

# Natural Language Processing with PyTorch

## PyTorch를 활용한 자연어처리

이 책은 딥러닝을 사용하여 자연어처리(Natural Language Processing)를 실무에 적용하고자 하는 입문자들을 대상으로 쓰여진 책입니다. 실무에 바로 투입 가능한 정도의 PyTorch 실습 코드를 활용하여 독자들의 이해를 돋고자 하였습니다. 시중에 널리 읽히고 있는 책들과 달리 딥러닝 자체에 대한 설명에 집중하기 보단, 자연어처리(특히 자연어생성, Natural Language Generation)에 더 집중하였습니다. 사실 인터넷이나 시중에는 자연어처리를 특정하여 다루는 자료는 다른 인공지능 분야(컴퓨터 비전 등)에 비하여 적기 때문에, 자연어처리를 공부하고자 하는 이들은 어려움을 겪을 수 밖에 없었습니다. 따라서, 딥러닝을 활용하여 자연어처리 기초부터 고급 이론 및 실습까지 제공하고자 합니다.

실제 시스템을 구현하며 얻은 경험과 인사이트들을 공유하고자 하였고, 배경이 되는 수학적인 이론에서부터, 실제 PyTorch 코드, 그리고 실전에서의 꼭 필요한 개념들을 담을 수 있도록 하였습니다. 따라서, 현재 딥러닝을 활용한 State of the Art 기술 뿐만 아니라, 딥러닝 이전의 기존의 전통적인 방식부터 차근차근 설명하여, 왜 이 기술이 필요하고, 어떻게 발전 해 왔으며, 어떤 부분이 성능 개선을 만들어냈는지 쉽게 이해할 수 있도록 설명하고자 합니다.

- Github Repo: [https://github.com/kh-kim/nlp\\_with\\_pytorch](https://github.com/kh-kim/nlp_with_pytorch)
- Gitbook: <https://kh-kim.gitbook.io/natural-language-processing-with-pytorch/>

## 현재 작성 중인 챕터

대부분의 챕터는 작성이 완료되었으며, 아래의 챕터가 아직 작성 중입니다.

- Word Embedding Vector

## Pre-requisites

- Python
- Calculus, Linear Algebra
- Probability and Statistics
- Basic Machine Learning

- Objective / loss function
- Linear / logistic regression
- Gradient descent
- Basic Deep Learning
- Back-propagation
- Activation function

지은이]: 김기현(Kim, Ki Hyun)



Figure 1:

### 연락처

Name	Kim, Ki Hyun
email	pointzz.ki@gmail.com
linkedin	<a href="https://www.linkedin.com/in/ki-hyun-kim/">https://www.linkedin.com/in/ki-hyun-kim/</a>
github:	<a href="https://github.com/kh-kim/">https://github.com/kh-kim/</a>

### 이력

- Principal Machine Learning Engineer @ Makinarocks
- Machine Learning Researcher @ SKPlanet
- Neural Machine Translation: Global 11번가
- Machine Learning Engineer @ Ticketmonster
- Recommender System: TMON
- Researcher @ ETRI

- Automatic Speech Translation: GenieTalk [Android], [iOS], [TV Ads]

## 패스트캠퍼스 강의 소개



현재 이 책을 바탕으로 패스트캠퍼스에서 자연어처리 입문 캠프, 자연어처리 심화 캠프도 진행하고 있습니다.

이 저작물은 크리에이티브 커먼즈 저작자표시-비영리-동일조건변경허락(BY-NC-SA)에 따라 이용할 수 있습니다.



# Index

- 소개글
- Natural Language Processing with Deeplearning
- Intro
- Deeplearning
- Why NLP is difficult
- Why Korean NLP is Hell
- Recent Trends
- Hello PyTorch
- Intro
- How to install
- PyTorch tutorial

- Word Senses: Similarity and Ambiguity
- Intro
- WordNet
- Appendix: TF-IDF
- How to get Similarity
- Word Sense Disambiguation
- Appendix: Monty–Hall Problem
- Selectional Preference
- Conclusion
- Preprocessing
- Intro
- Collecting corpus
- Cleaning corpus
- Tokenization (POS Tagging)
- Aligning parallel corpus
- Subword Segmentation
- Detokenization
- TorchText
- Word Embedding Vector
- Intro
- One-hot encoding
- Previous Methods
- Word2Vec
- GloVe
- FastText
- Doc2Vec
- Sequence Modeling
- Intro
- Recurrent Neural Network
- Long Short Term Memory
- Gated Recurrent Unit
- Gradient Clipping
- Text Classification
- Intro
- Naive Bayes
- Using RNN
- Using CNN
- Unsupervised Text Classification

- Language Modeling
- Intro
- n-gram
- Perplexity
- Appendix: Mean Square Error (MSE)
- n-gram Exercise
- Neural Network Language Model
- Applications
- Neural Machine Translation
- Intro
- Sequence-to-Sequence
- Attention
- Input Feeding
- Auto-regressive and Teacher Forcing
- Search
- Evaluation
- Source Code
- Advanced Topic on NMT
- Multilingual NMT
- Using Monolingual Corpora
- Fully Convolutional Seq2seq
- Transformer
- NLP with Reinforcement Learning
- Intro
- Math Basics
- Reinforcement Learning Basics
- Policy Gradients
- Reinforcement Learning on NLG
- Supervised NMT
- Unsupervised NMT
- Exploit Duality
- Duality
- Dual Supervised Learning
- Dual Unsupervised Learning
- Productization
- Pipeline
- Google's NMT
- Edinburgh's NMT

- Booking.com’s NMT
- Microsoft’s NMT
- References

# Natural Language Processing with Deeplearning



Figure 1: TARS from Interstellar – Image from web

## What is Natural Language Processing?

자연어처리(natural language processing, NLP) 분야는 인공지능의 큰 줄기 중에 하나입니다. 특히, 컴퓨터에게 사람이 사용하는 언어를 처리하고 이해하도록 함으로써, 사람과 컴퓨터 사이의 매개체 또는 인터페이스 역할을 할 수 있습니다. 따라서 computer science 뿐만 아니라, linguistics와 같은 다른 학문과의 융합적인 요소도 갖고 있습니다. 실제로 NLP는 아래와 같이 세부적인 주제로 나누어 볼 수 있습니다.

---

주제	설명
Phonetics and Phonology	the study of linguistic sounds
Morphology	the study of the meaning of components of words
Syntax	the study of the structural relationships between words
Semantics	the study of meaning
Discourse	they study of linguistic units larger than a single utterance

---

[Gao et al.2017]

따라서, 이러한 NLP의 세부적인 부분들이 합쳐져 최종적인 목표는 사람의 언어를 이해하여 컴퓨터로 하여금 여러가지 tasks를 수행할 수 있도록 하는 것입니다. 컴퓨터는 이제 우리와 뗄 수 없는 존재가 되었고, 그러므로 이미 실제로 NLP는 우리의 일상에 가장 깊숙히 들어와 있는 분야이기도 합니다. NLP에 의해서 수행되는 대표적인 task 또는 응용분야들은 다음과 같습니다.

- Siri, Alexa와 같이 사용자의 의도를 파악하고 대화하거나 도움을 주는 task
- 요약, 번역과 같은 task
- 감성분석과 같이 대량의 텍스트를 이해하고 수치화 하는 task
- 사용자로부터 입력을 받아 사용자가 원하는 것을 검색 및 답변을 주는 task

우리는 이 책을 통해서 위의 task들을 위한 기술들의 대부분을 다루고자 합니다. 위의 대부분의 기술들이 deep learning에 의해서 비약적인 발전이 있었지만, 그 기반이 되는 수십년의 역사 또한 중요합니다. 따라서 최신 기술에 대해서 다루기 위해서는 그 이전의 기술들에 대해서도 다루고, 무엇이 문제였으며, 어떻게 최신의 기술이 어떤 돌파구를 마련했는지 아는 것도 중요합니다. 따라서, 이 책은 이러한 task들에 대한 최신 기술 뿐만 아니라, deep learning 이전의 주요 기술들에 대해서도 간략히 다루어 기초부터 차근차근 쌓아올릴 수 있도록 하고자 합니다. # Deep Learning

딥러닝의 시대가 오고 딥러닝은 하나하나 머신러닝의 분야들을 정복해 나가기 시작했습니다. 가장 먼저 두각을 나타낸 곳은 ImageNet이었지만, 가장 먼저 상용화 부문에서 빛을 본 것은 음성인식 분야였습니다. 음성인식은 여러 components 중에서 고작 하나인 GMM을 DNN으로 대체하였지만, 성능에 있어서 십수년의 정체를 뚫고 한 차례 큰 발전을 이루어냈습니다. 상대적으로 가장 나중에 빛을 본 곳은 NLP분야였습니다. 아마도 image classification과 음성인식의 phone recognition과 달리 NLP는 sequential한 데이터라는 것이 좀 더 장벽으로 다가왔으리라 생각됩니다. 하지만, 결국엔 attention의 등장으로 인해서 요원해 보이던 기계번역 분야마저 end-to-end deep learning에 의해서 정복되게 되었습니다.

## Brief Introduction to History of Deep Learning

### Before 2010's

인공신경망을 위시한 인공지능의 유행은 지금이 처음이 아닙니다. 이전까지 두 번의 대유행이 있었고, 그에 따른 두 번의 빙하기가 있었습니다. 80년대에 처음 back-propagation이 제안된 이후로, 모든 문제는 해결 된 듯 해 보였습니다. 하지만, 다시금 여러가지 한계점을 드러내며 침체기를 맞이하였습니다. 모두가 인공신경망의 가능성을 부인하던 2006년, Hinton 교수는 Deep Belief Networks을 통해 여러 층의 hidden layer를 효과적으로 pretraining 시킬 수 있는 방법을 제시하였습니다. 하지만, 아직까지 가시적인 성과가 나오지 않았기 때문에 모두의 관심을 집중 시킬 순 없었습니다. 아~ 그런가보다 하고 넘어가는 수준이었겠지요.

실제로 주변의 90년대의 빙하기를 겪어보신 세대 분들은 처음 딥러닝이 주목을 끌기 시작했을 때, 모두 부정적인 반응을 보이기 마련이었습니다. 계속 해서 최고 성능을 갈아치우며, 모두가 열광할 때에도, 단순한 잠깐의 유행일 것이라 생각하는 분들도 많았습니다. 하지만 점차 딥러닝은 여러 영역을 하나둘 정복 해 나가기 시작했습니다.

### Image Recognition

2012년 이미지넷에서 인공신경망을 이용한 AlexNet([Krizhevsky et al. 2012])은 경쟁자들을 큰 차이로 따돌리며 우승을 하고, 딥러닝의 시대의 서막을 올립니다. AlexNet은 여러 층의 Convolutional Layer을 쌓아서 architecture를 만들었고, 기존의 우승자들과 확연한 실력차를 보여주었습니다. 당시에 AlexNet은 3GB 메모리의 Nvidia GTX580을 2개 사용하여 훈련하였는데, 지금 생각하면 참으로 격세지감이 아닐 수 없습니다.

이후, ImageNet은 딥러닝의 경연장이 되었고, 거의 모든 참가자들이 딥러닝을 이용하여 알고리즘을 구현하였습니다. 결국, ResNet([He et al. 2015])은 Residual Connection을 활용하여 150층이 넘는 deep architecture를 구성하며 우승하였습니다.

하지만, 사실 연구에서와 달리, 아직 실생활에서의 image recognition은 아직 다른 분야에 비해서 어려움이 있는 것은 사실입니다. image recognition 자체의 어려움이 워낙 높기 때문입니다. 따라서 아직도 이와 관련해서 산업계에서는 많은 연구와 개발이 이어지고 있습니다.

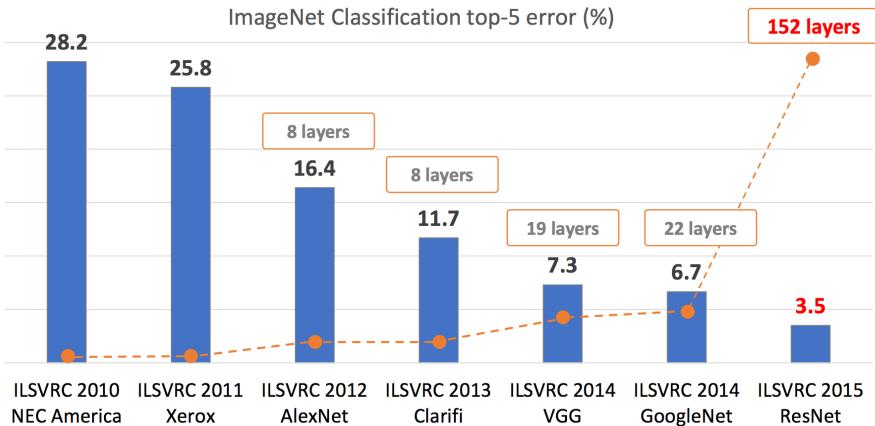


Figure 2: Recent History of ImageNet

## Speech Recognition

음성인식에 있어서도 딥러닝(당시에는 Deep Neural Network라는 이름으로 더욱 유명하였습니다.)을 활용하여 큰 발전을 이룩하였습니다. 오히려 이 분야에서는 vision분야에 비해서 딥러닝 기술을 활용하여 상용화에까지 성공한 더욱 인상적인 사례라고 할 수 있습니다.

사실 음성인식은 2000년대에 들어 큰 정체기를 맞이하고 있었습니다. GMM(Gaussian Mixture Model)을 통해 phone을 인식하고, 이를 HMM(Hidden Markov Model)을 통해 sequential하게 modeling하여 만든 Acoustic Model (AM)과 n-gram기반의 Language Model (LM)을 WFST(Weighted Finite State Transducer)방식을 통해 결합하는 전통적인 음성인식(Automatic Speech Recognition, ASR) 시스템은 위의 설명에서 볼 수 있듯이 너무나도 복잡한 구조와 함께 그 성능의 한계를 보이고 있었습니다.

그러던 중, 2012년 GMM을 DNN으로 대체하며, 십수년간의 정체를 단숨에 뛰어넘는 큰 혁명을 맞이하게 됩니다. (Vision, NLP에서 모두 보이는 익숙한 패턴입니다.) 그리고 점차 AM전체를 LSTM으로 대체하고, 또한 end-to-end model([Chiu et al.2017])이 점점 저변을 넓혀가고 있는 추세입니다.

