong hai ngôn ngữ

Dịch máy Nơ-ron

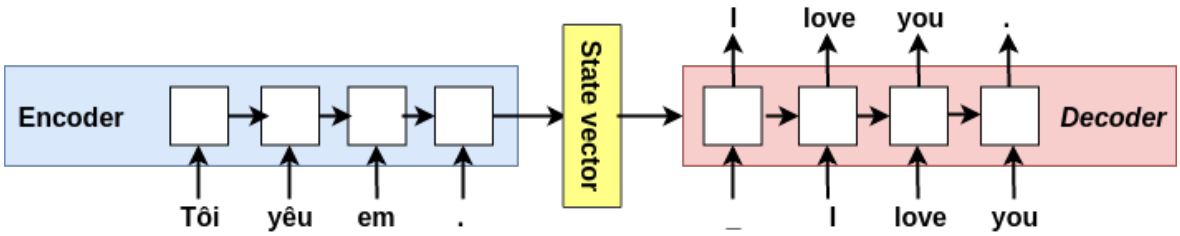
Mặc dù trên thực tế đã có nhiều hệ thống dịch máy được phát triển dựa trên dịch máy thống kê thời bấy giờ, tuy nhiên nó không hoạt động thực sự tốt bởi một số nguyên nhân. Một là việc những từ hay đoạn được dịch cục bộ và quan hệ của chúng với những từ cách xa trong câu nguồn thường bị bỏ qua. Hai là mô hình ngôn ngữ N-gram hoạt động không thực sự tốt đối với những bản dịch dài và ta phải tốn nhiều bộ nhớ để lưu trữ chúng. Ngoài ra việc sử dụng nhiều thành phần nhỏ được điều chỉnh riêng biệt như mô hình dịch, mô hình ngôn ngữ,.. cũng gây khó khăn cho việc vận hành và phát triển mô hình này.

Dịch máy nơ-ron (Neural machine translation) là một hướng tiếp cận mới trong dịch máy trong những năm gần đây được đề xuất đầu tiên bởi [**?**], [**?**], [**?**]. Giống như dịch máy thống kê, dịch máy nơ-ron cũng là một phương pháp thuộc hướng tiếp cận dựa trên ngữ liệu, trong khi dịch máy thống kê bao gồm nhiều mô-đun nhỏ được điều chỉnh riêng biệt, Dịch máy nơ-ron cố gắng dùng một mạng nơ-ron như là thành phần duy nhất của hệ thống, mọi thiết lập sẽ được thực hiện trên mạng này.

Hầu hết những mô hình dịch máy nơ-ron đều dựa trên kiến trúc *Bộ mã hóa - Bộ giải mã* (Encoder-Decoder) ([**?**], [**?**]). Bộ mã hóa thường là một mạng nơ-ron có tác dụng *"nén"* tất cả thông tin của câu trong ngôn ngữ nguồn vào một vector có kích thước cố định. Bộ giải mã, cũng là một mạng nơ-ron, sẽ tạo bản dịch trong ngôn ngữ đích từ vector có kích thước cố định kia. Toàn bộ hệ thống bao gồm bộ mã hóa và bộ giải mã sẽ được huấn luyện *"end-to-end"* để tạo ra bản dịch, quá trình này được mô tả



Hình 1.4: Ví dụ về kiến trúc bộ mã hóa - bộ giải mã trong dịch máy nơ-ron

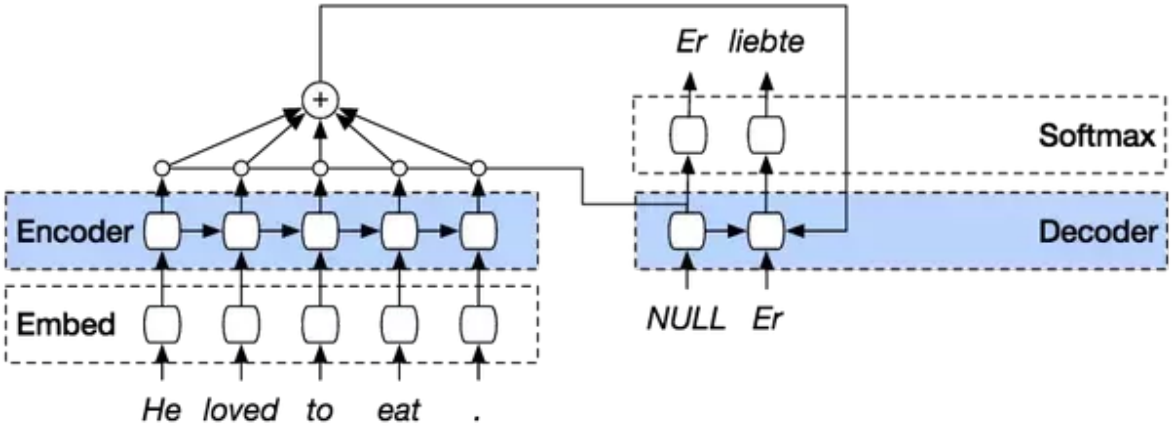


Hình 1.5: Kiến trúc bộ mã hóa - bộ giải mã được xây dựng trên mạng nơ-ron hồi quy

như hình 1.2.

Trong thực tế cả bộ mã hóa và giải mã thường dựa trên một mô hình mạng nơ-ron tên là *Mạng nơ-ron hồi quy* là một thiết kế mạng đặc trưng cho việc xử lý dữ liệu chuỗi. Mạng nơ-ron hồi quy cho phép chúng ta mô hình hóa những dữ liệu có độ dài không xác định, rất thích hợp cho bài toán dịch máy. Hình 1.3 mô tả chi tiết hơn về kiến trúc bộ mã hóa - giải mã sử dụng mạng nơ-ron hồi quy. Đầu tiên bộ mã hóa đọc qua toàn bộ câu nguồn và tạo ra một vector đại diện gọi là *vector trạng thái*. Điều này giúp cho toàn bộ những thông tin cần thiết hay quan hệ giữa các từ đều được tập hợp vào một nơi duy nhất. Bộ giải mã, lúc này đóng vai trò như một mô hình ngôn ngữ để tạo ra từng từ trong ngôn ngữ đích và sẽ dừng lại đến khi một ký tự đặc biệt xuất hiện.

Trong hình 2, có thể thấy rằng bộ giải mã tạo ra bản dịch chỉ dựa trên trạng thái ẩn cuối cùng, cũng chính là vector có kích thước cố định được tạo ra ở bộ mã hóa. Vector này phải mã hóa mọi thứ chúng ta cần biết về câu nguồn. Giả sử chúng ta có câu nguồn với độ dài là 50 từ, từ đầu tiên ở câu đích có lẽ sẽ có mối tương quan cao với từ đầu tiên ở câu nguồn. Điều này có nghĩa là bộ giải mã phải xem xét thông tin được mã hóa từ 50 *"time step"* trước đó. Mạng nơ-ron hồi quy được chứng minh là gặp khó khăn trong việc mã hóa những chuỗi dài [**?**]. Để giải quyết vấn đề này, thay vì dùng mạng nơ-ron hồi quy thuần, người ta sử dụng các biến thể của nó quy như



Hình 1.6: Cơ chế Attention trong dịch máy nơ-ron

Long short-term memory (LSTM). Trên lý thuyết, LSTM có thể giải quyết vấn đề mất mát thông tin trong chuỗi dài, nhưng trong thực tế vấn đề này vẫn chưa thể được giải quyết hoàn toàn. Một số nhà nghiên cứu đã phát hiện ra rằng đảo ngược chuỗi nguồn trước khi đưa vào bộ mã hóa tạo ra kết quả tốt hơn một cách đáng kể [**?**] bởi nó khiến cho những từ đầu tiên được đưa vào bộ mã hóa sau cùng, và được giải mã thành từ tương ứng ngay sau đó. Cách làm này tuy giúp cho bản dịch hoạt động tốt hơn trong thực tế, nhưng nó không phải là một giải pháp về mặt thuật toán. Hầu hết các đánh giá về dịch máy được thực hiện trên các ngôn ngữ như ngôn ngữ có trật tự câu tương đối giống nhau. Ví dụ trật tự dạng "chủ ngữ - động từ - vị ngữ" như tiếng Anh, Đức, Pháp hay Trung Quốc. Đối với dạng ngôn ngữ có một trật tự khác ví dụ "chủ ngữ - vị ngữ - động từ" như tiếng Nhật, đảo ngược câu nguồn sẽ không hiệu quả.

Attention là cơ chế giải phóng kiến trúc bộ mã hóa - bộ giải mã khỏi nhược điểm chỉ sử dụng một vector có chiều dài cố định làm đại diện cho câu đầu vào. Ý tưởng chính của cơ chế này là ở mỗi thời điểm phát sinh các từ trong bản dịch, bộ giải mã sẽ "nhìn" vào các phần khác nhau của câu nguồn trong quá trình mã hóa. Quan trọng hơn, cơ chế này cho phép mô hình học được cách chọn những phần cần thiết để tập trung vào dựa trên câu nguồn và những gì mà bộ giải mã đã giải mã được.

Cấu trúc của khóa luận

Trong khóa luận này, chúng tôi quyết định tập trung nghiên cứu về dịch máy nơ-ron và cơ chế Attention dựa trên nghiên cứu của nhóm tác giả tại đại học Stanford bao gồm

Minh-Thang Luong, Hieu Pham, Christopher Manning trong bài báo *Effective Approaches to Attention-based Neural Machine Translation* [**?**]. Các phần còn lại trong luận văn được trình bày như sau:

- Chương 2 trình bày về những thành nền tảng của kiến trúc bộ mã hóa - giải mã
- Chương 3 trình bày về cơ chế Attention, đây là phần chính của luận văn. Trong phần này gồm có hai phần nhỏ:
  - *Global attetion*: là cơ chế tập trung vào tất cả các trạng thái ở câu nguồn
  - *Local attetion*: tập trung vào một tập các trạng thái ở câu nguồn tại một thời điểm
- Chương 4 trình bày về các thí nghiệm và các phân tích về kết quả đạt trên hai tập dữ liệu Anh-Đức, Anh-Việt.
- Kết luận và hướng phát triển của luận văn.

Chương 2

Kiến Thức Nền Tảng

Trong chương này, chúng tôi sẽ trình bày những kiến thức nền tảng trên ba chủ đề bao gồm mạng nơ-ron hồi quy, mô hình ngôn ngữ nơ-ron và mô hình dịch máy nơ-ron. Mạng nơ-ron hồi quy (RNN) là xương sống của dịch máy nơ-ron. Nó được sử dụng để làm cả bộ mã hóa lẫn bộ giải mã. Ứng với mỗi vai trò, RNN sẽ có một thiết kế riêng. Một phiên bản cải tiến của RNN là *Long short-term memory* cũng được chúng tôi trình bày, phiên bản này giúp cho việc huấn luyện RNN trở nên dễ dàng hơn. Sau đó, dựa trên những kiến thức về mạng nơ-ron hồi quy, chúng tôi nói về khái niệm *mô hình ngôn ngữ* với chức năng tạo ra từ trong bộ giải mã, là bước quan trọng trong dịch máy nơ-ron. Cuối cùng, chúng tôi cũng trình bày về mô hình dịch máy nơ-ron theo kiến trúc bộ mã hóa - bộ giải mã với RNN và mô hình ngôn ngữ hồi quy là những thành phần nền tảng.