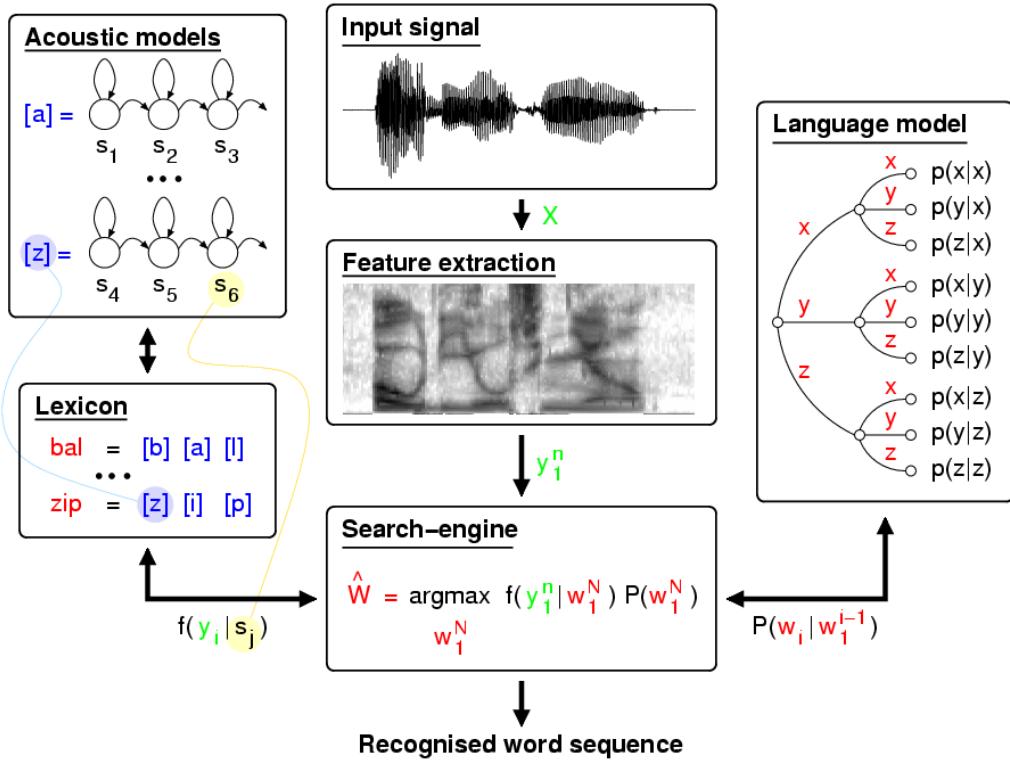


Figure 3: Traditional Speech Recognition System

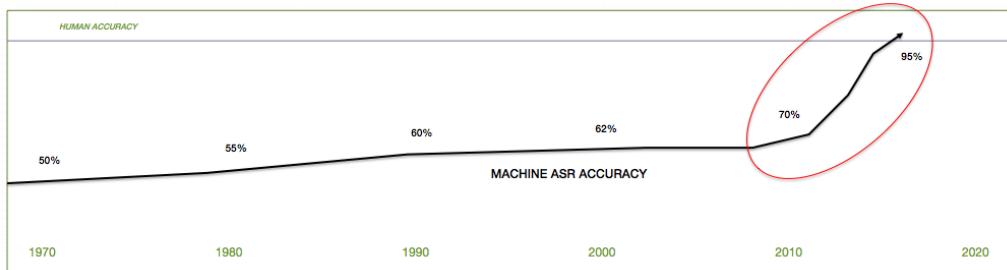


Figure 4: Accuracy of ASR



Figure 5: It was hard time for NLP until 2014.

## Machine Translation

사실 다른 인공지능의 다른 분야에 비해서 NLP 또는 기계번역 분야는 이렇다할 큰 성과를 거두지는 못하고 있었습니다. 하지만 결국 물밀듯이 밀려오는 딥러닝의 침략 앞에서 기계번역 또한 예외일 순 없었습니다. 딥러닝 이전의 기계번역은 통계 기반 기계번역(Statistical Machine Translation, SMT)가 지배하고 있었습니다. 비록 SMT는 규칙기반의 번역방식(Rule based Machine Translation, RBMT)에 비해서 언어간 확장이 용이한 장점이 있었고, 성능도 더 뛰어났지만, 음성인식과 마찬가지로 SMT는 역시 너무나도 복잡한 구조를 지니고 있었습니다.

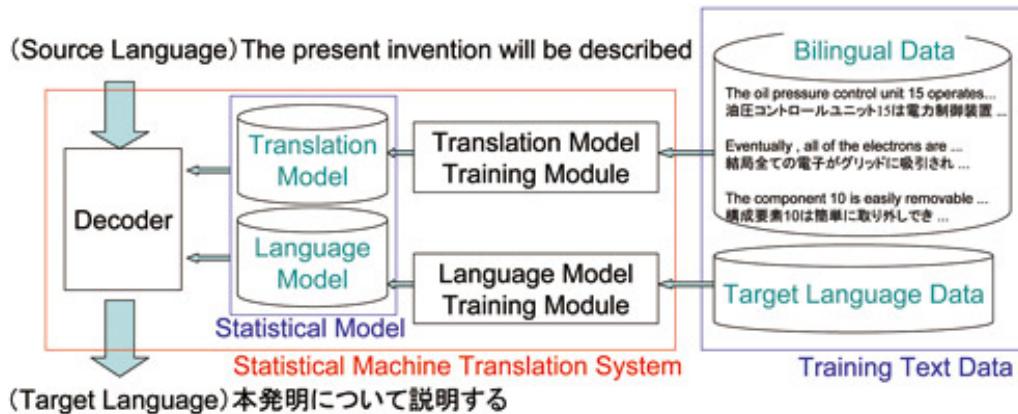


Figure 6: Sub-modules for Statistical Machine Translation (SMT)

2014년 Sequence-to-sequence(seq2seq)라는 architecture가 소개 되며, end-to-end neural machine translation의 시대가 열리게 되었습니다.

Seq2seq를 기반으로 attention mechanism([Bahdanau et al.2014], [Luong et al.2015])이 제안되며 결국 기계번역은 Neural Machine Translation에 의해서 대통합이 이루어지게 됩니다.

결국, 기계번역은 가장 늦게 혁명이 이루어졌지만, 가장 먼저 딥러닝만을 사용해 상용화가 된 분야가 되었습니다. 현재의 상용 기계번역 시스템은 모두 딥러닝에 의한 시스템으로 대체되었다고 볼 수 있습니다.

- 읽을거리:
- <https://devblogs.nvidia.com/introduction-neural-machine-translation-with-gpus/>
- <https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-2/>

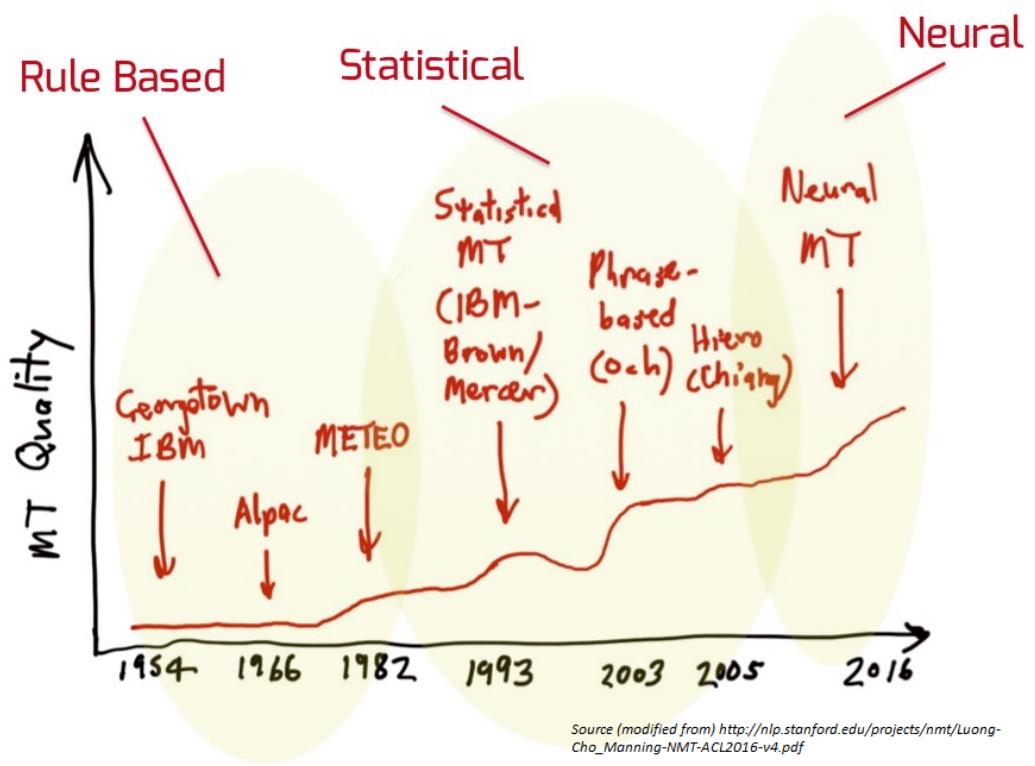


Figure 7: History of Machine Translation

- <https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-3/>

## Generative Learning

Neural Network은 pattern classification에 있어서 타 알고리즘에 비해서 너무나도 압도적인 성능을 보여주었기 때문에, image recognition, text classification과 같은 단순한 분류 문제(classification or discriminative learning)는 금방 정복되고 더 이상 연구자들의 흥미를 끌 수 없었습니다.

각 방식이 흥미를 두고 있는 것:

- Discriminative learning

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(Y|X; \theta)$$

- Generative learning

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(X; \theta)$$

따라서, 곧 연구자들은 또 다른 흥미거리를 찾아 나섰는데, 그것은 Generative Learning이었습니다. 기존의 classification 문제는  $X$ 가 주어졌을 때, 알맞은  $Y$ 를 찾아내는 것에 집중했다면, 이제는  $X$  자체에 집중하기 시작한 것입니다. 예를 들어 기존에는 사람의 얼굴 사진이 주어지면 남자인지 여자인지, 또는 더 나아가 이 사람이 누구인지 알아내는 것이었다면, 이제는 얼굴 자체를 묘사할 수 있는 모델을 훈련하고자 하였습니다.

이러한 과정에서 Adversarial learning (GAN, [Goodfellow et al.2014])이나 Variational Auto-encoder (VAE, [Kingma et al.2013])등이 주목받게 되었습니다. 아직 이러한 연구는 현재 진행형이라 할 수 있고, 이와 관련한 많은 문제들이 남아있습니다.

## Paradigm Shift on NLP from Traditional to Deep Learning

Deep learning 이전의 기존의 전형적인 NLP application의 구조는 보통 아래와 같습니다. Task에 따라서 phonology가 추가되기도 하고, 아래와 같이 여러가지 단계의 module로 구성되어 복잡한 디자인을 구성하게 됩니다. 따라서 매우 무겁고



Figure 8: generated by a progressively grown GAN trained on the CelebA-HQ dataset

복잡하여 구현 및 시스템 구성이 어려운 단점이 많았습니다. 더군다나, 각각의 module이 완벽하게 동작할 수 없기 때문에, 각기 발생한 error가 중첩 및 가중되어 뒤로 전파되는 error propagation등의 문제도 가질 수 있었습니다.

하지만, 위에서 언급한 기계번역의 사례처럼 NLP 전반에 걸쳐 deep learning의 물결이 들어오기 시작했습니다. 처음에는 각 sub-module을 대체하는 형태로 진행되었지만, 점차 기계번역의 사례처럼 결국 end-to-end model들로 대체되었습니다. 현재에도 chat-bot과 같은 아직 많은 task들에서 end-to-end learning이 이루어지지 않았지만, 최종적으로는 end-to-end model이 제안될 것이라 볼 수 있습니다.

Deep learning이 NLP에서도 주류가 되면서, 위와 같은 접근 방법의 변화들을 꼽을 수 있습니다. 사람의 언어는 Discrete한 symbol로 이루어져 있습니다. 비록 그 symbol간에는 유사성이 있을 수 있지만 기본적으로 모든 단어(또는 token)은 다른 symbol이라고 볼 수 있습니다. 따라서 기존의 전통적인 NLP에서는 discrete symbol로써 데이터를 취급하였습니다. 따라서 사람이 데이터를 보고 해석하기는 쉬운 장점이 있었지만, 모호성이나 유의성을 다루는데에는 어려움을 겪을 수 밖에 없었습니다.

하지만 word2vec등의 word embedding을 통해서 단어(또는 token)을 continuous한 vector로써 나타낼 수 있게 되고, 모호성과 유의성에서도 이득을 볼 수 있게 되었습니다. 또한, deep learning의 장점을 잘 살려 end-to-end model을

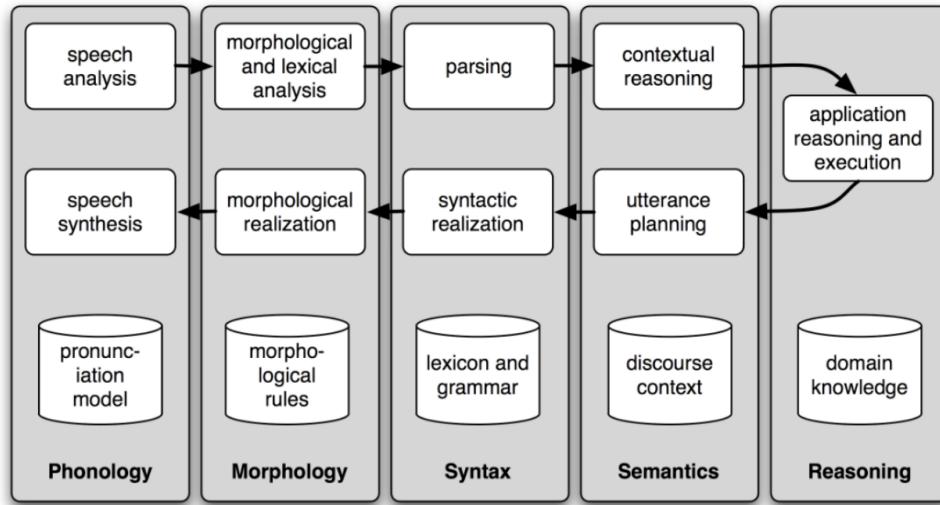


Figure 9: [Gao et al.2017]

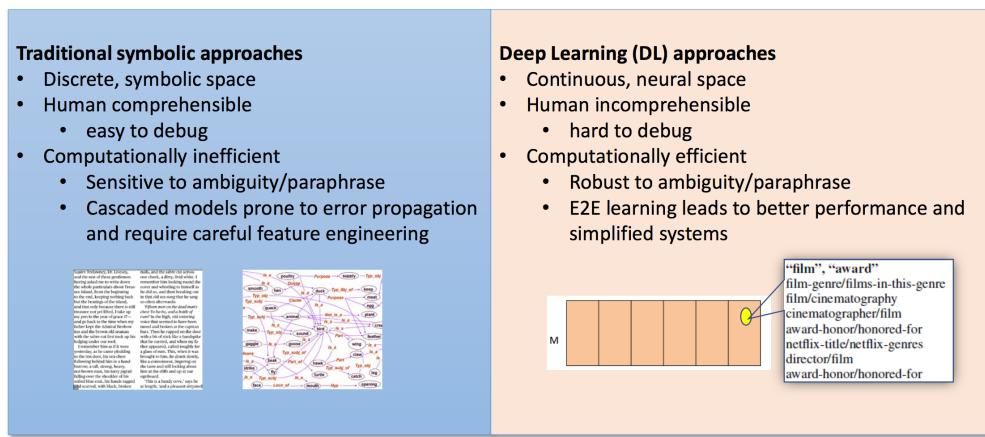


Figure 10: [Gao et al.2017]

구현함으로써 더욱 높은 성능을 뽑을 수 있게 되었습니다. 또한, RNN의 단점을 보완한 LSTM과 GRU에 대한 활용법이 고도화 되었고, attention의 등장으로 인해서 긴 time-step의 sequential 데이터에 대해서도 어렵지 않게 훈련할 수 있게 된 점도 큰 터닝 포인트라고 볼 수 있습니다.