Mạng nơ-ron hồi quy (Recurrent neural network)

Trong tự nhiên, dữ liệu không phải lúc nào cũng được sinh ra một cách ngẫu nhiên. Trong một số trường hợp, chúng được sinh ra theo một thứ tự. Xét trong dữ liệu văn bản, ví dụ ta cần điền vào chỗ trống cho câu sau *"Paris là thủ đô của nước __"*. Dễ biết được rằng chỉ có duy nhất một từ phù hợp cho chỗ trống này, đó là *"Pháp"*. Điều này có nghĩa là mỗi từ trong một câu không được tạo ra ngẫu nhiên mà nó được tạo ra dựa trên một liên hệ với những từ đứng trước nó. Các loại dữ liệu khác như những khung hình trong một bộ phim hoặc các đoạn âm thanh trong một bản nhạc cũng có

tính chất tương tự. Những loại dữ liệu mang thứ tự này được gọi chung là dữ liệu chuỗi (sequential data).

Trong quá khứ, một số mô hình xử lý dữ liệu chuỗi bằng cách giả định rằng đầu vào hiện tại có liên hệ với một số lượng xác định đầu vào trước đó, nhiều mô hình tạo ra một cửa sổ trượt để nối mỗi đầu vào hiện tại với một số lượng đầu vào trước đó nhằm tạo ra sự mô phỏng về tính phụ thuộc. Cách tiếp cận này đã được sử dụng cho mô hình *Deep belief network* trong xử lý tiếng nói [**?**]. Nhược điểm của những cách làm này là ta phải xác định trước kích thước của cửa sổ. Một mô hình với kích thước cửa sổ với chiều dài bằng 6 không thể nào quyết định được từ tiếp theo trong câu *"Hổ là loài động vật ăn __"* sẽ là *"thịt"* hay *"cỏ"*. Trong ví dụ này, từ tiếp theo của câu phụ thuộc mật thiết vào từ *"Hổ"* cách nó đúng 6 từ. Trên thực tế, có rất nhiều câu đòi hỏi sự phụ thuộc với nhiều từ xa hơn trước đó. Ta gọi những sự phụ thuộc kiểu như vậy là những *phụ thuộc dài hạn* (long term dependency).

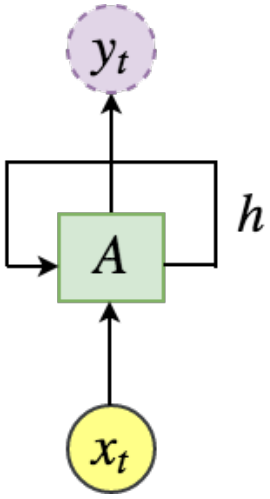
Mạng nơ-ron hồi quy (recurrent neural network) [**?**] gọi tắt là *RNN* là một nhánh của nạng nơ-ron nhân tạo được thiết kế đặc biệt cho việc mô hình hóa dữ liệu chuỗi. Khác với những mô hình đã đề cập giả định sự phụ thuộc chỉ xảy ra trong một vùng có chiều dài cố định. RNN, trên lý thuyết, có khả năng nắm bắt được các phụ thuộc dài hạn với chiều dài bất kỳ. Để làm được điều đó, trong quá trình học, RNN lưu giữ những thông tin cần thiết cho các phụ thuộc dài hạn bằng một vec-tơ được gọi là *trạng thái ẩn*.

Xét một chuỗi đầu vào $x = x_1, x_2, ..., x_n$. Ta gọi h_t là trạng thái ẩn tại *bước thời gian* (timestep) t, là lúc một mẫu dữ liệu x_t được đưa vào RNN để xử lý. Trạng thái ẩn h_t sẽ được tính toán dựa trên mẫu dữ liệu hiện tại x_t và trạng thái ẩn trước đó h_{t-1}. Có thể thể hiện h_t như một hàm hồi quy với tham số là đầu vào hiện tại và chính nó ở thời điểm trước đó:

$$h_t = f(h_{t-1}, x_t) \tag{2.1}$$

trong đó hàm f là một ánh xạ phi tuyến. Có thể hình dung h_t như một đại diện cho những đầu vào mà nó đã xử lý từ thời điểm ban đầu cho đến thời điểm t. Nói một cách khác, RNN sử dụng trạng thái ẩn như một dạng bộ nhớ để lưu giữ thông tin từ một chuỗi.

Thông thường, hàm f là một hàm phi tuyến như hàm σ hay hàm tanh. Xét một



Hình 2.1: Mô hình RNN đơn giản với kết nối vòng, h được xem như bộ nhớ được luân chuyển trong RNN. Chú ý rằng đường nét đứt ở đầu ra thể hiện rằng tại một thời điểm t, RNN có thể có hoặc không có một đầu ra.

RNN với công thức cụ thể như sau:

$$h_t = \tanh\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right) \tag{2.2}$$

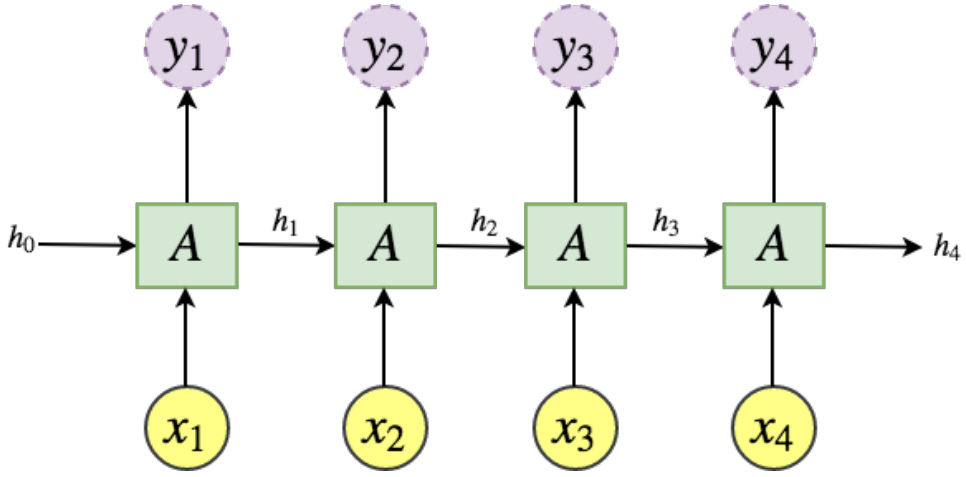
Ở đây W_{xh} là ma trận trọng số kết nối giữa đầu vào và trạng thái ẩn, W_{hh} là ma trận trọng số hồi quy kết nối trạng thái ẩn với chính nó trong các bước thời gian liền kề. Vec-tơ b_h là vec-tơ "bias" của trạng thái ẩn. Tại thời điểm bắt đầu, trạng thái ẩn h_0 có thể được khởi tạo bằng 0 hoặc là một vector chứa tri thức có sẵn như trường hợp của bộ giải mã như chúng tôi đã đề cập trong chương 1.

Tại mỗi bước thời gian t, tùy vào mục tiêu cụ thể của quá trình học mà RNN có thể có thêm một đầu ra y_t. Trong ngữ cảnh bài toán dịch máy nơ-ron, đầu ra của RNN trong quá trình giải mã chính là một từ trong ngôn ngữ đích hay nói chung là một đầu ra dạng rời rạc. Với mục tiêu đó, đầu ra dự đoán của RNN $\hat{y}_t$ sẽ có dạng là một phần phối xác suất trên tập các các lớp ở đầu ra. Phân phối này nhằm dự đoán vị trí xuất hiện của $\hat{y}_t$.

$$s_t = W_{hy}h_t + b_y \tag{2.3}$$

$$\hat{y}_t = softmax(s_t) \tag{2.4}$$

Trong công thức trên, W_{hy}, b_y là những tham số gắn với đầu ra của mô hình. W_{hy} là ma trận trọng số kết nối đầu ra, b_y là vec-tơ "bias" ở đầu ra. s_t là một vector có độ



Hình 2.2: Mô hình RNN được dàn trải (unrolled), ví dụ trong 4 bước thời gian.

dài bằng với là số lượng lớp ở đầu ra. Vector này được chuẩn hóa thành một phân phối xác suất $\hat{y}_t$ bằng một hàm *softmax*.

Để ý rằng các ma trận trọng số W_{xh}, W_{hh}, W_{hy} và các vector bias b_h, b_y là các tham số học của mô hình và chúng là duy nhất. Có nghĩa là khi những tham số này được học, bất kỳ một đầu vào nào cũng đều sử dụng chung một bộ tham số. Điều này chính là *sự chia sẻ tham số* (parameters sharing) trong mạng nơ-ron hồi quy. Chia sẻ tham số khiến cho mô hình học dễ dàng hơn, nó giúp cho RNN có thể xử lý chuỗi đầu vào với độ dài bất kỳ mà không làm tăng độ phức tạp của mô hình. Quan trọng hơn, nó giúp ích cho việc tổng quát hóa. Đây chính là điểm đặc biệt của RNN so với mạng nơ-ron truyền thẳng.

Hình 2.1 thể hiện định nghĩa hồi quy của RNN, và nó đúng cho chuỗi có độ dài bất kỳ. Với một số lượng hữu hạn các bước thời gian, mô hình RNN có thể được dàn trải ra (unrolled). Dạng dàn trải của RNN được miêu tả trực quan như trên hình 2.2. Với cách thể hiện này, RNN có thể được hiểu như là một mạng nơ-ron sâu với một tầng cho mỗi bước thời gian và các tham số học được chia sẻ giữa các bước thời gian đó. Dạng dàn trải cũng thể hiện rằng RNN có thể được huấn luyện qua nhiều bước thời gian bằng thuật toán lan truyền ngược (backpropagation). Thuật toán này được gọi là *Lan truyền ngược qua thời gian* (Backpropagation through time) [**?**]. Tất cả các mạng nơ-ron hồi quy phổ biến ngày nay đều áp dụng thuật toán này.

Huấn luyện mạng nơ-ron hồi quy

Long short-term memory

Hochreiter và Schmidhuber [1997] đã giới thiệu mô hình LSTM chủ yếu ở để khắc phục vấn đề biến mất gradient. Mô hình này tương tự với mô hình

Mô hình ngôn ngữ

Như chúng tôi đã đề cập trong chương 1, mô hình ngôn ngữ là một bộ phận quan trọng trong dịch máy thống kê nhằm bảo đảm tạo ra một bản dịch trơn tru. Một cách cụ thể, nhiệm vụ của mô hình ngôn ngữ là gán cho một câu một xác suất thể hiện độ khả dĩ của câu đó. Bên cạnh gán cho câu một xác suất, mô hình ngôn ngữ cũng gán một xác suất thể hiện khả năng một từ (hoặc một chuỗi từ) xuất hiện theo sau một chuỗi từ cho trước.

Xét một chuỗi các từ bất kỳ $w_{1:n}$,

Chương 3

Các Kết Quả Thí Nghiệm

Trong chương này, chúng tôi trình bày các kết quả thí nghiệm để đánh giá các đề xuất đã được nói ở chương trước. Bộ dữ liệu được dùng để tiến hành các thí nghiệm là bộ MNIST (bộ ảnh chữ số viết tay gồm các chữ số từ 0 đến 9). Các kết quả thí nghiệm cho thấy khi huấn luyện "Sparse Rectified Auto-Encoders" (SRAEs) với chuẩn L1 sẽ gặp phải vấn đề nơ-ron "ngủ", và chiến lược "ngủ - đánh thức" trong thuật toán "Sleep-Wake Stochastic Gradient Descent" (SW-SGD) của chúng tôi có thể giúp khắc phục vấn đề này. Các kết quả thí nghiệm cũng cho thấy cách ràng buộc trọng số đề xuất của chúng tôi cho kết quả tốt nhất trong số các cách ràng buộc trọng số có thể áp dụng cho SRAEs. Cuối cùng, thí nghiệm cũng cho thấy SRAEs với hai đề xuất trên của chúng tôi (SW-SGD và cách ràng buộc trọng số) có thể học được những đặc trưng cho kết quả phân lớp tốt khi so sánh với các loại "Auto-Encoders" khác.

Các thiết lập thí nghiệm

Chúng tôi tiến hành các thí nghiệm trên bộ dữ liệu MNIST [**?**]; bộ dữ liệu này gồm các ảnh xám (có kích thước 28×28) của mười chữ số viết tay từ 0 đến 9. Ở hình 3.1 là một số ảnh mẫu của bộ dữ liệu này. Dữ liệu được tiến hành tiền xử lý bằng cách lấy mỗi giá trị điểm ảnh chia cho 255 để đưa về đoạn $[0, 1]$. Chúng tôi sử dụng cách phân chia thường được sử dụng cho bộ dữ liệu này: 50000 ảnh dùng để huấn luyện, 10000 ảnh dùng để chọn các siêu tham số (validation), và 10000 ảnh dùng để kiểm tra (test).

Hình 3.1: Một số ảnh mẫu của bộ dữ liệu MNIST

Chúng tôi sử dụng ngôn ngữ lập trình Theano [**?**] bởi vì ngôn ngữ này cho phép dễ dàng cài đặt các thuật toán và dễ dàng sử dụng GPU (Graphical Processing Units) để tính toán song song. Loại GPU mà chúng tôi sử dụng là NVIDIA GTX 560.

Sau khi tiến hành xong bước học đặc trưng không giám sát, chúng tôi đánh giá các đặc trưng học được bằng cách sử dụng chúng để huấn luyện mô hình phân lớp "Softmax Regression" và đo độ lỗi phân lớp. Một cách cụ thể, cho tập huấn luyện $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$ với $x^{(i)} \in \mathbb{R}^{D_x}$ là véc-tơ điểm ảnh và $y^{(i)} \in \{0, \ldots, 9\}$ là nhãn lớp. Sau khi "Auto-Encoder" đã được huấn luyện trên tập không có nhãn $\{x^{(1)}, \ldots, x^{(N)}\}$, ta lần lượt đưa từng véc-tơ $x^{(i)}$ vào "Auto-Encoder" và thu được ở tầng ẩn véc-tơ đặc trưng tương ứng $h^{(i)}$; bằng cách này, ta có được tập huấn luyện mới $\{(h^{(1)}, y^{(1)}), \ldots, (h^{(N)}, y^{(N)})\}$. Kế đến, tập huấn luyện mới này được sử dụng để huấn luyện "Softmax Regression". Để dự đoán nhãn lớp cho một véc-tơ đầu vào mới x_{test}, đầu tiên ta sử dụng "Auto-Encoder" đã được huấn luyện để tính véc-tơ đặc trưng tương ứng h_{test}; sau đó đưa h_{test} này vào "Softmax Regression" đã được huấn luyện để tính giá trị nhãn lớp dự đoán.

Trong cả hai giai đoạn học không giám sát và có giám sát, chúng tôi sử dụng

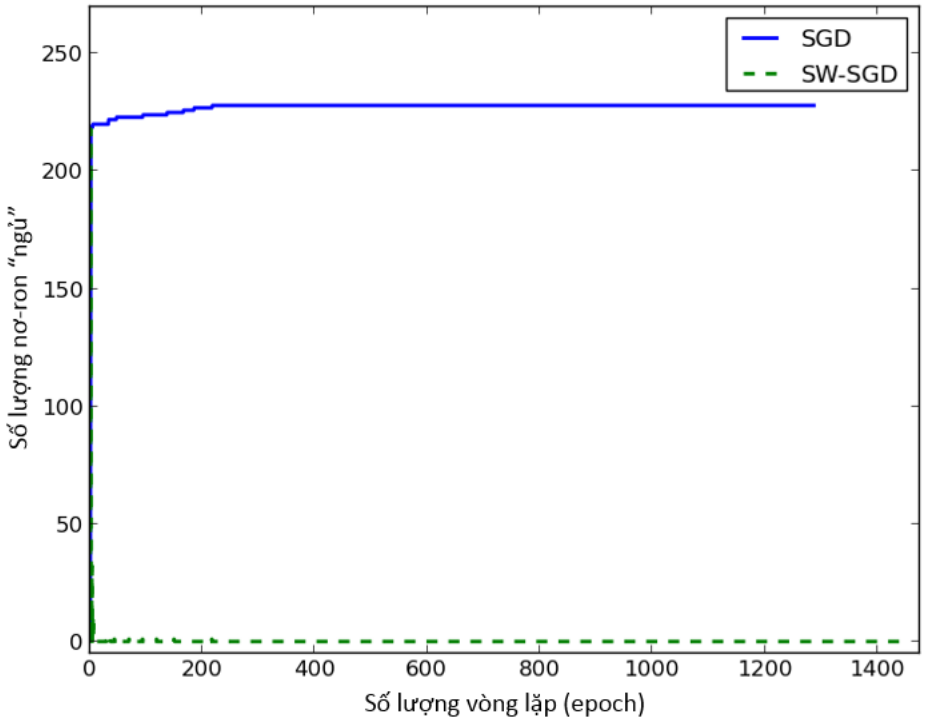
thuật toán để để cực tiểu hóa hàm chi phí là Stochastic Gradient Descent (SGD) với kích thước của "mini-batch" là 100 mẫu huấn luyện. Chiến lược "dừng sớm" (early stopping) được sử dụng để quyết định số vòng lặp (epoch) của SGD cũng như là để chống vấn đề quá khớp (trong giai đoạn học không giám sát, chúng tôi dừng quá trình tối ưu hóa dựa vào giá trị của hàm chi phí trên tập "validation"; còn trong giai đoạn học có giám sát, chúng tôi dựa vào độ lỗi phân lớp trên tập "validation"). Trong tất cả các thí nghiệm dưới đây, chúng tôi dùng SRAEs với 1000 nơ-ron ẩn, tham số "thỏa hiệp" giữa độ lỗi tái tạo và độ thưa λ bằng 0.25, hệ số học khi học không giám sát bằng 0.05, và hệ số học khi học có giám sát bằng 1 (số lượng nơ-ron ẩn được chọn theo [**?**], các siêu tham số còn lại được chọn dựa vào thực nghiệm).

SGD và SW-SGD

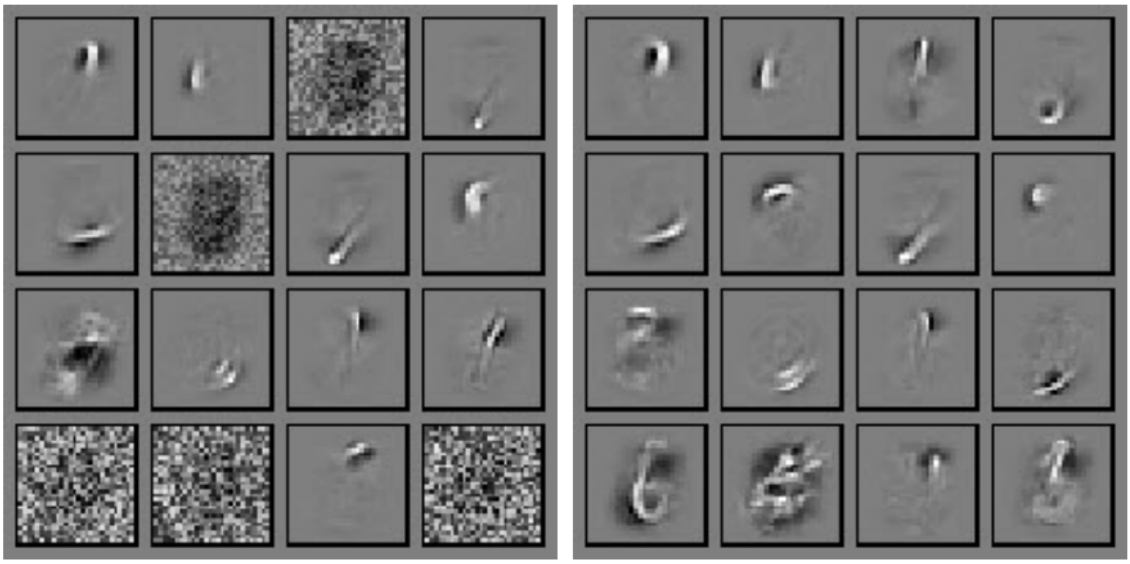
Để thấy được vấn đề gặp phải khi huấn luyện SRAEs với ràng buộc thưa bằng chuẩn L1 cũng như là tác dụng của chiến lược "ngủ - đánh thức" của chúng tôi, trong phần này chúng tôi so sánh việc huấn luyện SRAEs bằng thuật toán "Stochastic Gradient Descent" (SGD) và phiên bản điều chỉnh của chúng tôi, "Sleep-Wake Stochastic Gradient Descent" (SW-SGD). Trong thí nghiệm này, cách ràng buộc trọng số đề xuất của chúng tôi được sử dụng ($W^{(d)} = (W^{(e)})^T$, và các dòng của $W^{(e)}$ và các cột của $W^{(d)}$ được chuẩn hóa).