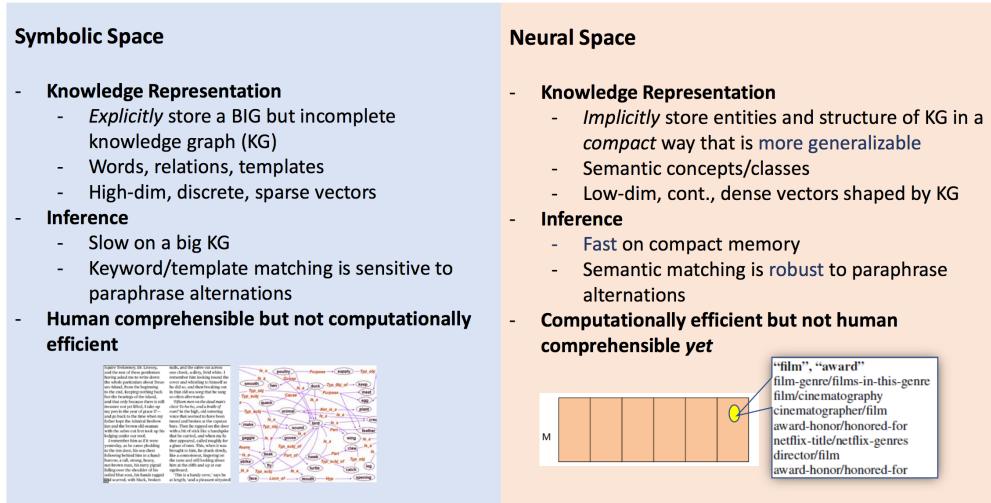
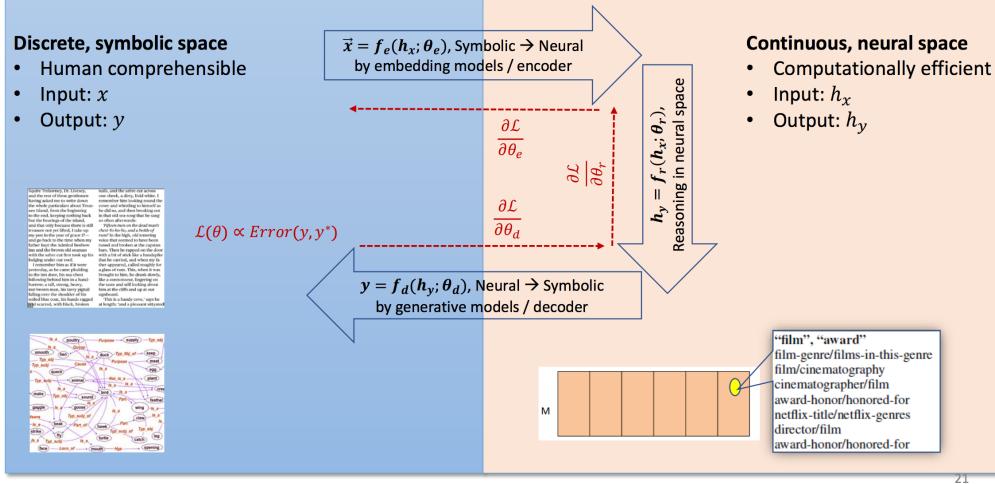


Figure 11: [Gao et al.2017]

## Conclusion

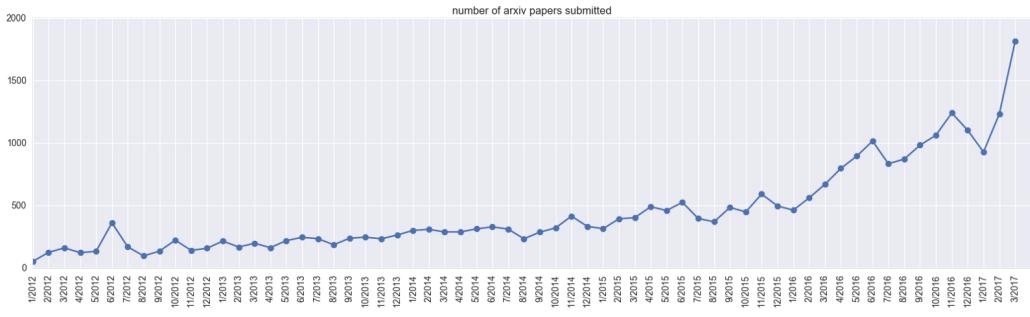
다시 말해, 비록 NLP는 discrete한 symbol로 이루어진 사람의 언어를 다루는 분야이지만, 성공적인 word embedding과 long term sequential data에 대한 효과적인 대응방법이 나옴에 따라서, 점차 다른 인공지능의 분야들처럼 큰 발전이 이루어지게 되었습니다.

이렇게 deep learning이 널리 퍼짐에 따라, NLP를 포함한 인공지능의 여러 분야에서 모두 큰 발전과 성공을 보이고 있습니다. 따라서, 이 책은 기존의 전통적인 방식과 새롭게 제안된 최신의 deep learning 기술을 모두 소개하고자 합니다.



21

Figure 12: [Gao et al.2017]



## Why NLP is difficult?

음성인식은 눈에 보이지 않는 signal을 다룹니다. 보이지도 않는 가운데 noise와 signal을 가려내야 하고, 소리의 특성상 noise와 signal은 그냥 더해져서 나타납니다. 게다가 time-step은 무지하게 길어요. 어려울 것 같습니다. 그렇다면 눈에 보이는 computer vision을 생각 해 보죠. 그런데 computer vision은 눈에 보이지만 이미지는 너무 크고, 다양합니다. 심지어 내 눈에는 다 똑같은 색깔인데 사실은 알고보면 다른 색이라고 합니다. 그럼 애초에 discrete한 단어들로 이루어져 있는 자연어처리를 해 볼까요? 그럼 NLP는 쉬울까요? 하지만 세상에 쉬운일은 없죠… Natural language processing도 다른 분야 못지않게 매우 어렵습니다. 어떠한 점들이 NLP를 어렵게 만드는 것일까요?

사람은 언어를 통해 타인과 교류하고, 의견과 지식을 전달 합니다. 소리로 표현된 말을 석판, 나무, 종이에 적기 시작하였고 사람의 지식은 본격적으로 축적되기 시작하였습니다. 이와 같이 언어는 사람의 생각과 지식을 내포하고 있습니다. 컴퓨터로 하여금 이러한 사람의 언어를 이해할 수 있게 한다면 컴퓨터에게도 지식과 의견을 전달 할 수 있을 것 입니다.

## Ambiguity

아래의 문장을 한번 살펴볼까요. 어떤 회사의 번역이 가장 정확한지 살펴 볼까요.  
(2018년 4월 기준)

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

---

Google  
was  
kick-  
ing  
her  
in  
the  
car  
that  
went  
to  
the  
park  
for  
tea.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

Micr**o**soft

was  
a  
car  
to  
her,  
in  
the  
car  
I  
had  
a  
car  
and  
went  
to  
the  
park.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

———  
Naver

got  
dumped  
by  
her  
on  
the  
way  
to  
the  
park  
for  
tea.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

Kakab

was  
in  
the  
car  
go-  
ing  
to  
the  
park  
for  
tea  
and  
I  
was  
in  
her  
car.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

---

SK I  
got  
dumped  
by  
her  
in  
the  
car  
that  
was  
go-  
ing  
to  
the  
park  
for  
a  
cup  
of  
tea.

---

안타깝게도 완벽한 번역은 없는 것 같습니다. 같은 차라는 단어가 세 번 등장하였고, 모두 다른 의미를 지니고 있습니다: tea, car, and kick (or dump). 일부는 표현을 빼뜨리기도 하였고, 다른 일부는 단어를 헛갈린 것 같습니다. 이렇게 단어의 중의성 때문에 문장을 해석하는데 모호함이 생기기도 합니다. 또 다른 상황을 살펴보겠습니다.

---

원문	나는 철수를 안 때렸다.
해석#1	철수는 맞았지만, 때린 사람이 나는 아니다.
해석#2	나는 누군가를 때렸지만, 그게 철수는 아니다.
해석#3	나는 누군가를 때린 적도 없고, 철수도 맞은 적이 없다.

---

위와 같이 문장 내 정보의 부족으로 인한 모호성이 발생 할 수 있습니다. 사람의 언어는 효율성을 극대화 하도록 발전하였기 때문에, 최소한의 표현으로 최대한의 정보를 표현하려 합니다. 따라서 앞뒤 문장의 context에 따라서 문장의 의미는 달라질 것입니다. 사실 첫 예제의 차도 주변 단어들(context)를 보면 중의성을 해소(word sense disambiguation)할 수 있습니다. 아래의 예제도 문장 내 정보의 부족이 야기한 구조 해석의 문제입니다.

---

원문	선생님은 울면서 돌아오는 우리를 위로 했다.
해석#1	(선생님은 울면서) 돌아오는 우리를 위로 했다.
해석#2	선생님은 (울면서 돌아오는 우리를) 위로 했다.

---

## Paraphrase

---

번호	표현
1.	여자가 김치를 어떤 남자에게 집어 던지고 있다.
2.	여자가 어떤 남자에게 김치로 때리고 있다.
3.	여자가 김치로 싸대기를 날리고 있다.
4.	여자가 배추 김치 한 포기로 남자를 때리고 있다.
5.	여자가 김치를 사용해 남자를 때리고 있다.
6.	남자가 여자에게 김치로 싸대기를 맞고 있다.
7.	남자가 여자로부터 김치로 맞고 있다.
8.	김치가 여자에게 무기로 사용되어 남자를 후려치고 있다.
9.	김치가 여자에게서 남자에게로 날라가고 있다.

---

영화나 드라마의 어떤 장면을 말로 표현 한다고 해 봅시다. 그럼 아주 다양한 표현이 나올 것 입니다. 하지만 알고보면 다 같은 장면을 묘사하고 있는 것이고, 그 안에 포함된 의미는 같다고 할 수 있을 것 입니다. 이와 같이 문장의 표현 형식은 다양하고, 비슷한 의미의 단어들이 존재하기 때문에 paraphrase의 문제가



Figure 14: 김치 싸대기로 유명한 드라마 모두 다 김치의 장면

존재합니다. 더군다나, 위에서 지적 한 것 처럼 미묘하게 사람들이 이해하고 있는 단어의 의미는 다를 수도 있을 것 입니다. 따라서 이 또한 더욱 paraphrase 문제의 어려움을 가중시킵니다.

### Discrete, Not Continuous

사실은 discrete하기 때문에 그동안 쉽다고 느껴졌습니다. 하지만 neural network에 적용하기 위해서는 continuous한 값으로 바꾸어주어야 합니다. Word embedding이 그 역할 훌륭하게 수행하고 있긴 합니다. 하지만 애초에 continuous한 값이 아니었기 때문에 neural network 상에서 여러가지 방법을 구현할 때에 제약이 존재합니다.

### Curse of Dimensionality

Discrete한 데이터이기 때문에 많은 종류의 데이터를 표현하기 위해서는 엄청난 dimension이 필요합니다. 예전에는 각 단어를 discrete한 symbol로 써 다루었기 때문에, 마치 vocabulary size =  $|V|$  만큼의 dimension이 있는 것이나 마찬가지였습니다. 이러한 sparseness를 해결하기 위해서 단어를 적절하게 segmentation하는 등

여러가지 노력이 필요하였습니다. 다행히 적절한 word embedding을 통해서 dimension reduction을 하여 이 문제를 해결함으로써, 이제는 이러한 문제는 예전보다 크게 다가오진 않습니다.

## Noise and Normalization

모든 분야의 데이터에서 noise를 signal로 부터 적절히 분리해 내는 일은 매우 중요합니다. 그러한 과정에서 자칫 실수하면 data는 본래의 의미마저 같이 잃어버릴 수도 있습니다. 이러한 관점에서 NLP는 어려움이 존재 합니다. 특히, 다른 종류의 데이터에 비해서 데이터가 살짝 바뀌었을 때의 의미의 변화가 훨씬 크기 때문입니다. 예를 들어 이미지에서 한 픽셀의 RGB값이 각각 0에서 255까지로 나타내어지고, 그 값중 하나의 수치가 1이 바뀌었다고 해도 해당 이미지의 의미는 변화가 없다고 할 수 있습니다. 하지만 단어는 discrete한 symbol이기 때문에, 단어가 살짝만 바뀌어도 문장의 의미가 완전히 다르게 변할 수도 있습니다. 또한, 마찬가지로 띠어쓰기나 어순의 차이로 인한 정제의 이슈도 큰 어려움이 될 수 있습니다. 이러한 어려움을 다루고 해결하기 위한 방법을 Preprocessing 챕터에서 다루도록 하겠습니다. # 무엇이 한국어 NLP를 더욱 어렵게 만드는가?

## 교착어 (어순)

종류	대표적 언어	특징
교착어	한국어, 일본어, 몽골어	어간에 접사가 붙어 단어를 이루고 의미와 문법적 기능이 정해짐
굴절어	라틴어, 독일어, 러시아어	단어의 형태가 변함으로써 문법적 기능이 정해짐
고립어	영어, 중국어	어순에 따라 단어의 문법적 기능이 정해짐

한국어는 교착어에 속합니다. 어순이 중요시되는 영어/중국어와 달리 어근에 접사가 붙어 의미와 문법적 기능이 부여됩니다. (굴절어의 경우에는 형태 자체가 변함으로써, 어근과 접사가 분명하게 구분되는 교착어와 다릅니다.) 따라서, 아래와 같은 재미있는 예시의 형태도 가능합니다.

위의 예처럼 접사에 따라 단어의 역할이 정의되기 때문에, 상대적으로 어순은 중요하지 않습니다. 아래는 4개의 단어가 나타날 수 있는 모든 조합을 적은 것입니다. “간다”가 “밥을” 뒤에 붙어 수식할 때를 제외하면 모두 같은 의미의 문장이 됩니다.

쉬운 예시로 우리말 "잡하시었겠더라"를 생각해 보자. 위 각주에서 설명하였듯 낱말 형성(조어)의 측면에서는 어근-접사로 나뉘고, 활용의 측면에서는 어간-어미로 나뉜다.  
[2]

어근	접사				
	파생 접사		굴절 접사		
잡-	-하-	-으)시-	-았-	-겠-	-더라 <sup>[3]</sup>
어간			선어말 어미		어말 어미
			어미		

각 파생+굴절 접사의 기능이 앞에서부터 피동, 주체 높임, 과거 시제, 추측, 전달임을 알 수 있다. 각각의 쓰임새가 분명하기에 여러 접사가 줄줄이 붙는다.