Hình 3.2 thể hiện số lượng nơ-ron "ngủ" của SRAEs trong khi thực hiện quá trình tối ưu hóa hàm chi phí với SGD và SW-SGD. Vấn đề gặp phải khi huấn luyện SRAEs với chuẩn L1 là trong quá trình tối ưu hóa, chuẩn L1 có thể đẩy các véc-tơ trọng số đi vào các nơ-ron ẩn vào trạng thái "ngủ" (nghĩa là, nơ-ron ẩn tương ứng luôn cho giá trị đầu ra bằng 0 với tất cả các mẫu huấn luyện) và sau đó, chúng sẽ không bao giờ còn được cập nhật nữa. Như có thể thấy từ hình 3.2, khi sử dụng SGD, số lượng nơ-ron "ngủ" tăng dần trong quá trình tối ưu hóa, đặc biệt là trong những vòng lặp đầu tiên, khi mà quá trình tối ưu hóa vẫn còn chưa ổn định. Vấn đề nơ-ron "ngủ" này của chuẩn L1 có thể được khắc phục một cách đơn giản bằng chiến lược "ngủ - đánh thức" của chúng tôi; quá trình tối ưu hóa của SW-SGD kết thúc mà không có nơ-ron "ngủ" nào cả.

Ở hình 3.3 là một số bộ lọc (một bộ lọc tương ứng với véc-tơ trọng số đi vào một



Hình 3.2: Số lượng nơ-ron "ngủ" của SRAEs trong khi thực hiện quá trình tối ưu hóa với SGD và với SW-SGD. Quá trình tối ưu hóa của SGD kết thúc với 228 nơ-ron "ngủ" trong tổng số 1000 nơ-ron; trong khi đó, SW-SGD kết thúc mà không có nơ-ron nào "ngủ". (Hai quá trình tối ưu hóa của SGD và SW-SGD kết thúc sau các số lượng vòng lặp khác nhau là do chiến lược "dừng sớm".)



(a) SGD (b) SW-SGD

Hình 3.3: Ở hình (a) là một số bộ lọc (một bộ lọc tương ứng với véc-tơ trọng số đi vào một nơ-ron ẩn) học được bởi SGD; ta có thể thấy có 5 bộ lọc nhìn vô nghĩa tương ứng với 5 nơ-ron "ngủ". Còn ở hình (b) là các bộ lọc học được bởi SW-SGD; tất cả các bộ lọc đều nhìn có nghĩa, mỗi bộ lọc dò tìm một đường nét nào đó của chữ số.

Bảng 3.1: Giá trị hàm chi phí của SRAEs trên tập huấn luyện và độ lỗi phân lớp (với "Softmax Regression") trên tập kiểm tra khi huấn luyện SRAEs với SGD và với SW-SGD.

	SGD	SW-SGD
Giá trị hàm chi phí của SRAEs trên tập huấn luyện	9.84	**9.48**
Độ lỗi phân lớp trên tập kiểm tra (%)	1.70	**1.62**

nơ-ron ẩn) học được bởi SGD và SW-SGD. Như ta có thể thấy, với SGD, có 5 nơ-ron "ngủ"; các bộ lọc của chúng nhìn vô nghĩa. Với SW-SGD, không có nơ-ron nào "ngủ"; tất cả các bộ lọc đều nhìn có nghĩa, mỗi bộ lọc dò tìm một đường nét nào đó của chữ số.

Nhờ sử dụng hết tất cả các nơ-ron ẩn, SW-SGD tìm được giá trị cực tiểu của hàm chi phí của SRAEs trên tập huấn luyện tốt hơn so với SGD; và các đặc trưng học được của SW-SGD cũng cho kết quả phân lớp (với "Softmax Regression") trên tập kiểm tra tốt hơn so với SGD (bảng 3.1).

Cách ràng buộc trọng số đề xuất của chúng tôi và các cách ràng buộc trọng số khác

Trong thí nghiệm thứ hai này, cách ràng buộc trọng số đề xuất cho SRAEs của chúng tôi được so sánh với các cách ràng buộc trọng số khác mà có thể áp dụng cho SRAEs. Cụ thể ở đây, chúng tôi so sánh với các cách ràng buộc trọng số sau:

- $W^{(d)}$ **được chuẩn hóa:** các véc-tơ cột của $W^{(d)}$ được ràng buộc là chuẩn hóa (có độ dài bằng 1); mỗi véc-tơ cột của $W^{(d)}$ tương ứng với véc-tơ trọng số đi ra ở mỗi nơ-ron ẩn.

- $W^{(e)}$ **và** $W^{(d)}$ **được chuẩn hóa:** các véc-tơ dòng của $W^{(e)}$ và các véc-tơ cột của $W^{(d)}$ được ràng buộc là chuẩn hóa (có độ dài bằng 1); mỗi véc-tơ dòng của $W^{(e)}$ và mỗi véc-tơ cột của $W^{(d)}$ lần lượt tương ứng với véc-tơ trọng số đi vào và véc-tơ trọng số đi ra ở mỗi nơ-ron ẩn.

- $W^{(d)} = (W^{(e)})^T$**:** $W^{(e)}$ và $W^{(d)}$ được ràng buộc là chuyển vị của nhau.

Cách ràng buộc trọng số của chúng tôi là kết hợp của hai ràng buộc: $W^{(e)}$ và $W^{(d)}$ được chuẩn hóa, và $W^{(d)} = (W^{(e)})^T$. Trong thí nghiệm này, chúng tôi dùng SW-SGD để huấn luyện SRAEs.

Như có thể thấy ở bảng 3.2, trong số các cách ràng buộc trọng số, cách ràng buộc của chúng tôi giúp SRAEs học được những đặc trưng cho kết quả phân lớp (với "Softmax Regression") tốt nhất trên tập kiểm tra. Ngoài ra, bảng 3.2 cũng so sánh thời gian huấn luyện SRAEs trên một vòng lặp (ứng với một lần duyệt qua toàn bộ các mẫu huấn luyện) với các cách ràng buộc trọng số khác nhau này (do chiến lược "dừng sớm", quá trình huấn luyện SRAEs với các cách ràng buộc khác nhau có thể kết thúc sau các số lượng vòng lặp khác nhau; do đó, để chính xác, ta nên so sánh theo thời gian huấn luyện xét trên một vòng lặp hơn là tổng thời gian huấn luyện). Các cách ràng buộc trọng số được sắp xếp theo thứ tự thời gian huấn luyện (trên một vòng lặp) tăng dần là: $W^{(d)} = (W^{(e)})^T$ (2 giây), $W^{(d)}$ được chuẩn hóa (3 giây), cách ràng buộc trọng số của chúng tôi (4 giây), $W^{(e)}$ và $W^{(d)}$ được chuẩn hóa (5 giây). Thứ tự này là hợp lý:

- Ràng buộc $W^{(d)} = (W^{(e)})^T$ có thời gian huấn luyện nhanh nhất vì SRAEs không phải thực hiện bước chuẩn hóa.

- Ràng buộc $W^{(d)}$ được chuẩn hóa có thời gian huấn luyện lâu hơn vì bộ giải mã của SRAEs phải thực hiện bước chuẩn hóa khi lan truyền tiến; và do đó, khi lan truyền ngược, việc tính toán các đạo hàm riêng theo các tham số của bộ giải mã cũng sẽ tốn thời gian hơn bình thường.

- Ở cách ràng buộc trọng số của chúng tôi, khi lan truyền tiến, mặc dù cần phải thực hiện bước chuẩn hóa ở cả bộ mã hóa và bộ giải mã, nhưng nhờ vào ràng buộc $W^{(d)} = (W^{(e)})^T$, ta chỉ cần phải thực hiện bước chuẩn hóa cho bộ trọng số của bộ mã hóa, rồi sau đó dùng lại bộ trọng số đã được chuẩn hóa này cho bộ giải mã. Thời gian huấn luyện của cách ràng buộc này lâu hơn cách ràng buộc $W^{(d)}$ được chuẩn hóa ở trên vì khi lan truyền ngược, ngoài việc tính toán các đạo hàm riêng theo các tham số của bộ giải mã đã được chuẩn hóa, ta cũng cần phải tính toán các đạo hàm riêng theo các tham số của bộ mã hóa đã được chuẩn hóa (khi bộ mã hóa hay bộ giải mã phải thực hiện bước chuẩn hóa khi lan truyền tiến thì việc tính toán các đạo hàm riêng theo các tham số của chúng khi lan truyền ngược sẽ lâu hơn so với khi không thực hiện bước chuẩn hóa).

- Ràng buộc $W^{(e)}$ và $W^{(d)}$ được chuẩn hóa có thời gian huấn luyện lâu nhất vì khi lan truyền tiến, ta phải thực hiện bước chuẩn hóa riêng cho bộ mã hóa và bộ giải mã; và khi lan truyền ngược, ta phải tính toán các đạo hàm riêng theo các tham số của bộ giải mã và bộ mã hóa đã được chuẩn hóa.

Mặc dù thời gian huấn luyện (trên một vòng lặp) của cách ràng buộc trọng số của chúng tôi là khá cao khi so sánh với cách ràng buộc trọng số khác, nhưng nhìn chung nó vẫn nhanh (nhờ vào việc sử dụng GPU để tính toán song song). Tổng thời gian huấn luyện là khoảng 2.5 giờ.

Bảng 3.2: So sánh giữa cách ràng buộc trọng số cho SRAEs của chúng tôi với các cách ràng buộc trọng số khác mà có thể áp dụng cho SRAEs. Cách ràng buộc trọng số của chúng tôi giúp SRAEs học được những đặc trưng mà cho kết quả phân lớp (với "Softmax Regression") tốt nhất trên tập kiểm tra. Ngoài ra, thời gian huấn luyện trên một vòng lặp của SRAEs với các cách ràng buộc trọng số khác nhau cũng được trình bày ở cột cuối cùng của bảng.

Cách ràng buộc trọng số	**Độ lỗi phân lớp trên tập kiểm tra (%)**	**Thời gian huấn luyện của một vòng lặp (giây)**
$W^{(d)}$ được chuẩn hóa	3.28	3
$W^{(e)}$ & $W^{(d)}$ được chuẩn hóa	2.51	5
$W^{(d)} = (W^{(e)})^T$	2.04	2
Cách ràng buộc của chúng tôi	**1.62**	4

SRAEs và các loại "Auto-Encoders" khác

Cuối cùng, chúng tôi cũng so sánh SRAEs (sử dụng cách ràng buộc trọng số của chúng tôi và dùng SW-SGD để huấn luyện) với các loại "Auto-Encoders" khác, bao gồm:

- **"Denoising Auto-Encoders" (DAEs)** [**?**]: DAEs muốn học được các đặc trưng "bền vững" bằng cách làm nhiễu véc-tơ đầu vào rồi sau đó cố gắng tái tạo lại véc-tơ đầu vào ban đầu từ véc-tơ đã bị làm nhiễu này (véc-tơ đầu vào đã bị làm nhiễu $\rightarrow$ véc-tơ đặc trưng $\rightarrow$ cố gắng tái tạo lại véc-tơ đầu vào không bị nhiễu).

- **"Contractive Auto-Encoders" (CAEs)** [**?**]: DAEs muốn học được các đặc trưng thỏa hai tính chất: (i) có thể tái tạo tốt véc-tơ đầu vào ban đầu, và (ii) bất biến đối với sự thay đổi nhỏ của véc-tơ đầu vào (bằng cách phạt chuẩn Frobenius của ma trận Jacobian của véc-tơ đặc trưng đối với véc-tơ đầu vào).

- **"Higher Order Contractive Auto-Encoders" (HCAEs)** [**?**]: HCAEs là mở rộng của CAEs; bên cạnh độ lỗi tái tạo và chuẩn Frobenius của ma trận Jacobian, HCAEs còn phạt thêm chuẩn Frobenius của ma trận Hessian.

Bảng 3.3 so sánh các đặc trưng học được (theo độ lỗi phân lớp trên tập kiểm tra) của SRAEs với các loại "Auto-Encoders" trên. Với DAEs, CAEs, HCAEs, [**?**] dùng 1000 nơ-ron ẩn, hàm kích hoạt sigmoid ở cả tầng ẩn và tầng đầu ra, độ lỗi tái tạo

Bảng 3.3: So sánh giữa SRAEs (sử dụng cách ràng buộc trọng số của chúng tôi và dùng SW-SGD để huấn luyện) với các loại "Auto-Encoders" khác, bao gồm: "Denoising Auto-Encoders" (DAEs), "Contractive Auto-Encoders" (CAEs), "Higher Order Contractive Auto-Encoders" (HCAEs).

Thuật toán học đặc trưng	**Độ lỗi phân lớp trên tập kiểm tra (%)**
DAEs [**?**]	2.05
CAEs [**?**]	1.82
SRAEs	1.62
HCAEs [**?**]	**1.20**

"cross-entropy", và ràng buộc $W^{(e)}$ và $W^{(d)}$ là chuyển vị của nhau. Như có thể thấy, các đặc trưng học được bởi SRAEs cho kết quả phân lớp (với "Softmax Regression") trên tập kiểm tra tốt hơn DAEs và CAEs, nhưng không tốt bằng HCAEs. Tuy nhiên, để ý là HCAEs phức tạp hơn nhiều so với SRAEs của chúng tôi với rất nhiều siêu tham số cần phải lựa chọn.

Chương 4

Kết Luận và Hướng Phát Triển

Kết luận

Trong luận văn này, chúng tôi nghiên cứu về bài toán học đặc trưng không giám sát bằng "Sparse Auto-Encoders" (SAEs). SAEs có thể học được những đặc trưng tương tự như "Sparse Coding", nhưng điểm lợi là quá trình huấn luyện SAEs có thể được thực hiện một cách hiệu quả thông qua thuật toán lan truyền ngược, và với một véc-tơ đầu vào mới, SAEs có thể tính được véc-tơ đặc trưng tương ứng rất nhanh. Tuy nhiên, trong thực tế, không dễ để có thể làm SAEs "hoạt động"; có hai điểm ta cần phải làm rõ: (i) ràng buộc thưa, và (ii) ràng buộc trọng số. Đóng góp của luận văn là làm rõ SAEs ở hai điểm này. Cụ thể như sau:

- Về ràng buộc thưa, mặc dù chuẩn L1 là cách tự nhiên (vì L1 được dùng trong Sparse Coding) và đơn giản để ràng buộc tính thưa của véc-tơ đặc trưng, nhưng L1 lại thường không được dùng trong SAEs với lý do vẫn còn chưa rõ ràng. Thay vì dùng L1, các bài báo về SAEs thường ràng buộc thưa bằng cách ép giá trị đầu ra trung bình của mỗi nơ-ron ẩn về một giá trị cố định gần 0. Nhưng giá trị cố định này lại thêm một siêu tham số vào danh sách các siêu tham số vốn đã có rất nhiều của SAEs; điều này sẽ làm cho quá trình chọn lựa các siêu tham số trở nên "phiền phức" hơn và tốn thời gian hơn. *Trong luận văn, chúng tôi cố gắng hiểu khó khăn gặp phải khi huấn luyện SAEs với chuẩn L1; từ đó, đề xuất một phiên bản hiệu chỉnh của thuật toán "Stochastic Gradient Descent" (SGD), gọi là "Sleep-Wake Stochastic Gradient Descent" (SW-SGD), để khắc phục khó khăn gặp phải này. Ở đây, chúng tôi tập trung nghiên cứu SAEs với*

hàm kích hoạt "rectified linear" ở tầng ẩn vì hàm này tính nhanh và có thể cho tính thưa thật sự (đúng bằng 0); chúng tôi gọi SAEs với hàm kích hoạt này là "Sparse Rectified Auto-Encoders" (SRAEs).

- Về ràng buộc trọng số, có một số cách đã được đề xuất để ràng buộc trọng số của SAEs, nhưng không rõ là tại sao ta lại nên ràng buộc trọng số như vậy. Liệu có cách ràng buộc trọng số nào tốt hơn? *Trong luận văn, chúng tôi đề xuất một cách ràng buộc trọng số mới và hợp lý cho SRAEs.*

Các kết quả thí nghiệm trên bộ dữ liệu MNIST (bộ ảnh chữ số viết tay từ 0 đến 9) cho thấy:

- Khi huấn luyện SRAEs với chuẩn L1 sẽ gặp phải vấn đề nơ-ron "ngủ" và chiến lược "ngủ - đánh thức" đề xuất của chúng tôi trong thuật toán SW-SGD có thể giúp khắc phục vấn đề này.

- Cách ràng buộc trọng số đề xuất của chúng tôi giúp SRAEs học được những đặc trưng cho kết quả phân lớp tốt nhất so với các cách ràng buộc trọng số khác mà có thể áp dụng cho SRAEs.

- SRAEs với SW-SGD và cách ràng buộc trọng số của chúng tôi có thể học được những đặc trưng cho kết quả phân lớp tốt so với các loại "Auto-Encoders" khác.

Hướng phát triển

Thật ra, luận văn mới chỉ giải quyết được một phần nhỏ và mang tính kỹ thuật (làm cho SAEs hoạt động) của bài toán học đặc trưng không giám sát. Câu hỏi lớn và mang tính định hướng dài hạn là: *Thế nào là một biểu diễn đặc trưng tốt?* Theo GS. Yoshua Bengio, một trong những nhà nghiên cứu tiên phong trong lĩnh vực học biểu diễn đặc trưng, thì: *Một biểu diễn đặc trưng tốt cần* ***phân tách (disentangle)*** *được các yếu tố giải thích ẩn bên dưới*. Để phân tách được các yếu tố giải thích ẩn, ta cần có sự hiểu biết trước (prior) về các yếu tố ẩn. Ở đây, ta quan tâm đến các sự hiểu biết trước mang tính tổng quát, có thể áp dụng để học đặc trưng trong nhiều bài toán liên quan đến trí tuệ nhân tạo (thị giác máy tính, xử lý ngôn ngữ tự nhiên, ...). Định hướng phát triển

của luận văn là tích hợp thêm các hiểu biết trước khác vào SAEs nhằm phân tách tốt hơn các yếu tố giải thích ẩn. Dưới đây là một số hiểu biết trước mà có thể tích hợp vào SAEs:

- **Học sâu**: thế giới xung quanh ta có thể được mô tả bằng một kiến trúc phân cấp; cụ thể là, các yếu tố hay các khái niệm (concept) trừu tượng (ví dụ như con mèo, cái cây, ...) bao gồm các khái niệm ít trừu tượng hơn; các khái niệm ít trừu tượng hơn này lại bao gồm các khái niệm ít trừu tượng hơn nữa ... Do đó, ta muốn học nhiều tầng biểu diễn đặc trưng với độ trừu tượng tăng dần. Mặc dù, SRAEs có thể được dùng để học từng tầng đặc trưng một, nhưng mục tiêu mà chúng tôi hướng đến là: học *đồng thời* nhiều tầng biểu diễn đặc trưng một cách không giám sát.

- **Gom cụm tự nhiên**: các mẫu thuộc các lớp khác nhau nằm trên các đa tạp (manifold) khác nhau và các đa tạp này được phân tách tốt với nhau bởi các vùng có mật độ thấp; hơn nữa, số chiều của các đa tạp này nhỏ hơn rất nhiều so với số chiều của không gian ban đầu. Ta thấy rằng sự gom cụm tự nhiên này sẽ dẫn đến tính thưa. Cụ thể là, các đa tạp khác nhau (ứng với các lớp khác nhau) sẽ được mô tả bởi các hệ trục tọa độ khác nhau. Với một véc-tơ đầu vào x thì chỉ có hệ trục tọa độ của đa tạp ứng với lớp mà x thuộc về được kích hoạt. Nếu ta hiểu véc-tơ đặc trưng h của x chứa các hệ số của các hệ trục tọa độ này thì h sẽ thưa bởi vì chỉ có các hệ số của hệ trục tọa độ được kích hoạt là có giá trị khác 0. Do đó, thay vì ràng buộc tính thưa một cách đơn thuần bằng chuẩn L1, ta có thể tìm cách để ràng buộc tính thưa từ góc nhìn gom cụm tự nhiên nói trên.

Phụ Lục: Các Công Trình Đã Công Bố

Hội nghị quốc tế:

- **K. Tran** and B. Le, "Demystifying Sparse Rectified Auto-Encoders," in *Proceedings of the Fourth Symposium on Information and Communication Technology*, ser. SoICT'13. New York, NY, USA: ACM, 2013, pp. 101–107. [Online]. Available: http://doi.acm.org/10.1145/2542050.2542065

PROCEEDINGS OF

THE FOURTH SYMPOSIUM ON INFORMATION AND COMMUNICATION TECHNOLOGY

SoICT 2013

Da Nang, Vietnam
December 5-6, 2013

ISBN: 978-1-4503-2454-0

Symposium on Information and Communication Technology 2013

SoICT 2013

Table of Contents

Demystifying Sparse Rectified Auto-Encoders

Kien Tran
Department of Computer Science
Faculty of Information Technology
Vietnam University of Science - HCM
ttkien@fit.hcmus.edu.vn

Bac Le
Department of Computer Science
Faculty of Information Technology
Vietnam University of Science - HCM
lhbac@fit.hcmus.edu.vn

ABSTRACT

Sparse Auto-Encoders can learn features similar to Sparse Coding, but the training can be done efficiently via the back-propagation algorithm as well as the features can be computed quickly for a new input. However, in practice, it is not easy to get Sparse Auto-Encoders working; there are two things that need investigating: sparsity constraint and weight constraint. In this paper, we try to understand the problem of training Sparse Auto-Encoders with L1-norm sparsity penalty, and propose a modified version of Stochastic Gradient Descent algorithm, called Sleep-Wake Stochastic Gradient Descent (SW-SGD), to solve this problem. Here, we focus on Sparse Auto-Encoders with rectified linear units in the hidden layer, called Sparse Rectified Auto-Encoders (SRAEs), because such units compute fast and can produce true sparsity (exact zeros). In addition, we propose a new reasonable way to constrain SRAEs' weights. Experiments on MNIST dataset show that the proposed weight constraint and SW-SGD help SRAEs successfully learn meaningful features that give excellent performance on classification task compared to other Auto-Encoder variants.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning—*connectionism and neural nets, concept learning, parameter learning*; I.2.10 [**Artificial Intelligence**]: Vision and Scene Understanding—*representation, data structures, and transforms*; I.4.7 [**Image Processing and Computer Vision**]: Feature Measurement—*feature representation*

General Terms

Algorithms, Design, Experimentation

Keywords

unsupervised feature learning, deep learning, sparse coding, sparse auto-encoders, rectified linear units

SoICT'13, December 05 - 06 2013, Danang, Viet Nam

http://dx.doi.org/10.1145/2542050.2542065.