Figure 15: [Image from 나무위키(교착어)]

번호	문장	정상여부
1.	나는 밥을 먹으러 간다.	O
2.	간다 나는 밥을 먹으리.	O
3.	먹으려 간다 나는 밥을.	O
4.	밥을 먹으려 간다 나는.	O
5.	나는 먹으려 간다 밥을.	O
6.	나는 간다 밥을 먹으리.	O
7.	간다 밥을 먹으려 나는.	O
8.	간다 먹으려 나는 밥을.	O
9.	먹으려 나는 밥을 간다.	X
10.	먹으려 밥을 간다 나는.	X
11.	밥을 간다 나는 먹으리.	X
12.	밥을 나는 먹으려 간다.	O
13.	나는 밥을 간다 먹으리.	X
14.	간다 나는 먹으려 밥을.	O
15.	먹으려 간다 밥을 나는.	O
16.	밥을 먹으려 나는 간다.	O

이러한 특징은 Parsing, POS Tagging 부터 Language Modeling에 이르기까지 한국어 NLP를 훨씬 어렵게 만드는 이유 중에 하나입니다.

또한 접사가 붙어 같은 단어가 다양하게 생겨나기 때문에, 하나의 어근에서 생겨난 비슷한 의미의 단어가 정말 많이 생성됩니다. 따라서 이들을 모두 다르게 처리할 수 없기 때문에, 추가적인 segmentation을 통해서 같은 어근에서 생겨난 단어를 처리하게 됩니다. 이와 관련한 내용은 Preprocessing 챕터에서 다루도록 하겠습니다.

- 읽을거리:
- <http://zomzom.tistory.com/1074><sup>23</sup>
- <https://m.blog.naver.com/reading0365/221057575669>

## 띄어쓰기의 어려움

애초에 동양권에서는 띄어쓰기라는 것이 존재하지 않았고 근대에 들어와서 도입된 것이기 때문에 띄어쓰기에 맞춰 발전 해 온 언어는 아닙니다. 따라서 띄어쓰기에



Figure 16: 내동생 고기 vs 내동 생고기



Figure 17: 농협용인육가공 vs 농협 용인 육가공

추가적인 segmentation을 통해서 띠어쓰기를 정제(normalization) 해 주는 process가 마찬가지로 필요하게 됩니다.

## 평서문과 의문문의 차이

언어	평서문	의문문
영어	I ate my lunch.	Did you have lunch?
한국어	점심 먹었어.	점심 먹었어?

물론 한국어에서도 의문문을 나타낼 수 있는 접사가 있습니다 – 니. 따라서, 점심 먹었이라고 하면 굳이 물음표가 붙지 않더라도 의문문인 것을 알 수 있습니다. 하지만 많은 경우에 그냥 의문문과 평서문이 같은 형태의 문장을 띠는 것이 사실입니다. 따라서, 마침표나 물음표가 붙지 않을 경우에는 알 수가 없는 경우가 많습니다. 특히나, 음성인식의 결과물로 나오는 text의 경우에는 더욱더 어렵습니다.

## 주어 생략

영어는 기본적으로 특성상 명사가 굉장히 중요시 됩니다. 따라서 정말 특별한 경우를 제외하고는 주어가 생략되는 경우가 없습니다. 하지만 한국어는 동사를 중요시하기 때문에, 주어가 자주 생략됩니다. 인간은 context 정보를 잘 활용하여 생략된 정보를 메꿀 수 있지만, 컴퓨터는 할 수가 없습니다. 따라서 위의 평서문과 의문문의 예에서도 볼 수 있듯이 한국어는 주어가 생략 되었는데, 컴퓨터는 누가 점심을 먹었고 누구에게 점심을 먹었냐고 물어보는지 알 수가 없습니다. 따라서 기계번역을 비롯하여 문장의 정확한 의미를 파악하는데 상당히 어렵게 됩니다.

- 읽을거리:
- <http://www.hani.co.kr/arti/society/schooling/261322.html>
- <https://namu.wiki/w/%EC%A3%BC%EC%96%B4%EB%8A%94%20%EC%97%86%EB%8B%A4>

## 한자 기반의 언어

언어	단어	조합
영어	Concentrate	con(=together) + centr(=center) + ate(=make)

언어	단어	조합
한국어	집중(□□)	□(모을 집) + □(가운데 중)

이와 같이 원래 한국어도 한자 기반의 언어이기 때문에, 한자의 조합으로 이루어진 단어들이 많습니다. 이러한 단어들은 각 글자가 의미를 지니고 있고 그 의미들이 합쳐져 하나의 단어의 뜻을 이루게 됩니다. 이것은 영어에서도 마찬가지입니다. “water”와 같이 잉글로색슨족의 언어가 기원인 단어가 아니라면, latin 기반의 단어들은 각기 뜻을 가진 sub-word들이 합쳐져서 하나의 단어의 의미를 이루게 됩니다.

하지만 문제는 한글이 한자를 대체하면서 생겨났습니다. 한자는 표어 문자입니다. 문자 하나당 하나의 단어를 나타냅니다. 읽는 소리는 같을 지라도, 형태와 그 뜻은 다릅니다. 하지만 이것을 표음 문자인 한글이 나타내게 되면서, 정보의 손실이 생겨버렸습니다. (사실 표음 문자가 가장 발달한 글자의 형태입니다.) 인간은 이러한 정보의 손실로 생겨난 모호성(ambiguity)의 문제를 context를 통해서 효과적으로 해소할 수 있지만, 컴퓨터는 그렇지 못합니다. 따라서 다른 언어에 비해서 중의성이 더 가중되어 버렸습니다.

Type	Text
<hr/>	
원문	저는
여기	
한	
가지	
문제점이	
있다고	
생각합니다.	
<hr/>	
형태	쪽에
따른	는
segmentation	
한	
가지	
문제점	
이	
있	
다고	
생각	
합니다	
<hr/>	

---

TypeText  
coun□저  
based□는  
sub-□여기  
word□한  
segmentation  
□문  
제  
점  
□이  
□있  
□다고  
□생각  
□합니다  
□.  
—

Preprocessing 챕터에서 다루겠지만, 이렇게 마지막까지 subword level로 segmentation할 경우에, 더욱 중의성 문제가 가중되어 버립니다.

문제점(□□□)이라는 단어가 문(□, 물을 문) 제(□, 제목 제) 점(□, 점 점)이라고 각각 segmentation 되었습니다. 하지만 결제(□□)의 제(□, 건널 제)도 있고, 제공(□□)의 제(□, 끌 제)도 있을 겁니다. 그런데 neural network에서 제라는 token은 결국 embedding vector로 변환이 될 겁니다. 따라서, embedding vector는 제(□, 제목 제), 제(□, 건널 제), 제(□, 끌 제) 세가지 모두에 대해서 embedding을 하게 될 것이고 저 뜻의 중앙 방향으로 vector가 애매하게 embedding 될 것 입니다. 사실, 굳이 neural network로 가지 않더라도, traditional NLP에서도 헷갈릴 것 또한 자명한 사실입니다. # Recent Trends in NLP

## Conquering on Basic NLP

이전에 다루었던 대로, 인공지능의 다른 분야에 비해서 NLP는 가장 늦게 빛을 보기 시작하였다고 하였지만, 여러 task에 deep learning을 적용하려는 시도는 많이 이루어졌고, 진전은 있었습니다. 2010년에는 RNN을 활용하여 language modeling을 시도[Mikolov et al.2010][Sundermeyer et al.2012]하여 기존의 n-gram 기반의 language model의 한계를 극복하려 하였습니다. 그리하여 기존의 n-gram 방식과의 interpolation을 통해서 더 나은 성능의 language model을 만들어낼 수

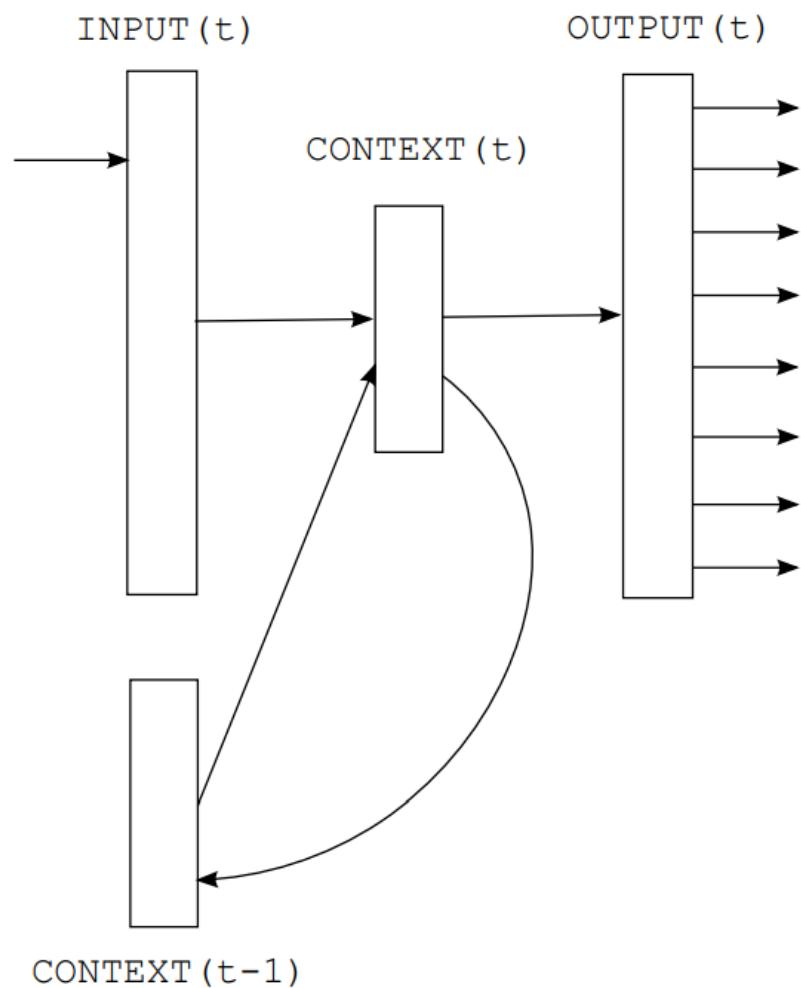


Figure 1: *Simple recurrent neural network.*

Figure 18:

있었지만, 기존에 language model이 사용되던 음성인식과 기계번역에 적용되기에는 구조적인 한계(Weighted Finite State Transducer, WFST의 사용)로 인해서 더 큰 성과를 거둘 수는 없었습니다. – 애초에 n-gram 기반 언어모델의 한계는 WFST에 기반하였기 때문이라고도 볼 수 있습니다. 닭이 먼저냐, 달걀이 먼저냐의 문제와 같음.

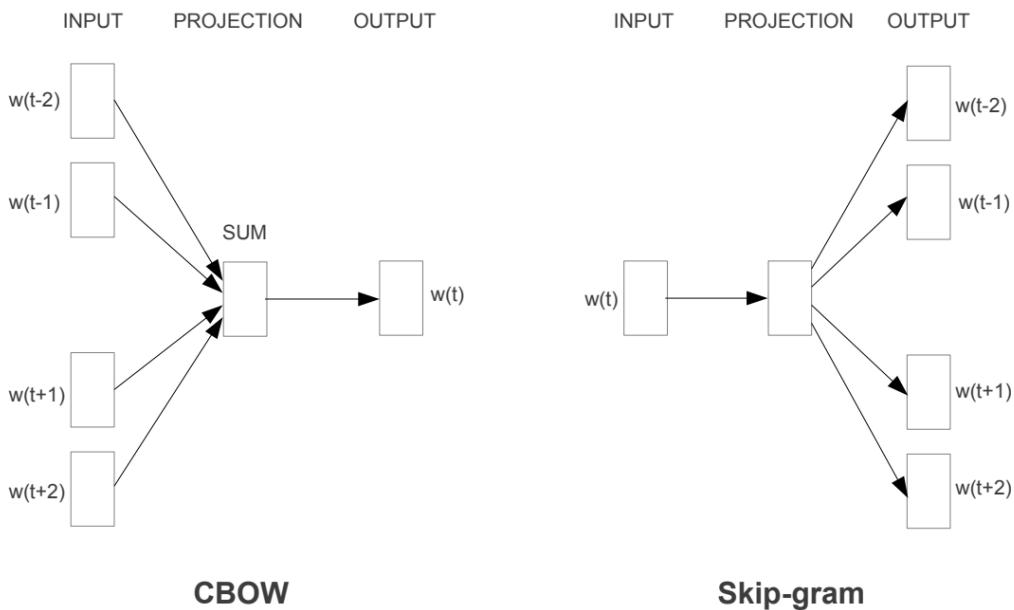


Figure 19:

그러던 와중에 Mikolov는 2013년 Word2Vec[Mikolov et al.2013]을 발표합니다. 단순한 구조의 neural network를 사용하여 효과적으로 단어들을 hyper plane(또는 vector space)에 성공적으로 projection(투영) 시킴으로써, 본격적인 NLP 문제에 대한 딥러닝 활용의 신호탄을 쏘아 올렸습니다. 아래와 같이 우리는 고차원의 공간에 단어가 어떻게 배치되는지 알 수 있음으로 해서, deep learning을 활용하여 NLP에 대한 문제를 해결하고자 할 때에 network 내부는 어떤식으로 동작하는지에 대한 insight를 얻을 수 있었습니다.

이때까지는 문장이란 단어들의 time series이기 때문에, 당연히 Recurrent Neural Network(RNN)을 통해 해결해야 한다는 고정관념이 팽배해 있었습니다 – Image=CNN, NLP=RNN. 하지만 2014년, Kim은 CNN만을 활용해 기존의 Text Classification보다 성능을 끌어올린 방법을 제시[Kim et al.2014]하며 한차례 파란을 일으킵니다. 이 방법은 word embedding vector와 결합하여 더 성능을 극대화 할

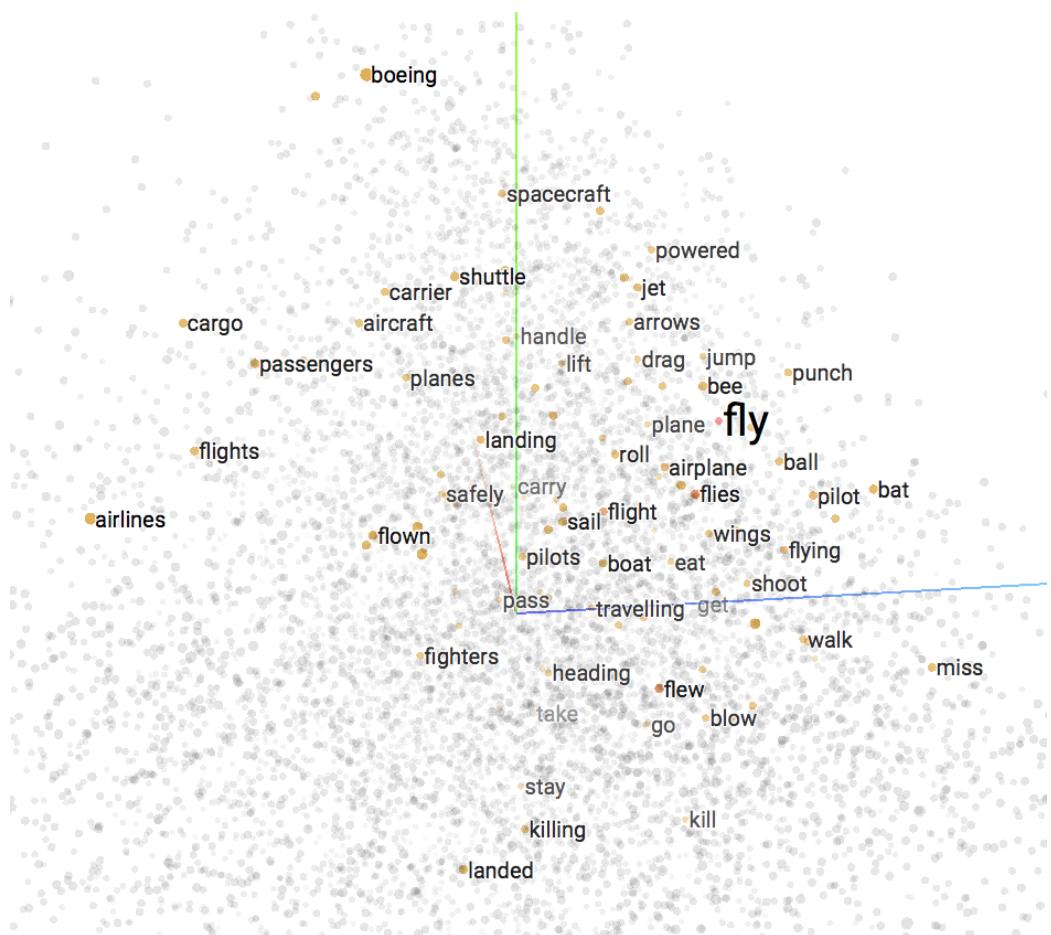


Figure 20:

수 있었습니다. 위의 paper를 통해서 학계는 NLP에 대한 시각을 한차례 더 넓힐 수 있게 됩니다.