1. INTRODUCTION

Recently, unsupervised feature learning and deep learning have attracted a lot of interest from various fields such as computer vision, audio processing, text processing, and so on. The idea is that instead of designing features manually, one lets the learning algorithms automatically learn features from unlabeled data; and deep learning means learning multiple levels of features with increasing abstraction. Auto-Encoders (AEs) and Restricted Boltzmann Machines (RBMs) are two main groups of algorithms that have been used in unsupervised feature learning and deep learning [1]. AEs belong to the non-probabilistic group while RBMs belong to the probabilistic group. One big disadvantage of RBMs compared to AEs is that the objective function of RBMs is intractable. For this reason, here we will focus on the study of AEs.

Several criteria have been proposed to guide AEs to learn useful representation. They include: sparsity criterion [6], denoising criterion [14], and contraction criterion [13, 12]. Among them, sparsity is an interesting and promising one (here sparsity means forcing the majority of elements of the feature vector to be zeros). The first reason is that it has the inspiration from biology. In the brain, there is a very small fraction of neurons active simultaneously. Sparsity was first introduced in Sparse Coding and interestingly, it helped learn features similar to the primary visual cortex [11]. AEs with sparsity criterion, called Sparse Auto-Encoders (SAEs), can learn features much like Sparse Coding, but unlike Sparse Coding, the training can be done efficiently via the back-propagation algorithm, and with a new input, the features can be computed quickly. Secondly, sparsity can help learn high-level features - concepts. The intuitive justification is that there are only a few concepts per example; therefore, sparsity can help learn a dictionary of concepts and each example will be explained just by a small number of concepts. Thirdly, sparsity can potentially help speed up the training of SAEs. With each example, in the forward propagation phase, there is only a small fraction of neurons active; and hence, in the backward propagation phase, there is only a small fraction of parameters (corresponding to active neurons) updated. This point can be made use of to speed up the training process. It is important because if the training is fast, the model can be scaled up (i.e. increase the number of features); in unsupervised feature learning and deep learning, large-scale is a key factor to get good performance [4, 7].

Despite above advantages, it is not easy to get SAEs working in practice. To make SAEs work, there are two things

that need investigating: sparsity constraint and weight constraint. Although L1-norm is a natural (because it is used in Sparse Coding) and simple (in case the feature vector has positive values, it is just simply sum of them) way to constrain sparsity, it is not often used in SAEs for reasons that remain to be understood [1]. Instead of L1-norm, people often constrain sparsity in SAEs by pushing the average output of a hidden neuron (e.g. over a minibatch) to a fixed target (close to zero) [6, 4, 3]. But this fixed target adds one more hyper-parameter to the list of SAEs' hyper-parameters which already has many ones. As a result, the process of tuning hyper-parameters will become more tedious and more time-consuming. Regarding weight constraint, many different ways were used in the literature. [3, 14, 13, 12] tied the weights of encoder and decoder together. [6, 4] used weight decay; this way even adds one more hyper-parameter. [15] constrained the weights of decoder to have unit norm. However, it is not clear which way should be used as well as why weights should be constrained like those.

Two questions remain to be answered: (i) why is L1-norm sparsity penalty not often used in SAEs?; (ii) is there a better and more reasonable way to constrain SAEs' weights? In this paper, we try to understand the problem of training SAEs with L1-norm sparsity penalty. Then, we propose a modified version of Stochastic Gradient Descent algorithm (SGD), called Sleep-Wake Stochastic Gradient Descent (SW-SGD), to remedy this problem. Here we focus on SAEs with rectified linear units (ReLUs) in the hidden layer because such units compute fast and can produce true sparsity (exact zeros) [10, 5, 15]. We call these Sparse Rectified Auto-Encoders (SRAEs). Furthermore, we propose a new reasonable way to constrain SRAEs' weights. With these two ingredients, our proposed weight constraint and SW-SGD, our experiments show that SRAEs can successfully learn meaningful features that give excellent classification performance on MNIST dataset compared to other Auto-Encoder variants.

The rest of the paper is organized as follows. We start by reviewing Sparse Coding and Sparse Auto-Encoders (SAEs) to see advantages of SAEs compared to Sparse Coding. Then, Section 3 presents Sparse Rectified Auto-Encoders (SRAEs): Subsection 3.1 explains the problem of training SRAEs with L1-norm sparsity penalty and describes our remedy for this problem; Subsection 3.2 presents our proposed weight constraint for SRAEs. Experiment and analysis are shown in Section 4 followed by the conclusion in Section 5.

2. REVIEW OF SPARSE CODING AND SPARSE AUTO-ENCODERS

2.1 Sparse Coding

Sparse Coding was first introduced in neuroscience to model the primary visual cortex [11]. The goal is to find an over-complete set of basic vectors so that each input can be explained just by a small number of basis vectors (i.e. the feature vector is sparse). Specifically, given the unlabeled data $\{x^{(1)}, \dots, x^{(N)}\}$ with $x^{(n)} \in \mathbb{R}^D$, Sparse Coding solves the following optimization problem:

$$\begin{aligned} \underset{\phi,a}{\text{minimize}} \quad & \sum_{n=1}^{N} \left(\|x^{(n)} - \sum_{k=1}^{K} a_k^{(n)} \phi^{(k)}\|_2^2 + \lambda \|a^{(n)}\|_1 \right) \\ \text{subject to} \quad & \|\phi^{(k)}\|_2^2 = 1, \forall k = 1, \dots, K \end{aligned} \quad (1)$$

Here, the optimization variables are *the basis vectors* $\phi = \{\phi^{(1)}, \dots, \phi^{(K)}\}$ with each $\phi^{(k)} \in \mathbb{R}^D$, and *the coefficient vectors* (the feature vectors) $a = \{a^{(1)}, \dots, a^{(N)}\}$ with each $a^{(n)} \in \mathbb{R}^K$; $a_k^{(n)}$ is the coefficient of basic $\phi^{(k)}$ for input $x^{(n)}$. With this optimization problem, we want to learn a representation having the following properties:

- Preserving information about the input (by minimizing the reconstruction error).
- Being sparse (by minimizing the L1-norm of the feature vector).

λ is the hyper-parameter controlling the trade-off between reconstruction error and sparsity penalty.

The problem (1) can be solved by iteratively optimizing over a and ϕ alternately while holding the other set of variables fixed [9]. However, this process often takes a long time to converge. Furthermore, after training, to find the feature vector for a new input, we still have to do optimization (with fixed ϕ).

2.2 Sparse Auto-Encoders

An Auto-Encoder (AE) is a feed-forward neural network with two layers. The first layer, called *encoder*, maps the input x to the hidden representation a: $a = f\left(W^{(e)}x + b^{(e)}\right)$ where $f(\cdot)$ is some activation function (e.g. sigmoid), $W^{(e)}$ and $b^{(e)}$ are parameters of the encoder. The second layer, called *decoder*, then tries to reconstruct the input from the hidden representation a: $\hat{x} = W^{(d)}a + b^{(d)}$ where $\hat{x}$ is the reconstructed input, $W^{(d)}$ and $b^{(d)}$ are parameters of the decoder. In this way, we hope that the hidden representation can capture the structure of the input.

In Sparse Auto-Encoders (SAEs), besides reconstruction error, we also constrain the representation to be sparse (i.e. with a input, there are only a few hidden neurons active). Specifically, given the unlabeled data $\{x^{(1)}, \dots, x^{(N)}\}$ with $x^{(n)} \in \mathbb{R}^D$, SAEs minimize the following objective function:

$$J(W^{(e)}, b^{(e)}, W^{(d)}, b^{(d)}) = \sum_{n=1}^{N} \|x^{(n)} - \hat{x}^{(n)}\|_2^2 + \lambda s(a^{(n)}) \quad (2)$$

where: $a^{(n)} = f\left(W^{(e)}x^{(n)} + b^{(e)}\right)$; $\hat{x}^{(n)} = W^{(d)}a^{(n)} + b^{(d)}$; $s(\cdot)$ is some function that encourages the feature vector $a^{(n)}$ to be sparse; and λ is the hyper-parameter controlling the trade-off between reconstruction error and sparsity penalty.

Similar to Sparse Coding, SAEs aim at learning a representation that both preserves information about the input and is sparse. The difference between them is that SAEs have an explicit parametric encoder, while Sparse Coding has an implicit non-parametric encoder. This point helps training SAEs be more efficient than Sparse Coding; it can be done via the back-propagation algorithm. In addition, with a new input, SAEs can compute the corresponding feature vector very quickly just by one step.

3. SPARSE RECTIFIED AUTO-ENCODERS

The typical activation functions have been used in neural networks are the sigmoid function and the tanh function. Recently, a new activation function which have been found to work very well is the rectified linear function [10, 5, 15]:

$f(x) = \max(0, x)$. Units with such activation function are called rectified linear units (ReLUs).

ReLUs fit well with SAEs because such units naturally produce a sparse feature vector. Unlike logistic units that give small positive values when the input is not aligned with the filters (the incoming weight vectors of hidden units), ReLUs often give exact zeros. Furthermore, ReLUs compute faster than logistic or tanh units because they do not involve exponentiation and division; they just have to compute the max operation. Finally, ReLUs can potentially help jointly train multi-layers of features (instead of training layer by layer in greedy fashion) because ReLUs have been used to train supervised deep networks successfully [5, 15]. Therefore, here we will focus on SAEs with ReLUs (in the hidden layer). We call them Sparse Rectified Auto-Encoders (SRAEs).

3.1 Sparsity Constraint in SRAEs

The typical way that have been used to constrain sparsity in Sparse Auto-Encoders (SAEs) is pushing the average output $\bar{a}_j$ of hidden neuron j (over a minibatch) to some fixed target ρ (a value close to zero) [6, 4, 3]. In case the hidden neuron's output $\in [0, 1]$ (e.g. sigmoid unit), this can be done through the Kullback-Leibler (KL) divergence: $\sum_j \mathrm{KL}(\rho \| \bar{a}_j) = \sum_j \rho \log \frac{\rho}{\bar{a}_j} + (1-\rho) \log \frac{(1-\rho)}{(1-\bar{a}_j)}$. In case using ReLUs, the squared error can be used: $\sum_j (\bar{a}_j - \rho)^2$. Note that this way does not directly encourage the feature vector (corresponding to a example) to be sparse, but encourage the values of a feature (the outputs of a hidden neuron) over examples to be sparse. It, however, indirectly leads to a sparse feature vector because the reconstruction error tends to make learned features differ from each other; therefore, with an example, if some feature is active (having a non-zero value), the majority of the rest will be inactive (having a zero value).

This way, however, adds one more hyper-parameter (the fixed target ρ) to the list of SAEs' hyper-parameters which already has many ones (the trade-off parameter λ, the number of features, learning rate, minibatch size, and so on). As a result, the process of tuning hyper-parameters will become more annoying and more time-consuming. Why do not use L1-norm? It is natural because L1-norm is used in Sparse Coding. In addition, it doesn't have any extra hyper-parameter. It is also very simple; in case using ReLUs, it is just the sum of elements of the feature vector a. In the following section, we will explain the problem of training SAEs, in particular SRAEs, with L1-norm.

3.1.1 The Difficulty of Training SRAEs with L1-norm

The problem of training SAEs with L1-norm is that during the optimization process, L1-norm can drive the incoming weight vector of a hidden neurons to the state in which the hidden neuron is always inactive (produce zero with all examples in the dataset). And once the incoming weight vector has been in such a state, it will be stuck there forever and never get updated; the outgoing weight vector of this hidden neuron will also never get updated. Formally, let's consider a hidden neuron j which have a weight $W^{(e)}_{ji}$ connecting to an input neuron i and a weight $W^{(d)}_{kj}$ connecting to an output neuron k. The gradients of the objective function J in equation (2) (with the sparsity function $s(\cdot) = \|\cdot\|_1$) with respect to $W^{(e)}_{ji}$ and $W^{(d)}_{kj}$ are:

$$\frac{\partial J}{\partial W^{(d)}_{kj}} = \sum_{n=1}^{N} 2(\hat{x}^{(n)}_k - x^{(n)}_k) a^{(n)}_j \tag{3}$$

$$\frac{\partial J}{\partial W^{(e)}_{ji}} = \sum_{n=1}^{N} (\epsilon^{(n)}_j + \lambda) f'(a^{(n)}_j) x^{(n)}_i \tag{4}$$

where:

- $x^{(n)}_k$ and $\hat{x}^{(n)}_k$ are respectively the k^{th} element of the input vector $x^{(n)}$ and the reconstructed input vector $\hat{x}^{(n)}$.
- $a^{(n)}_j$ is the j^{th} element of the feature vector $a^{(n)}$.
- $\epsilon^{(n)}_j$ is the "error" that the hidden neuron j receives from the output layer (corresponding to the input $x^{(n)}$).

From equations (3) and (4), one can easily see that, during the optimization, if once the hidden neuron j has been in the state having a_j equal zero with all examples, the gradients $\frac{\partial J}{\partial W^{(d)}_{kj}}$ and $\frac{\partial J}{\partial W^{(e)}_{ji}}$ will be zeros with all examples (in case $f(\cdot)$ is the rectified linear function, the derivative $f'(0)$ equals 0) and the weights of this neuron will never get updated anymore. We call such neurons *"sleep" neurons*. Especially, the "easy to get exact zeros" property of ReLUs can make this problem easier to happen during the optimization.

The above problem may explain why people often don't use L1-norm in SAEs but instead, push the average output of a hidden neuron to a fixed target close to zero (but not zero!); this way may prevent the hidden neuron from the situation in which it is inactive for all examples and then never get updated. With sigmoid units, the KL divergence can be used and the average output cannot be zero because if so, the KL divergence will give an infinite penalty. With ReLUs, the KL divergence cannot be used because the outputs of ReLUs are not in $[0, 1]$. The squared error can be used instead but we found experimentally that the "sleep" neuron problem still happens. It is because with a zero average output, unlike the KL divergence, the squared error still gives a very small penalty. See Figure 1 for a comparison of them with the fixed target ρ of 0.1.

Although using L1-norm, Sparse Coding clearly doesn't have this problem because the encoder of Sparse Coding is implicit.

3.1.2 Sleep-Wake Stochastic Gradient Descent

To remedy the problem of training SRAEs with L1-norm, we propose a modified version of Stochastic Gradient Descent algorithm (SGD), called Sleep-Wake Stochastic Gradient Descent (SW-SGD). The idea is that during each epoch of SGD, we track the average outputs of hidden neurons. Then, after each epoch, we check if there are any "sleep" neurons (having the average output equal zero), and we will "wake-up" them by simply re-initializing their incoming weight vectors (including the biases). Despite its simplicity, our experiments showed that this strategy can help SRAEs successfully learn meaningful features without any "sleep" features.

3.2 Weight Constraint in SRAEs

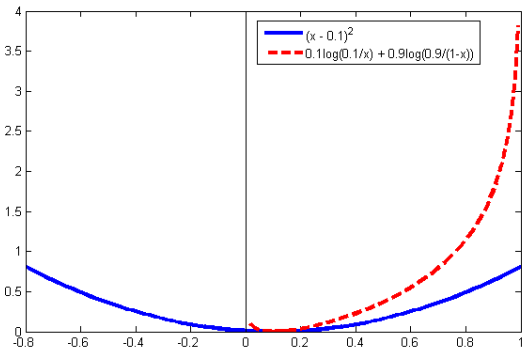


Figure 1: Comparison of KL divergence to squared error with the fixed target ρ of 0.1. When the average output of a hidden neuron is zero, KL divergence gives an infinite penalty while squared error still gives a very small penalty.

Besides sparsity constraint, weight constraint is also a key ingredient to get SAEs working. There are several ways have been used to constrain SAEs' weights:

- **Tied weights**: the weights of encoder and decoder are tied together ($W^{(d)} = (W^{(e)})^T$) [3]. This way was also used in other Auto-Encoder variants such as Denoising Auto-Encoders and Contractive Auto-Encoders [14, 13, 12]. Note that all [3, 14, 13, 12] used sigmoid units in the hidden layer. There is a trivial descent direction of SAEs' objective function in which the hidden neuron's output a_j is scaled down (by scaling down the incoming weight vector of this hidden neuron) and the outgoing weight vector of this hidden neuron is scaled up by some large constant; as a result, the sparsity penalty can decrease arbitrary while the reconstruction error is unchanged. Tied weights can help prevent from this trivial direction, but it is not clear what is going on when the encoder's weights and the decoder's weights are tied together, especially in case using sigmoid units.

- $W^{(d)}$ **norm constraint**: [15] constrained the basis vectors of the decoder (the outgoing weight vectors of hidden neurons) to have unit norm. This constraint is similar to Sparse Coding and also helps prevent from the scale problem. But how about the encoder's weights? For example, to be fair between features, the incoming weight vectors of hidden neurons should have the same norm.

- **Weight decay**: weights of the encoder and decoder are kept small by penalizing the sum of squares of them [6, 4]. As two previous ways, this way prevents SAEs from the scale problem too. It can be interpreted as a "soft" way to constrain the norms of the incoming weights vector of hidden units to be approximately equal to each other and the norms of the outgoing weight vectors of hidden units to be approximately equal to each other. However, this way introduces one more hyper-parameter; it's annoying.

3.2.1 *Our Proposed Weight Constraint for SRAEs*

In this section, we propose a reasonable way to constrain SRAEs' weights. It also doesn't introduce any extra hyper-parameter. Concretely, our way consists of two constraints:

- First, we tie the encoder's weights and the decoder's weights together: $W^{(d)} = \left(W^{(e)}\right)^T$

- Second, we also constrain the incoming weight vectors as well as the outgoing weight vectors of hidden units to have unit norm.

With an example x, if one just pays attention to non-zero rectified linear units, the whole system is a linear system. Therefore, with two above constraints, the encoder will project linearly the input vector x onto a few normalized basis vectors (in the whole set of normalized basis vectors) corresponding to non-zero hidden units; and then, the decoder will reconstruct the input vector from these basis vectors: $\hat{x} = W^T W x$ where x is a column vector and rows of W corresponds to normalized basis vectors selected by ReLUs (here, we just ignore the biases for simplicity). In other words, with above constraints, SRAEs will learn a set of normalized basis vectors such that different inputs can be explained by different small subsets of basis vectors (by projecting linearly the input onto the subset of basis vectors selected by ReLUs and then reconstructing the input from this subset).

The second constraint, however, cannot be enforced by gradient-based methods. To overcome this problem, we change the forward propagation formula of SRAEs as follows:

$$\hat{x} = (\hat{W}^{(e)})^T \max(0, \hat{W}^{(e)} x + b^{(e)}) + b^{(d)} \qquad (5)$$

where $\hat{W}^{(e)}$ is a row-normalized matrix of $W^{(e)}$ (each row of $W^{(e)}$ corresponds to an unnormalized basis vector). Here, the learned parameters are still $W^{(e)}$, $b^{(e)}$, and $b^{(d)}$. In this way, gradient-based methods can be used as usual.

Finally, the first constraint, tied weights, also helps save about haft of memory compared to untied weights. It will be beneficial when using GPU (for parallel computing).

4. EXPERIMENTS

4.1 Setup

We experimented on the MNIST dataset which composes of grayscale images (28×28 pixels) of 10 hand-written digits (from 0 to 9) [8]. Figure 2 shows some examples of this dataset. The images were preprocessed by scaling to $[0, 1]$. We used the usual split: $50,000$ examples for training, $10,000$ examples for validation, and $10,000$ examples for test.

We conducted all experiments using the Python Theano library [2], which allows for quick development and easy use of GPU (for parallel computing). We used a single NVIDIA GTX 560 GPU.

After the unsupervised feature learning phase, we evaluated the learned features by feeding them to a softmax regression and measuring the classification error. Concretely, given the training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$ where $x^{(i)} \in \mathbb{R}^D$ is the image vector and $y^{(i)} \in \{0, \ldots, 9\}$ is the class label, we fed $x^{(i)}$ to the trained Auto-Encoder (the Auto-Encoder was trained on the unlabeled data $\{x^{(1)}, \ldots, x^{(N)}\}$)

Figure 2: Some examples of MNIST dataset

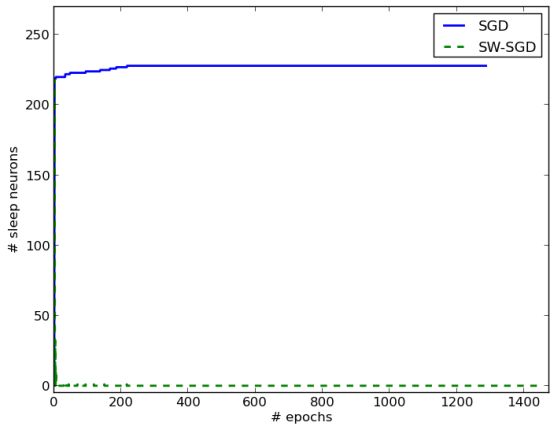


Figure 3: The number of "sleep" hidden neurons of SRAEs during the optimization process with SGD and SW-SGD. The optimization of SGD ended up with 228/1000 "sleep" neurons while SW-SGD ended up without any "sleep" neurons. (These two optimizations terminated after different number of epochs because of the early stopping strategy.)

to get the corresponding feature vector $f^{(i)}$; by this way, we got the new training set $\{(f^{(1)}, y^{(1)}), \ldots, (f^{(N)}, y^{(N)})\}$. Then, we used this new training set to train a softmax regression. With a test example x, we first used the trained Auto-Encoder to compute the feature vector f; then, we fed f to the trained softmax regression to get the class prediction.