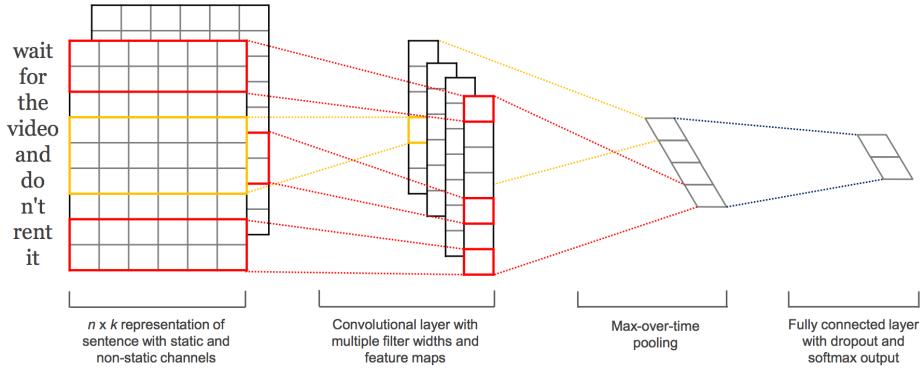


Figure 21:

이외에도 POS(Part-of-Speech) tagging, Sentence parsing, NER(Named Entity Recognition), SR(Semantic Role) labeling 등에서도 기존의 state of the art를 뛰어넘는 성과를 이루냅니다. 하지만 딥러닝의 등장으로 인해 대부분의 task들이 end-to-end를 통해 문제를 해결하고자 함에 따라, (또한, 딥러닝 이전에도 이미 매우 좋은 성과를 내고 있었거나, 딥러닝의 적용 후에도 큰 성능의 차이가 없음에) 큰 파란을 일으키지는 못합니다. – 당연히 그정도는 좋아지는거 아니야? 이런 느낌…?

## Flourish of NLG

2014년 NLP에 큰 혁명이 다가옵니다. Sequence-to-Sequence의 발표[Sutskever et al.2014]에 이어, Attention 기법이 개발되어 성공적으로 기계번역에 적용[Bahdanau et al.2014]하여 큰 성과를 거둡니다. 이에 NLP분야는 일대 혁명을 맞이합니다. 기존의 한정적인 적용 사례에서 벗어나, 주어진 정보에 기반하여 자유롭게 문장을 생성할 수 있게 된 것입니다. 따라서, 기계번역 뿐만 아니라, summarization, 챗봇 등 더 넓고 깊은 주제의 NLP의 문제를 적극적으로 해결해보려 시도 할 수 있게 되었습니다.

또한, 이와 같이 NLP 분야에서 딥러닝을 활용하여 큰 성과를 거두자, 더욱더 많은 연구가 활기를 띠게 되어 관련한 연구가 쏟아져 나오게 되었고, 기계번역은 가장 먼저 end-to-end 방식을 활용하여 상용화에 성공하였을 뿐만 아니라, Natural Language Processing에 대한 이해도가 더욱 높아지게 되었습니다.

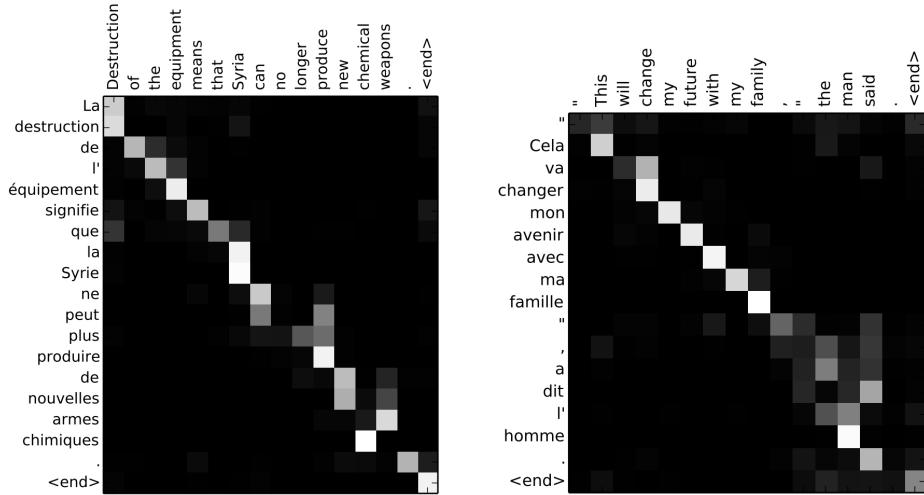


Figure 22:

## Advanced Technique with Memory

Attention이 큰 성공을 거두자, continuous한 방식으로 memory에 access하는 기법에 대한 관심이 커졌습니다. 곧이어 Neural Turing Machine(NTM)[Graves et al.2014]이 대담한 이름대로 큰 파란을 일으키며 주목을 받았습니다. Continuous한 방식으로 memory에서 정보를 read/write하는 방법을 제시하였고, 이어서 Differential Neural Computer (DNC)[Graves et al.2016]가 제시되며 memory 활용방법에 대한 관심이 높아졌습니다.

이러한 memory를 활용하는 기법은 Memory Augmented Neural Network(MANN)이라 불리우며, 이 기법이 발전한다면 최종적으로는 우리가 원하는 정보를 neural network 상에 저장하고 필요할 때 잘 조합하여 꺼내쓰는, Question Answering (QA) task와 같은 문제에 효율적으로 대응 할 수 있게 될 것입니다.

- 참고사이트:
- <https://jamiekang.github.io/2017/05/08/neural-turing-machine>
- <https://sites.google.com/view/mann-emnlp2017/>

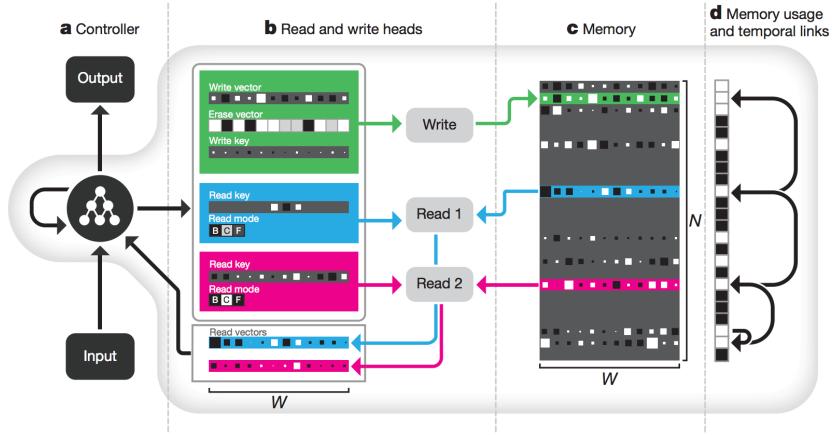


Figure 23:

## Convergence of NLP and Reinforcement Learning

일찌감치 Variational Auto Encoder(VAE)[Kingma et al.2013]와 Generative Adversarial Networks(GAN)[Goodfellow et al.2014]을 통해 Computer Vision 분야는 기존의 discriminative learning 방식을 벗어나 generative learning에 관심이 옮겨간 것과 달리, NLP분야는 그럴 필요가 없었습니다. 이미 language modeling 자체가 문장에 대한 generative learning이기 때문입니다.

하지만, 기계번역의 연구결과서 큰 성과를 띠면서 학계는 다른 어려움에 부딪히게 됩니다. Deep learning에서 사용하는 cross entropy와 실제 기계번역을 위한 objective function과 괴리(discrepancy)가 있었기 때문입니다. 따라서, 마치 Computer Vision에서 기존의 MSE loss의 한계를 벗어나기 위해 GAN을 도입한 것처럼, 기존의 loss function과 다른 무엇인가가 필요하였습니다.

이때 성공적으로 강화학습의 policy gradients 방식을 NLP에 적용함으로써[Bahdanau et al.2016][Yu et al.2016], 마치 vision분야의 adversarial learning을 NLP에서도 흉내낼 수 있게 되었습니다. 이렇게, 강화학습(RL)을 사용하여 실제 task에서의 objective function으로부터 reward를 받을 수 있게 됨에 따라, 더욱 성능을 극대화 할 수 있게 되었습니다.

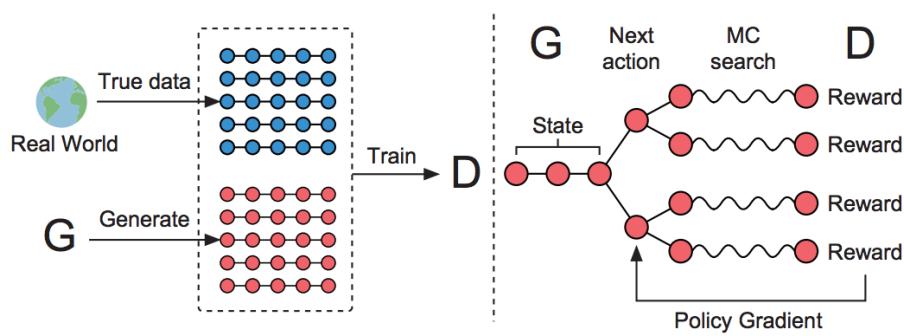


Figure 24:

# Introduction to PyTorch



Figure 1: Geoffrey Hinton – Image from web

## Before Deeplearning using PyTorch

### 장비 구성

당연히 예산에 제한이 없으면 가장 비싼 제품들로만 pick하면 될 수 있습니다. 하지만 현실은 다르기에, 여러가지 고려해야 할 점들을 우선순위별로 나열 해 보겠습니다.

#### CPU

잘 짜여진 PyTorch 코드는 대부분 GPU 사용량을 최대화 합니다. 따라서 보통의 경우 parallel한 연산은 모두 GPU에 넘어가게 되고, 일부 피할 수 없는 sequential한 연산만

남아 CPU의 사용량은 1개 core에 집중되어 100% 내외를 가리키게 됩니다. 따라서, Core의 숫자가 많은 것도 좋지만, 개별 코어의 clock이 높은 것이 더 중요합니다.

물론 본격적인 딥러닝을 하기 이전에 preprocessing 또는 word embedding의 단계에서는 CPU core가 많은 것이 좋을 수 있습니다. 따라서 이러한 작업간의 중요도를 잘 고려하여 CPU를 선택하면 됩니다. 잘 모르겠고 귀찮을 땐 그냥 \_i7-?700K\_를 하면 됩니다.

결론: core 갯수보단 clock이 높아야 한다.

RAM



Figure 2:

메모리 가격이 하늘 높은 줄 모르고 치솟고 있습니다. 이러한 상황에서 원하는 만큼의 메모리를 확보하는 것은 쉬운일이 아닙니다. 하지만 메모리가 최소 16GB에서 권장 32GB는 되어야 합니다. 보통 Xeon CPU가 아닌 메인보드의 경우에는 4개 슬롯에 64GB가 최대치인 경우가 많습니다. 보통 preprocessing 작업을 할 때에 메모리가 많이 필요할 수 있습니다. 따라서 어느정도의 메모리는 시스템 구성에 필요합니다.

결론: 메모리는 많을 수록 좋다. 모르면 32GB

## GPU



딥러닝 연구자들에게 꿈의 머신, Nvidia DGX-1

일반적인 사용자라면 Nvidia GTX 계열의 Graphic Card를 보통 선택하면 됩니다. (모든 deep learning framework은 CUDA와 함께 동작하기 때문에 Radeon 그래픽 카드는 소용이 없습니다.) Cuda Core가 많고, clock이 높을 수록 속도가 빠른것을 의미합니다. memory bandwidth도 매우 중요합니다. 가장 중요한것은 memory size입니다. Memory size가 가장 큰 GPU일수록 고가이기도 합니다. 대략 9세대(maxwell)보다 10세대(pascal)의 경우 1.3배 정도 속도 차이가 납니다.

메모리가 클 수록 더 큰 배치사이즈를 돌릴 수 있습니다. 또한 크고 복잡한 아키텍쳐를 GPU에 올릴 수 있습니다. 크고 복잡한 아키텍쳐인데 메모리 사이즈가 작으면 작은 배치사이즈로 돌려야 하고, 훈련 속도는 역시 느려지게 될 것 입니다. (참고: 배치사이즈가 2배 크다고 훈련이 2배 빠르지는 않습니다.) 또한, 메모리가 넉넉하면 GPU memory에 애초에 모든 데이터를 로드 한 채로 훈련을 실행 시킬 수도 있습니다. 이는 구현의 난이도를 매우 낮추어 줄 것 입니다.

결론: 메모리가 클 수록 좋다. 하지만 메모리가 크면 비싸다.

## Power

파워 서플라이에는 돈을 아끼면 안됩니다. 실제 700W를 뽑아주는 녀석을 장착하면 그래픽 카드 1개 기준으로 부족할 일은 없습니다. (단, 중급 그래픽 카드의 경우에는 600W도 가능)

결론: 700W. 뺑파워는 안됩니다.

## Cooling System

GPU에서 뿜어져 나오는 열이 엄청나기 때문에 쿨링 시스템이 매우 중요합니다. 따라서 케이스의 선택도 중요하고, 내부에 fan을 설치하는 것도 좋습니다.

## How to Install

Linux (Ubuntu) 기준으로 PyTorch를 설치하고 실행하는 것을 살펴 보도록 하겠습니다.

### Anaconda

대부분 linux를 설치하면 기본적으로 python이 설치되어 있는 것을 볼 수 있습니다. 대부분의 경우 아래와 같은 경로에 설치되어 있습니다.

```
$ sudo which python  
/usr/bin/python
```

이 경우에는 시스템 전체 사용자들이 공통으로 사용하는 python이기 때문에 anaconda를 설치하여 해당 사용자만을 위한 python을 설치하고, 그 위에 여러 package를 자유롭게 install 또는 uninstall 하는 것이 편리합니다. 또한, 경우에 따라 발생할 수 있는 권한 문제에서도 훨씬 자유롭습니다. 따라서 Anaconda를 사용하는 것을 권장합니다. Anaconda는 아래의 주소에서 다운로드 받을 수 있습니다.

<https://www.anaconda.com/download/#linux>

또한 많은 package가 기본으로 설치되는 anaconda와 달리 Miniconda를 설치하여 훨씬 더 가볍게 사용할 수도 있습니다.

### 2.7 vs 3.6

Python을 처음 접하는 많은 사용자들이 2.7과 3.6 사이에서 어떤 것을 택해야 할지 고민하게 됩니다. 처음 python을 접하는 사람들은 3.6을 택하는 것을 추천합니다. 특히, 이 책에서 다루는 NLP와 관련된 text encoding의 default가 UTF-8로 되어 있어서 훨씬 더 편리하게 사용할 수 있습니다. 따라서, 시간을 아끼기 위해서는 3.6으로 시작할 것을 권장합니다. 다만, 2.7에서 작성된 코드를 3.6에서 사용하기

위해서는 코드를 약간 수정해야 할 필요성이 있습니다. (참고로, 대부분의 경우 3.6에서 작성한 코드는 2.7에서 잘 돌아갈 가능성이 훨씬 더 높습니다.)

## 왜 PyTorch 인가?



Figure 3:

Tensorflow를 개발한 Google에 맞서, PyTorch는 Facebook의 주도하에 개발이 진행되고 있습니다. 자체 딥러닝 전용 H/W인 TPU를 가지고 있어 상대적으로 Nvidia GPU에서 보다 자유로운 Google과 달리, PyTorch는 Nvidia도 참여한 project이기 때문에 Nvidia의 CUDA GPU에 더욱 최적화 되어 있습니다. 실제로도, Nvidia에서도 적극 PyTorch를 권장하는 모습이며, 특히 NLP 분야에서는 Tensorflow에 비하여 적극 권장하기도 합니다.

일찌감치 Tensorflow를 내세운 Google과 달리, PyTorch는 그에비해 훨씬 뒤늦게 deep learning framework 개발에 뛰어들었기 때문에, 상대적으로 훨씬 적은 유저풀을 갖고 있습니다.

하지만, PyTorch가 가진 장점과 매력 때문에, 산업계보다는 학계에서 적극적으로 PyTorch의 사용을 늘려가고 있는 추세이며, 이러한 트렌드는 산업계에도 점점 퍼져나가고 있습니다. 따라서, Tensorflow는 paper를 구현한 수많은 github source code와 pretrained model parameter가 있는 것이 장점이긴 하지만, PyTorch도 빠르게 따라잡고 있는 추세입니다. – 하지만 아직은 Tensorflow의 아성을 넘기에는 부족합니다.

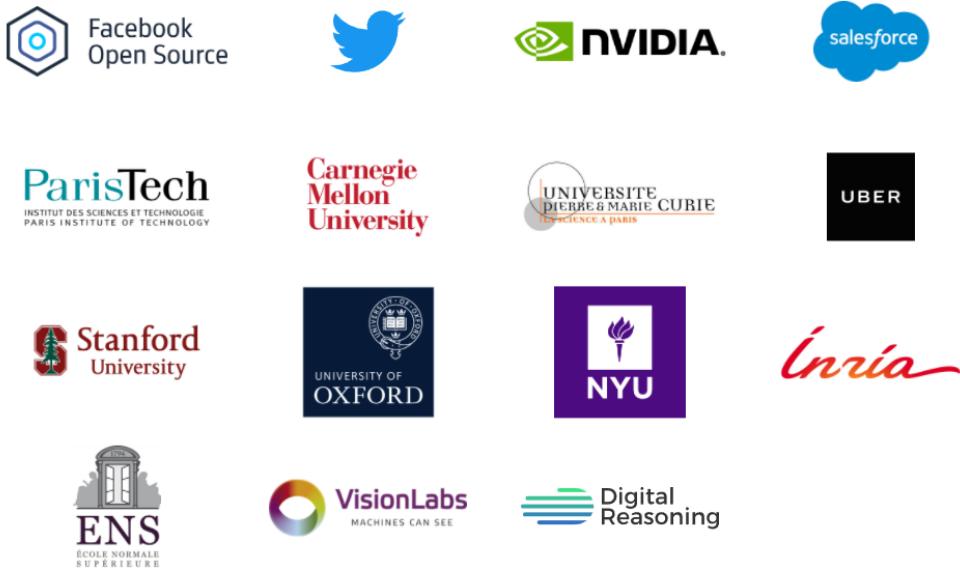


Figure 4:

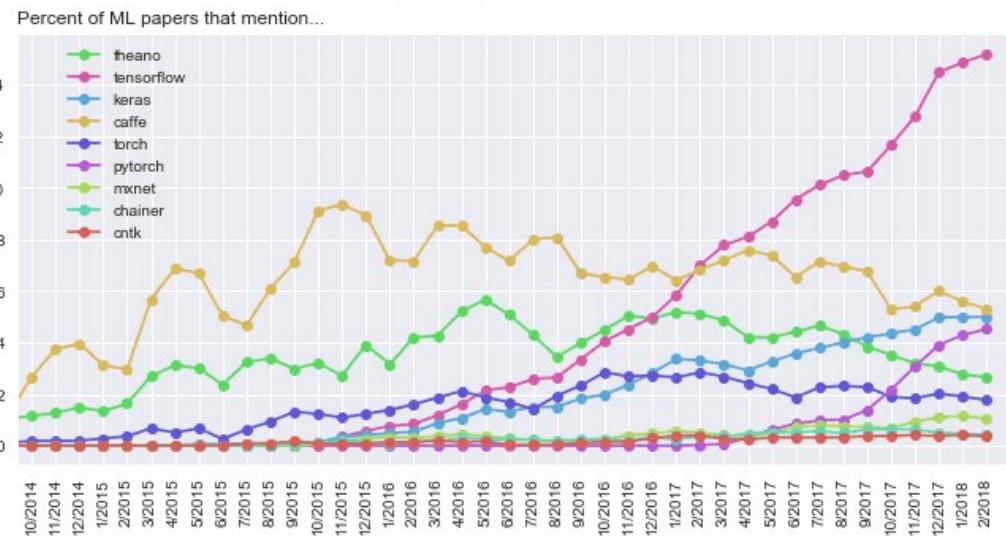


Figure 5: Plot of how some of the more popular frameworks evolved over time.  
Image from Karpathy's medium



Andrej Karpathy ✅  
@karpathy

팔로우



I've been using PyTorch a few months now  
and I've never felt better. I have more energy.  
My skin is clearer. My eye sight has  
improved.

오전 11:56 - 2017년 5월 26일

384 리트윗 1,499 마음에 들어요



33

384

1,499

[Image from Karpathy's twitter]

Tesla의 AI 수장인 Karpathy는 자신의 트위터에서 파이토치를 찬양하였습니다. 그럼 무엇이 그를 찬양하도록 만들었는지 좀 더 알아보도록 하겠습니다. PyTorch는 major deep learning framework 중에서 가장 늦게 나온 편인 만큼, 그동안 여러 framework의 장점을 모두 갖고 있습니다.

- Python First, 깔끔한 코드
- 먼저 Tensorflow와 달리 Python First를 표방한 PyTorch는 tensor 연산과 같이 속도에 크리티컬 한 부분을 제외하고는 대부분의 모듈이 python으로 짜여 있습니다. 따라서 코드가 깔끔합니다.
- NumPy/SciPy과 뛰어난 호환성
- Theano의 장점인 NumPy와의 호환성이 PyTorch에도 그대로 들어왔습니다. 따라서 기존 numpy를 사용하던 사용자들은 처음 파이토치를 접하더라도 큰 위화감 없이 그대로 적용할 수 있습니다.
- Autograd
- 단지 값을 앞으로 전달(feed-forward)시키며 차례차례 계산 한 것일 뿐인데, backward() 호출 한번에 gradient를 구할 수 있습니다.
- Dynamic Graph
- Tensorflow의 경우 session이라는 개념이 있어서, session이 시작되면 model architecture 등의 graph 구조의 수정이 어려웠습니다. 하지만, PyTorch는 그러한 개념이 없어 편리하게 사용 할 수 있습니다.

# PyTorch Short Tutorial

## Tensor

PyTorch의 tensor는 numpy의 array와 같은 개념입니다. PyTorch 상에서 연산을 수행하기 위한 가장 기본적인 객체로써, 앞으로 우리가 수행할 모든 연산은 이 객체를 통하여 됩니다. 따라서 PyTorch는 tensor를 통해 값을 저장하고 그 값들에 대해서 연산을 수행할 수 있는 함수를 제공합니다.

아래의 예제는 같은 동작을 수행하는 PyTorch 코드와 NumPy 코드 입니다.

```
import torch

x = torch.Tensor(2, 2)
x = torch.Tensor([[1, 2], [3, 4]])
x = torch.from_numpy(x)

import numpy as np

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

보시다시피, PyTorch는 굉장히 NumPy와 비슷한 방식의 코딩 스타일을 갖고 있고, 따라서 코드를 보고 해석하거나 새롭게 작성함에 있어서 굉장히 수월합니다.

Tensor는 아래와 같이 다양한 자료형을 제공 합니다.

Data	CPU	GPU
type	dtype	tensor
32-	<code>torch.float</code>	<code>FloatTensor</code>
bit	or	
float-	<code>torch.float</code>	
ing		
point		
64-	<code>torch.double</code>	<code>DoubleTensor</code>
bit	or	
float-	<code>torch.double</code>	
ing		
point		

Data type	CPU dtype	GPU tensor
16-bit float	<code>torch.float16Tensor</code>	<code>HalfTensor</code>
8-bit unsigned integer	<code>torch.uint8Tensor</code>	<code>ByteTensor</code>
8-bit signed integer	<code>torch.int8Tensor</code>	<code>CharTensor</code>
16-bit signed integer	<code>torch.int16Tensor</code>	<code>ShortTensor</code>
32-bit signed integer	<code>torch.int32Tensor</code>	<code>IntTensor</code>

	Data type	CPU dtype	GPU tensor
64-bit	<code>torch.int64</code>	<code>torch.LongTensor</code>	<code>LongTensor</code>
in-place	<code>or</code>		
tensor	<code>torch.long</code>		
gradient	<code>(signed)</code>		

`torch.Tensor`를 통해 선언하게 되면 디폴트 타입인 `torch.FloatTensor`로 선언하는 것과 같습니다.

좀 더 자세한 참고를 원한다면 PyTorch docs를 방문하시면 됩니다.

## Autograd

PyTorch는 자동으로 미분 및 back-propagation을 해주는 Autograd 기능을 갖고 있습니다. 따라서 우리는 대부분의 tensor간의 연산들을 크게 신경 쓸 필요 없이 수행하고, back-propagation을 수행하는 명령어를 호출 해 주기만 하면 됩니다.

이를 위해서, PyTorch는 tensor들 사이의 연산을 할 때마다 computational graph를 생성하여 연산의 결과물이 어떤 tensor로부터 어떤 연산을 통해서 왔는지 추적하고 있습니다. 따라서 우리가 최종적으로 나온 스칼라(scalar)에 미분과 back-propagation(역전파)을 수행하도록 하였을 때, 자동으로 각 tensor 별로 자기 자신의 자식노드(child node)에 해당하는 tensor를 찾아서 계속해서 back-propagation 할 수 있도록 합니다.

```
import torch

x = torch.FloatTensor(2, 2)
y = torch.FloatTensor(2, 2)
y.requires_grad_(True)

z = (x + y) + torch.FloatTensor(2, 2)
```

위의 예제에서처럼  $x$ 와  $y$ 를 생성하고 둘을 더하는 연산을 수행하면  $x + y$ , 이에 해당하는 tensor가 생성되어 computational graph에 할당 됩니다. 그리고 다시 생성

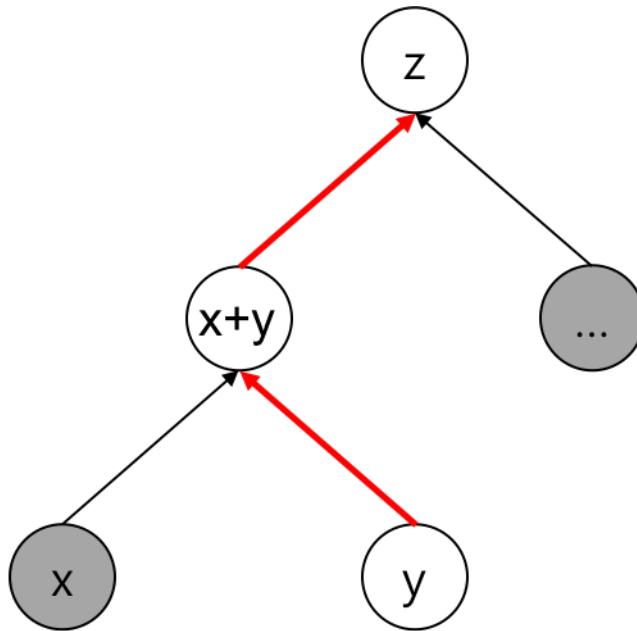


Figure 6:

된  $2 \times 2$  tensor를 더해준 뒤, 이를  $z$ 에 assign(할당) 하게 됩니다. 따라서  $z$ 로부터 back-propagation을 수행하게 되면, 이미 생성된 computational graph를 따라서 gradient를 전달 할 수 있게 됩니다.

Gradient를 구할 필요가 없는 연산의 경우에는 아래와 같이 with 문법을 사용하여 연산을 수행할 수 있습니다. back-propagation이 필요 없는 추론(inference) 등을 수행 할 때 유용하며, gradient를 구하기 위한 사전 작업들(computational graph 생성)을 생략할 수 있기 때문에, 연산 속도 및 메모리 사용에 있어서도 큰 이점을 지니게 됩니다.

```

import torch

with torch.no_grad():
    x = torch.FloatTensor(2, 2)
    y = torch.FloatTensor(2, 2)
    y.requires_grad_(True)

    z = (x + y) + torch.FloatTensor(2, 2)

```

## How to Do Basic Operations (Forward)

이번에는 Linear Layer(또는 fully-connected layer, dense layer)를 구현 해 보도록 하겠습니다. M by N의 입력 matrix가 주어지면, N by P의 matrix를 곱한 후, P size의 vector를 bias로 더하도록 하겠습니다. 수식은 아래와 같을 것 입니다.

$$y = xW^t + b$$

사실 이 수식에서  $x$ 는 vector이지만, 보통 우리는 딥러닝을 수행 할 때에 mini-batch 기준으로 수행하므로,  $x$ 가 matrix라고 가정 하겠습니다.

이를 좀 더 구현하기 쉽게 아래와 같이 표현 해 볼 수도 있습니다.

$$y = f(x; \theta) \text{ where } \theta = \{W, b\}$$

이러한 linear layer의 기능은 아래와 같이 PyTorch로 구현할 수 있습니다.