In both unsupervised and supervised phase, we used Stochastic Gradient Descent as the optimization algorithm with mini-batch size 100 and early stopping (in the unsupervised phase, we stopped the optimization based on the objective value on the validation set; in the supervised phase, we based on the classification error on the validation set). In all experiments, we used SRAEs with 1000 hidden units, a trade-off parameter λ of 0.25, an unsupervised learning rate of 0.05, and a supervised learning rate of 1.

4.2 SGD versus SW-SGD

To see the problem of training SRAEs with L1-norm sparsity penalty and the effect of our "sleep-wake" strategy, we compared training SRAEs with ordinary Stochastic Gradient Descent (SGD) and our modified version, Sleep-Wake Stochastic Gradient Descent (SW-SGD). In this experiment, we used our proposed weight constraint (tied weights + $W^{(e)}$ norm constraint + $W^{(d)}$ norm constraint).

Figure 3 shows the number of "sleep" hidden neurons of SRAEs during the optimization process with SGD and with SW-SGD. The problem of training SRAEs with L1-norm sparsity penalty is that during the optimization, L1 penalty can push the incoming weight vectors of hidden neurons to "sleep" states (meaning that the corresponding hidden neurons always give zero outputs with all examples in the dataset) and then, they will never get updated anymore; as can be seen from the figure, with ordinary SGD, the number of "sleep" neurons increased during the optimization, especially during the first epochs when the optimization had not stable yet. The SGD optimization finally ended up with 228/1000 "sleep" neurons. This problem of L1 penalty can be remedied by our simple "sleep-wake" strategy; the SW-SGD optimization ended up without any "sleep" neurons.

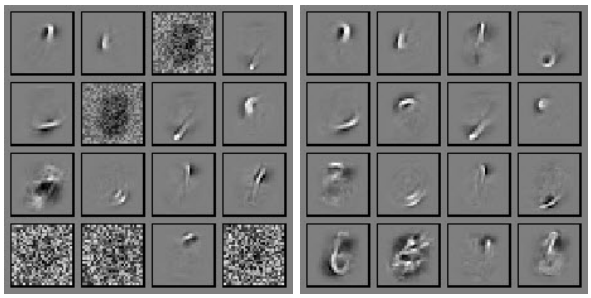
Figure 4 visualizes some example filters (the incoming weight vectors of hidden neurons) learned by SGD and SW-SGD. With SGD, there are five "sleep" filters; they look meaningless. With SW-SGD, there are not any "sleep" filters; all of them look meaningful, like "pen stroke" detectors.

Making use of all filters, SW-SGD achieved better training unsupervised objective value and better test classification performance (with softmax regression) than SGD (Table 1).

4.3 Our Proposed Weight Constraint versus Other Weight Constraints

In this second experiment, we compared our proposed weight constraint for SRAEs to other weight constraints that are possible to be applied to SRAEs. Concretely, we considered the following weight constraints:

- $W^{(d)}$ **norm constraint**: the outgoing weight vectors of hidden units (the columns of $W^{(d)}$) are constrained to have unit norm.



(a) SGD (b) SW-SGD

Figure 4: Figure (a) shows example filters learned by SGD; one can recognize there are five "sleep" filters looking meaningless. Figure (b) shows example filters learned by SW-SGD; all filters look meaningful, like "pen stroke" detectors.

Table 1: Unsupervised objective value on the training set and classification error (with softmax regression) on the test set when training SRAEs with SGD and with SW-SGD

	SGD	SW-SGD
Train Unsupervised Objective Value	9.84	**9.48**
Test Classification Error (%)	1.70	**1.62**

- $W^{(e)}$ **&** $W^{(d)}$ **norm constraint**: both the incoming and outgoing weight vectors of hidden units (the rows of $W^{(e)}$ and the columns of $W^{(d)}$ respectively) are constrained to have unit norm.
- **Tied weights**: the encoder's weights and the decoder's weights are tied together ($W^{(d)} = (W^{(e)})^T$).

Our weight constraint combines both $W^{(e)}$ **&** $W^{(d)}$ **norm constraint** and **tied weights**. In this experiment, we used SW-SGD to train SRAEs. As can be seen from Table 2, our weight constraint gave the best test classification performance (with softmax regression). In the last column, we also show the (approximate) training time per epoch of SRAEs with these different weight constraints (because of the early stopping strategy, the training processes of SAREs with different weight constraints can terminate after different number of epochs; therefore, it will be more accurate to compare them in term of the training time per epoch rather than the total training time). Weight constraints sorted from lowest to highest training time per epoch are: tied weights (2 seconds), $W^{(d)}$ norm constraint (3 seconds), our weight constraint (4 seconds), and $W^{(e)}$ & $W^{(d)}$ norm constraint (5 seconds). This order is reasonable because:

- In tied weights, SRAE doesn't have to do normalization in the forward propagation phase.
- In $W^{(d)}$ norm constraint, SRAE's decoder has to do normalization in the forward propagation phase; and because of this, in the back-propagation phase, the computation of derivatives with respect to the decoder's parameters will also become more expensive than usual.
- In our weight constraint, although we have to do normalization in both the encoder and decoder, we just have to compute the encoder's normalized weights and use them for the decoder thanks to the tied weights constraint. Its epoch time is higher than $W^{(d)}$ norm constraint above because in the back-propagation phase, the computation of derivatives with respect to both the encoder's parameters and the decoder's parameters is more expensive than usual.
- In $W^{(e)}$ & $W^{(d)}$ norm constraint, the training time per epoch is highest because SRAE has to do normalization in the encoder and decoder separately and the computation of derivatives with respect to both the encoder's parameters and the decoder's parameters is more expensive than usual.

Although the training time per epoch of our weight constraint is pretty high compared to other weigh constraints, it's still fast (thanks to the use of GPU). Its total training time is roughly 2.5 hours.

Table 2: Comparison of our weight constraint to other possible weight constraints. Our weight constraint gave the best classification performance (with softmax regression) on the test set. The last column shows the training time per epoch (roughly) of SRAEs with these different weight constraints.

Weight Constraint	Test Error (%)	Epoch Time (sec)
$W^{(d)}$ norm constraint	3.28	3
$W^{(e)}$ & $W^{(d)}$ norm constraint	2.51	5
Tied weights	2.04	2
Our weight constraint	**1.62**	4

Table 3: Comparison of SRAEs (with our weight constraint and SW-SGD) to other Auto-Encoder variants, including: Denoising Auto-Encoders (DAEs), Contractive Auto-Encoders (CAEs), and Higher Order Contractive Auto-Encoders (HCAEs), in term of classification error (with softmax regression) on the test set

Feature Learning Algorithm	Test Error (%)
DAEs [12]	2.05
CAEs [12]	1.82
SRAEs	1.62
HCAEs [12]	**1.20**

4.4 SRAEs versus Other Auto-Encoder Variants

Finally, we also compared SRAEs (with our weight constraint and SW-SGD) to other Auto-Encoder variants, including:

- **Denoising Auto-Encoders (DAEs)** [14]: want to learn robust features by making the input corrupted and trying to reconstruct the "clean" input from this corrupted version.
- **Contractive Auto-Encoders (CAEs)** [13]: want to learn features robust to small changes of the input by besides the reconstruction error, penalizing the Frobenius norm of the Jacobian of the feature vector with respect to the input vector.
- **Higher Order Auto-Encoders (HCAEs) [12]**: are the extension of CAEs; besides the reconstruction error and the Jacobian norm, HCAEs also penalize the approximated Hessian norm.

Table 3 compares the test classification performance of SRAEs to these Auto-Encoder variants. Note that with DAEs, CAEs, and HCAEs, [12] used 1000 hidden units, the sigmoid activation function in the hidden and output layer, the cross-entropy reconstruction error, and tied weights. Our SRAEs were better in term of test classification performance than DAEs and CAEs but worse than HCAEs. However, HCAEs are more complicated than our SRAEs with many hyper-parameters which need to be tuned.

5. CONCLUSION

In this paper, we have investigated SRAEs and in particular, two key ingredients to get SRAEs working: spar-

sity constraint and weight constraint. We have tried to understand the optimization problem when training SRAEs with L1-norm sparsity penalty and proposed a simple modified version of SGD, called SW-SGD, to remedy this problem. We have also proposed a reasonable weight constraint for SRAEs. Our experiments on the MNIST dataset have shown that our weight constraint and SW-SGD work well with SRAEs and can help SRAEs learn meaningful features that give excellent classification performance compared to other Auto-Encoder variants.

Our future work will include:

- Making use of sparsity to speed up the training.
- Unsupervised deep learning: SRAEs can be used to learn multiple layers of representation in greedy fashion but the interesting question is how to jointly learn multiple layers of representation?

6. REFERENCES

[1] Y. Bengio, A. C. Courville, and P. Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[2] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[3] A. Coates. *Demystifying Unsupervised Feature Learning*. PhD thesis, Stanford University, 2012.

[4] A. Coates, A. Y. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. In *International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011.

[5] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, volume 15, pages 315–323, 2011.

[6] I. Goodfellow, H. Lee, Q. V. Le, A. Saxe, and A. Y. Ng. Measuring invariances in deep networks. In *Advances in neural information processing systems*, pages 646–654, 2009.

[7] Q. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. Corrado, J. Dean, and A. Ng. Building high-level features using large scale unsupervised learning. In J. Langford and J. Pineau, editors, *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, ICML '12, pages 81–88, New York, NY, USA, July 2012. Omnipress.

[8] Y. LeCun. The MNIST database. `http://yann.lecun.com/exdb/mnist/`.

[9] H. Lee, A. Battle, R. Raina, and A. Ng. Efficient sparse coding algorithms. In *Advances in neural information processing systems*, pages 801–808, 2006.

[10] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[11] B. A. Olshausen et al. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

[12] S. Rifai, G. Mesnil, P. Vincent, X. Muller, Y. Bengio, Y. Dauphin, and X. Glorot. Higher order contractive auto-encoder. *Machine Learning and Knowledge Discovery in Databases*, pages 645–660, 2011.

[13] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 833–840, 2011.

[14] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[15] M. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, et al. On rectified linear units for speech processing. ICASSP, 2013.