```
import torch

def linear(x, W, b):
    y = torch.mm(x, W) + b

    return y

x = torch.FloatTensor(16, 10)
W = torch.FloatTensor(10, 5)
b = torch.FloatTensor(5)

y = linear(x, W, b)
```

## Broadcasting

Broadcasting에 대해서 설명 해 보겠습니다. 역시 NumPy에서 제공되는 broadcasting과 동일하게 동작합니다. `matmul()`을 사용하면 임의의 차원의 tensor끼리 연산을 가능하게 해 줍니다. 이전에는 강제로 2차원을 만들거나 하여 곱해주는 수 밖에 없었습니다. 다만, 입력으로 주어지는 tensor들의 차원에 따라서 규칙이 적용됩니다. 그 규칙은 아래와 같습니다.

```

>>> # vector x vector
>>> tensor1 = torch.randn(3)
>>> tensor2 = torch.randn(3)
>>> torch.matmul(tensor1, tensor2).size()

-0.4334
[torch.FloatTensor of size ()]

>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([3])
>>> # batched matrix x broadcasted vector
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3])
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
>>> # batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])

```

마찬가지로 몇 셤 연산에 대해서도 broadcasting이 적용될 수 있는데 그 규칙은 아래와 같습니다. 곱셈에 비해서 좀 더 규칙이 복잡하니 주의해야 합니다.

```

>>> x=torch.FloatTensor(5,7,3)
>>> y=torch.FloatTensor(5,7,3)
# same shapes are always broadcastable (i.e. the above rules always hold)

>>> x=torch.FloatTensor()
>>> y=torch.FloatTensor(2,2)
# x and y are not broadcastable, because x does not have at least 1 dimension

```

```

# can line up trailing dimensions
>>> x=torch.FloatTensor(5,3,4,1)
>>> y=torch.FloatTensor( 3,1,1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist

# but:
>>> x=torch.FloatTensor(5,2,4,1)
>>> y=torch.FloatTensor( 3,1,1)
# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3

# can line up trailing dimensions to make reading easier
>>> x=torch.FloatTensor(5,1,4,1)
>>> y=torch.FloatTensor( 3,1,1)
>>> (x+y).size()
torch.Size([5, 3, 4, 1])

# but not necessary:
>>> x=torch.FloatTensor(1)
>>> y=torch.FloatTensor(3,1,7)
>>> (x+y).size()
torch.Size([3, 1, 7])

>>> x=torch.FloatTensor(5,2,4,1)
>>> y=torch.FloatTensor(3,1,1)
>>> (x+y).size()
RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-

```

Broadcasting 연산의 가장 주의해야 할 점은, 의도하지 않은 broadcasting연산으로 인해서 예상치 못한 버그가 발생할 가능성입니다. 원래는 같은 크기의 tensor끼리 연산을 해야 하는 부분인데, 실수에 의해서 다른 크기가 되었을 때, 원래대로라면 덧셈 또는 곱셈을 하고 runtime error가 나서 알아차렸겠지만, broadcasting으로 인해서 runtime error가 나지 않고 의도치 않은 연산을 통해 프로그램이 정상적으로 종료 될 수 있습니다. 하지만 실행 결과로는 결국 기대하던 값과 다른 값이 나오게 되어, 이에

대한 원인을 찾느라 고생할 수도 있습니다. 따라서 주의가 필요합니다.

참고사이트: - <http://pytorch.org/docs/master/torch.html?highlight=matmul#torch.matmul> - <http://pytorch.org/docs/master/notes/broadcasting.html#broadcasting-semantics>

## nn.Module

이제까지 우리가 원하는 수식을 어떻게 어떻게 feed-forward 구현 하는지 살펴보았습니다. 이것을 좀 더 편리하고 깔끔하게 사용하는 방법에 대해서 다루어 보도록 하겠습니다. PyTorch는 nn.Module이라는 class를 제공하여 사용자가 이 위에서 자신이 필요로 하는 model architecture를 구현할 수 있도록 하였습니다.

nn.Module의 상속한 사용자 정의 class는 다시 내부에 nn.Module을 상속한 class를 선언하여 소유 할 수 있습니다. 즉, nn.Module 안에 nn.Module 객체를 선언하여 사용 할 수 있습니다. 그리고 nn.Module의 forward() 함수를 override하여 feed-forward를 구현할 수 있습니다. 이외에도 nn.Module의 특성을 이용하여 한번에 weight parameter를 save/load 할 수도 있습니다.

그럼 앞서 구현한 linear 함수 대신에 MyLinear라는 class를 nn.Module을 상속받아 선언하고, 사용하여 똑같은 기능을 구현 해 보겠습니다.

```
import torch
import torch.nn as nn

class MyLinear(nn.Module):

    def __init__(self, input_size, output_size):
        super(MyLinear, self).__init__()

        self.W = torch.FloatTensor(input_size, output_size)
        self.b = torch.FloatTensor(output_size)

    def forward(self, x):
        y = torch.mm(x, self.W) + self.b

    return y
```

위와 같이 선언한 MyLinear class를 이제 직접 사용해서 정상 동작 하는지 확인 해

보겠습니다.

```
x = torch.FloatTensor(16, 10)
linear = MyLinear(10, 5)
y = linear(x)
```

forward()에서 정의 해 준대로 잘 동작 하는 것을 볼 수 있습니다. 하지만, 위와 같이 W와 b를 선언하면 문제점이 있습니다. parameters() 함수는 module 내에 선언 된 learnable parameter들을 iterative하게 주는 iterator를 반환하는 함수입니다. 한번, linear module 내의 learnable parameter들의 크기를 size()함수를 통해 확인 해 보도록 하겠습니다.

```
>>> params = [p.size() for p in linear.parameters()]
>>> print(params)
[]
```

아무것도 들어있지 않은 빈 list가 찍혔습니다. 즉, linear module 내에는 learnable parameter가 없다는 이야기입니다. 아래의 웹페이지에 그 이유가 자세히 나와 있습니다.

참고사이트: <http://pytorch.org/docs/master/nn.html?highlight=parameter#parameters>

A kind of Tensor that is to be considered a module parameter. Parameters are Tensor subclasses, that have a very special property when used with Module s – when they’re assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in parameters() iterator. Assigning a Tensor doesn’t have such effect. This is because one might want to cache some temporary state, like last hidden state of the RNN, in the model. If there was no such class as Parameter, these temporaries would get registered too.

따라서 우리는 Parameter라는 class를 사용하여 tensor를 wrapping해야 합니다. 그럼 아래와 같이 될 것 입니다.

```
class MyLinear(nn.Module):

    def __init__(self, input_size, output_size):
        super(MyLinear, self).__init__()

        self.W = nn.Parameter(torch.FloatTensor(input_size, output_size), requires_grad=True)
        self.b = nn.Parameter(torch.FloatTensor(output_size), requires_grad=True)
```

```

def forward(self, x):
    y = torch.mm(x, self.W) + self.b

    return y

```

그럼 아까와 같이 다시 linear module 내부의 learnable parameter들의 size를 확인해 보도록 하겠습니다.

```

>>> params = [p.size() for p in linear.parameters()]
>>> print(params)
[torch.Size([10, 5]), torch.Size([5])]

```

잘 들어있는 것을 확인 할 수 있습니다. 그럼 깔끔하게 바꾸어 보도록 하겠습니다.  
아래와 같이 바꾸면 제대로 된 구현이라고 볼 수 있습니다.

```

class MyLinear(nn.Module):

    def __init__(self, input_size, output_size):
        super(MyLinear, self).__init__()

        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        y = self.linear(x)

        return y

```

nn.Linear class를 사용하여 W와 b를 대체하였습니다. 그리고 아래와 같이 print를 해 보면 내부의 Linear Layer가 잘 찍혀 나오는 것을 확인 할 수 있습니다.

```

>>> print(linear)
MyLinear(
    (linear): Linear(in_features=10, out_features=5, bias=True)
)

```

## Backward (Back-propagation)

이제까지 원하는 연산을 통해 값을 앞으로 전달(feed-forward)하는 방법을 살펴보았습니다. 이제 이렇게 얻은 값을 우리가 원하는 값과의 차이를 계산하여 error를 뒤로 전달(back-propagation)하는 것을 해 보도록 하겠습니다.

예를 들어 우리가 원하는 값은 아래와 같이 100이라고 하였을 때, linear의 결과값 matrix의 합과 목표값과의 거리(error 또는 loss)를 구하고, 그 값에 대해서 backward()함수를 사용함으로써 gradient를 구합니다. 이때, error는 scalar로 표현 되어야 합니다. vector나 matrix의 형태여서는 안됩니다.

```
objective = 100

x = torch.FloatTensor(16, 10)
linear = MyLinear(10, 5)
y = linear(x)
loss = (objective - y.sum())**2

loss.backward()
```

위와 같이 구해진 각 parameter들의 gradient에 대해서 gradient descent 방법을 사용하여 error(loss)를 줄여나갈 수 있을 것입니다.

## train() and eval()

```
# Training...
linear.eval()
# Do some inference process.
linear.train()
# Restart training, again.
```

위와 같이 PyTorch는 train()과 eval() 함수를 제공하여 사용자가 필요에 따라 model에 대해서 훈련시와 추론시의 모드 전환을 쉽게 할 수 있도록 합니다. nn.Module을 상속받아 구현하고 생성한 객체는 기본적으로 training mode로 되어 있는데, eval()을 사용하여 module로 하여금 inference mode로 바꾸어주게 되면, (gradient를 계산하지 않도록 함으로써) inference 속도 뿐만 아니라, dropout 또는 batch-normalization과 같은 training과 inference 시에 다른 forward() 동작을 하는 module들에 대해서 각기 때에 따라 올바른 동작을 하도록 합니다. 다만, inference가 끝나면 다시 train()을 선언 해 주어, 원래의 훈련모드로 돌아가게 해 주어야 합니다.

## Example

이제까지 배운 것들을 활용하여 임의의 함수를 근사(approximate)하는 신경망을 구현해 보도록 하겠습니다.

1. Random(임의)으로 생성한 tensor들을
2. 우리가 근사하고자 하는 정답 함수에 넣어 정답을 구하고,
3. 그 정답( $y$ )과 신경망을 통과한  $\hat{y}$ 과의 차이(error)를 Mean Square Error(MSE)를 통해 구하여
4. SGD를 통해서 최적화(optimize)하도록 해 보겠습니다.

MSE의 수식은 아래와 같습니다.

$$\mathcal{L}_{MSE}(x, y) = \frac{1}{N} \sum_{i=1}^N (x_n - y_n)^2$$

먼저 1개의 linear layer를 가진 MyModel이라는 모듈(module)을 선언합니다.

```
import random

import torch
import torch.nn as nn

class MyModel(nn.Module):

    def __init__(self, input_size, output_size):
        super(MyModel, self).__init__()

        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        y = self.linear(x)

    return y
```

그리고 아래와 같이, 임의의 함수가 동작한다고 가정하겠습니다.

$$f(x_1, x_2, x_3) = 3x_1 + x_2 - 2x_3$$

해당 함수를 python으로 구현하면 아래와 같습니다. 물론 신경망 입장에서는 내부 동작 내용을 알 수 없는 함수입니다.

```
def ground_truth(x):
    return 3 * x[:, 0] + x[:, 1] - 2 * x[:, 2]
```

아래는 입력을 받아 feed-forward 시킨 후, back-propagation하여 gradient descent까지 하는 함수입니다.

```
def train(model, x, y, optim):
    # initialize gradients in all parameters in module.
    optim.zero_grad()

    # feed-forward
    y_hat = model(x)
    # get error between answer and inferenced.
    loss = ((y - y_hat)**2).sum() / x.size(0)

    # back-propagation
    loss.backward()

    # one-step of gradient descent
    optim.step()

    return loss.data
```

그럼 위의 함수들을 사용하기 위해서 하이퍼 파라미터(hyper-parameter)를 설정하겠습니다.

```
batch_size = 1
n_epochs = 1000
n_iter = 10000

model = MyModel(3, 1)
optim = torch.optim.SGD(model.parameters(), lr = 0.0001, momentum=0.1)

print(model)
```

위의 값을 사용하여 평균 손실(loss) 값이 .001보다 작을 때 까지 훈련 시킵니다.

```
for epoch in range(n_epochs):
```

```

avg_loss = 0

for i in range(n_iter):
    x = torch.rand(batch_size, 3)
    y = ground_truth(x.data)

    loss = train(model, x, y, optim)

    avg_loss += loss
avg_loss = avg_loss / n_iter

# simple test sample to check the network.
x_valid = torch.FloatTensor([[.3, .2, .1]])
y_valid = ground_truth(x_valid.data)

model.eval()
y_hat = model(x_valid)
model.train()

print(avg_loss, y_valid.data[0], y_hat.data[0, 0])

if avg_loss < .001: # finish the training if the loss is smaller than .001.
    break

```

위와 같이 임의의 함수에 대해서 실제로 신경망을 근사(approximate)하는 아주 간단한 예제를 살펴 보았습니다. 사실은 신경망이라기보단, 선형 회귀(linear regression) 함수라고 봐야 합니다. 하지만, 앞으로 책에서 다루어질 구조(architecture)들과 훈련 방법들도 이 예제의 연장선상에 지나지 않습니다.

이제까지 다룬 내용을 바탕으로 PyTorch상에서 딥러닝을 수행하는 과정은 아래와 같이 요약 해 볼 수 있습니다.

1. nn.Module 클래스를 상속받아 Model 아키텍쳐 선언(forward함수를 통해)
2. Model 객체 생성
3. SGD나 Adam등의 Optimizer를 생성하고, Model의 parameter를 등록
4. 데이터로 미니배치를 구성하여 feed-forward → computation graph 생성
5. 손실함수(loss function)를 통해 최종 결과값(scalar) loss를 계산
6. 손실(loss)에 대해서 backward() 호출 → computation graph 상의 tensor들에 gradient가 채워짐

- 3번의 optimizer에서 step()을 호출하여 gradient descent 1-step 수행

## Using GPU

PyTorch는 당연히 GPU상에서 훈련하는 방법도 제공합니다. 아래와 같이 cuda()함수를 통해서 원하는 객체를 GPU memory상으로 copy(Tensor의 경우)하거나 move(nn.Module의 하위 클래스인 경우) 시킬 수 있습니다.

```
>>> # Note that tensor is declared in torch.cuda.  
>>> x = torch.cuda.FloatTensor(16, 10)  
>>> linear = MyLinear(10, 5)  
>>> # .cuda() let module move to GPU memory.  
>>> linear.cuda()  
>>> y = linear(x)
```

또한, cpu()함수를 통해서 다시 PC의 memory로 copy하거나 move할 수 있습니다.

## Word Senses: Similarity and Ambiguity



Figure 1: Philip Resnik

### Word Sense from Thesaurus

이전 섹션에서, 단어는 내부에 의미를 지니고 있고 그 의미는 개념과 같아서 계층적 구조를 지닌다고 하였습니다. 만약 그 계층구조를 잘 분석하고 분류하여 데이터베이스로 구축한다면, 우리가 자연어처리를 하고자 할 때 매우 큰 도움이 될 것입니다. 이런 용도로 구축된 데이터베이스를 어휘분류사전(thesaurus)라고 부릅니다. 이번 섹션에서는 thesaurus의 대표인 WordNet에 대해 다루어 보겠습니다.

### WordNet

WordNet(워드넷)은 1985년부터 심리학 교수인 George Armitage Miller 교수의 지도하에 프린스턴 대학에서 만든 프로그램입니다. 처음에는 주로

기계번역(Machine Translation)을 돋기 위한 목적으로 만들어졌으며, 따라서 동의어 집합(Synset) 또는 상위어(Hypernym)나 하위어(Hyponym)에 대한 정보가 특히 잘 구축되어 있는 것이 장점입니다. 단어에 대한 상위어와 하위어 정보를 구축하게 됨으로써, Directed Acyclic Graph(유방향 비순환 그래프)를 이루게 됩니다. (Tree구조가 아닌 이유는 하나의 노드가 여러 상위 노드를 가질 수 있기 때문입니다.)

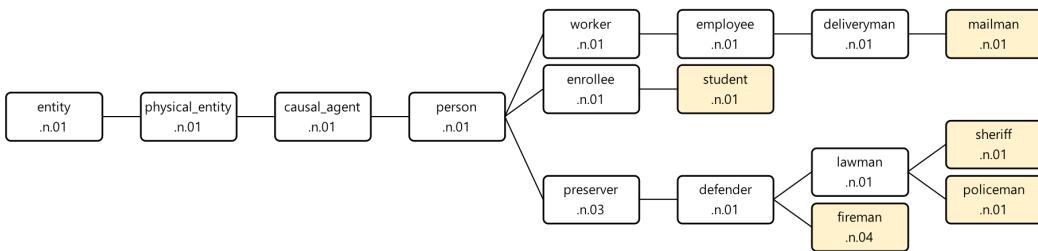


Figure 2: 각 단어별 top-1 sense의 top-1 hypernym만 선택하여 tree로 나타낸 경우

WordNet은 프로그램으로 제공되므로 다운로드 받아 설치할 수도 있고, 웹사이트에서 바로 이용 할 수도 있습니다. 또한, NLTK에 랩핑(wrapping)되어 포함되어 있어, import하여 사용 가능합니다. 아래는 WordNet 웹사이트에서 bank를 검색한 결과입니다.

그림에서와 같이 bank라는 단어에 대해서 명사(noun)일때의 의미 10개, 동사(verb)인 경우의 의미 8개를 정의 해 놓았습니다. 명사 bank#2의 경우에는 여러 다른 표현(depository financial institution#1, banking concern#1)들도 같이 게시되어 있는데, 이것은

이처럼 WordNet은 단어 별 여러가지 가능한 의미를 미리 정의 하고, numbering 해 놓았습니다. 또한 각 의미별로 비슷한 의미를 갖는 동의어(Synonym)를 링크 해 놓아, Synset을 제공합니다. 이것은 단어 중의성 해소에 있어서 매우 좋은 labeled data가 될 수 있습니다. 만약 WordNet이 없다면, 각 단어별로 몇 개의 의미가 있는지 조차 알 수가 없을 것입니다. 즉, WordNet 덕분에 우리는 이 데이터를 바탕으로 supervised learning(지도 학습)을 통해 단어 중의성 해소 문제를 풀 수 있습니다.

## WordNet Search - 3.1

- [WordNet home page](#) - [Glossary](#) - [Help](#)

Word to search for:

Display Options:

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations

Display options for sense: (gloss) "an example sentence"

Display options for word: word#sense number

### Noun

- S: (n) bank#1 (sloping land (especially the slope beside a body of water))  
*"they pulled the canoe up on the bank"; "he sat on the bank of the river and watched the currents"*
- S: (n) depository financial institution#1, bank#2, banking concern#1, banking company#1 (a financial institution that accepts deposits and channels the money into lending activities) *"he cashed a check at the bank"; "that bank holds the mortgage on my home"*
- S: (n) bank#3 (a long ridge or pile) *"a huge bank of earth"*
- S: (n) bank#4 (an arrangement of similar objects in a row or in tiers) *"he operated a bank of switches"*
- S: (n) bank#5 (a supply or stock held in reserve for future use (especially in emergencies))
- S: (n) bank#6 (the funds held by a gambling house or the dealer in some gambling games) *"he tried to break the bank at Monte Carlo"*
- S: (n) bank#7, cant#2, camber#2 (a slope in the turn of a road or track; the outside is higher than the inside in order to reduce the effects of centrifugal force)
- S: (n) savings bank#2, coin bank#1, money box#1, bank#8 (a container (usually with a slot in the top) for keeping money at home) *"the coin bank was empty"*
- S: (n) bank#9, bank building#1 (a building in which the business of banking transacted) *"the bank is on the corner of Nassau and Witherspoon"*
- S: (n) bank#10 (a flight maneuver; aircraft tips laterally about its longitudinal axis (especially in turning)) *"the plane went into a steep bank"*

### Verb

- S: (v) bank#1 (tip laterally) *"the pilot had to bank the aircraft"*
- S: (v) bank#2 (enclose with a bank) *"bank roads"*
- S: (v) bank#3 (do business with a bank or keep an account at a bank) *"Where do you bank in this town?"*
- S: (v) bank#4 (act as the banker in a game or in gambling)
- S: (v) bank#5 (be in the banking business)
- S: (v) deposit#2, bank#6 (put into a bank account) *"She deposits her paycheck every month"*
- S: (v) bank#7 (cover with ashes so to control the rate of burning) *"bank a fire"*
- S: (v) count#8, bet#3, depend#2, swear#5, rely#1, bank#8, look#10, calculate#6, reckon#5 (have faith or confidence in) *"you can count on me to help you any time"; "Look to your friends for support"; "You can bet on that!"; "Depend on your family in times of crisis"*

Figure 3: WordNet 웹사이트에서 단어 'bank'를 검색 한 결과

## 한국어 WordNet

다행히 영어 WordNet과 같이 한국어를 위한 WordNet도 존재 합니다. 다만, 아직까지 표준이라고 할만큼 정해진 것은 없고 몇 개의 WordNet이 존재하고 있습니다. 아직까지 지속적으로 발전하고 있는 만큼, 작업에 따라 필요한 한국어 WordNet을 이용하면 좋습니다.

이름	기관	웹사이트
KorLex	부산대학교	<a href="http://korlex.pusan.ac.kr/">http://korlex.pusan.ac.kr/</a>
Korean WordNet(KWN)	KAIST	<a href="http://wordnet kaist.ac.kr/">http://wordnet kaist.ac.kr/</a>

## Word Similarity Based on Path in WordNet

```
from nltk.corpus import wordnet as wn

def get_hyponyms(synset):
    current_node = synset
    while True:
        print(current_node)
        hypernym = current_node.hypernyms()
        if len(hypernym) == 0:
            break
        current_node = hypernym[0]
```

위의 코드를 사용하면 WordNet에서 특정 단어의 최상위 부모 노드까지의 경로를 구할 수 있습니다. 아래와 같이 'policeman'은 'firefighter', 'sheriff'와 매우 비슷한 경로를 가짐을 알 수 있습니다. 'student'와도 매우 비슷하지만, 'mailman'과 좀 더 비슷함을 알 수 있습니다.

```
>>> get_hyponyms(wn.synsets('policeman')[0])
Synset('policeman.n.01')
Synset('lawman.n.01')
Synset('defender.n.01')
Synset('preserver.n.03')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
```

```

Synset('entity.n.01')

>>> get_hypernyms(wn.synsets('firefighter')[0])
Synset('fireman.n.04')
Synset('defender.n.01')
Synset('preserver.n.03')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')

>>> get_hypernyms(wn.synsets('sheriff')[0])
Synset('sheriff.n.01')
Synset('lawman.n.01')
Synset('defender.n.01')
Synset('preserver.n.03')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')

>>> get_hypernyms(wn.synsets('mailman')[0])
Synset('mailman.n.01')
Synset('deliveryman.n.01')
Synset('employee.n.01')
Synset('worker.n.01')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')

>>> get_hypernyms(wn.synsets('student')[0])
Synset('student.n.01')
Synset('enrollee.n.01')
Synset('person.n.01')
Synset('causal_agent.n.01')
Synset('physical_entity.n.01')
Synset('entity.n.01')

```

위로 부터 얻어낸 정보들을 취합하여 그래프로 나타내면 아래와 같습니다. 그림에서

각 leaf 노드들은 코드에서 쿼리로 주어진 단어들이 됩니다.

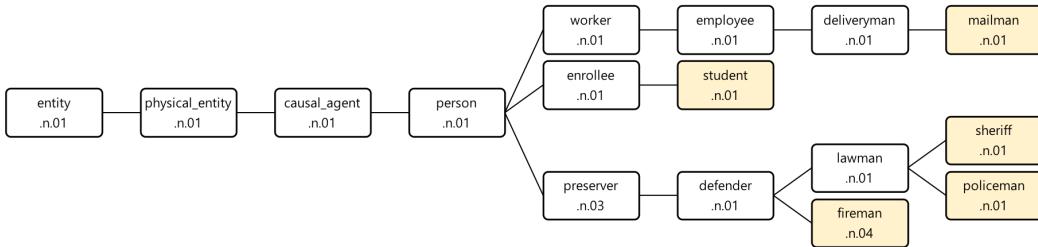


Figure 4: 각 단어들의 쿼리 결과 구조도

이때 각 노드들간에 거리를 우리는 구할 수 있습니다. 아래의 그림에 따르면 student에서 fireman으로 가는 최단거리에는 enrollee, person, preserver, defender 노드들이 위치하고 있습니다. 따라서 student와 fireman의 거리는 5임을 알 수 있습니다.

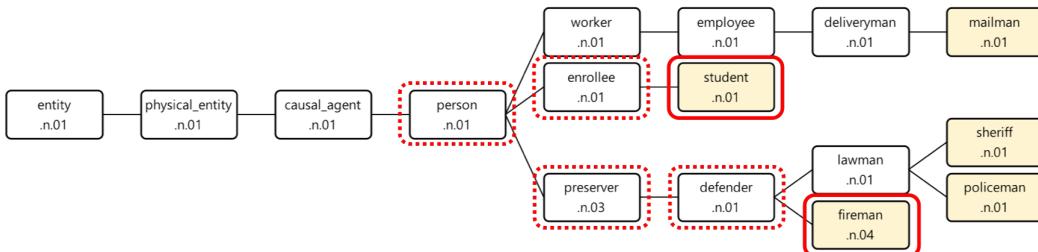


Figure 5: student와 fireman 사이에 위치한 노드들(점선)

이처럼 우리는 각 leaf 노드 간의 최단 거리를 알 수 있고, 이것을 유사도로 치환하여 활용할 수 있습니다. 당연히 거리가 멀수록 단어간의 유사도는 떨어질테니, 아래와 같은 공식을 적용 해 볼 수 있습니다.

$$\text{similarity}(w, w') = - \log \text{distance}(w, w')$$

위와 같이 사전(thesaurus) 기반의 정보를 활용하여 단어간의 유사도를 구할 수 있습니다. 하지만 사전을 구축하는데는 너무 많은 비용과 시간이 소요 됩니다. 또한, 아무 사전이나 되는 것이 아닌, hypernym과 hyponym이 잘 반영되어 있어야 할 것입니다. 이처럼, 사전에 기반한 유사도를 구하는 방식은 비교적 정확한 값을 구할 수 있으나, 한계가 뚜렷합니다. # Appendix: TF-IDF

이번 섹션에서는 텍스트 마이닝(Text Mining)에서 중요하게 사용되는 TF-IDF를 다뤄보겠습니다. TF-IDF는 어떤 단어  $w$ 가 문서  $d$  내에서 얼마나 중요한지 나타내는 수치입니다. 이 수치가 높을 수록  $w$ 는  $d$ 를 대표하는 성격을 띠게 된다고 볼 수도 있습니다.

TF-IDF의 TF는 Term Frequency로 단어의 문서 내에 출현한 횟수를 의미 합니다. 그리고 IDF는 Inverse Document Frequency로 그 단어가 출현한 문서의 숫자의 역수(inverse)를 의미 합니다. 다시 설명하면, Document Frequency (DF)는 해당 단어가 출현한 문서의 수가 되고, inverse가 붙어 역수를 취한 것 입니다.

$$\text{TF-IDF}(w, d) = \frac{\text{TF}(w, d)}{\text{DF}(w)}$$

TF는 단어가 문서 내에서 출현한 횟수입니다. 따라서 그 숫자가 클수록 문서 내에서 중요한 단어일 확률이 높습니다. 하지만, 'the'와 같은 단어도 TF값이 매우 클 것입니다. 하지만 'the'가 중요한 경우는 거의 없을 것 입니다. 따라서 이때 IDF가 필요 합니다. DF는 그 단어가 출현한 문서의 숫자를 의미 하기 때문에, 그 값이 클수록 'the'와 같이 일반적으로 많이 쓰이는 단어일 가능성이 높습니다. 따라서 IDF를 구해 TF에 곱해줌으로써, 'the'와 같은 단어들에 대한 penalty를 줍니다. 최종적으로 우리가 얻는 숫자는, 다른 문서들에서는 잘 나타나지 않지만 특정 문서에서만 잘 나타난 경우에 횟수가 높아지기 때문에, 특정 문서에서 얼마나 중요한 역할을 차지하는지 나타내는 수치가 될 수 있습니다.

## TF-IDF Example

우리는 아래와 같이 (이미 간단하게 tokenize 된) 여러 문서가 주어졌을 때, 단어들의 TF-IDF를 구하는 방법을 살펴보겠습니다.

### 번호 내용

1 지능  
지수  
라는  
말  
들  
어  
보  
셨  
을  
겁니다

. 여러분  
의  
지성  
을  
일컫  
는  
말  
이  
죠

. 그런데  
심리  
지수  
란  
건  
뭘까요  
?

사람

들

이

특정

한

식

으로

행동

하

는

8 이유

에

대해

여러분

은

얼마나

안

번호 내용  
2 최상  
의  
제시  
유형  
은  
학습  
자  
에  
좌우  
되  
는  
것  
이  
아니  
라  
학습  
해야  
할  
내용  
에  
따라  
좌우  
됩니다  
  
·  
예  
를  
들  
어  
여러분  
이  
운전  
하  
기  
를  
배울  
때  
실제로  
9 몸  
으로  
체감  
하  
는  
경험  
없이  
느낄까

### 번호 내용

3 그러나

이

이야기

는

세

가지

이유

로

인해

신화

입니다

.

첫째

,

가장

중요

한

건

실험실

가운

은

흰색

이

아니

라

회색

이

었

다

라는

점

이

죠

.

둘째

,

참

여자

10 들

은

실험

하

기

전

에

## 변호 내용

```
def get_term_frequency(document, word_dict=None):
    if word_dict is None:
        word_dict = {}
    words = document.split()

    for w in words:
        word_dict[w] = 1 + (0 if word_dict.get(w) is None else word_dict[w])

    return word_dict

def get_document_frequency(documents):
    dicts = []
    vocab = set([])
    df = {}

    for d in documents:
        tf = get_term_frequency(d)
        dicts += [tf]
        vocab = vocab | set(tf.keys())

    for v in list(vocab):
        df[v] = 0
        for dict_d in dicts:
            if dict_d.get(v) is not None:
                df[v] += 1

    return df

doc1 = '''
.
.
.
?'''

doc2 = '''
.
.
.
'''
```

```

doc3 = '''
    .
    .
    .

def get_tfidf(docs, top_k=30):
    vocab = {}
    tfs = []
    for d in docs:
        vocab = get_term_frequency(d, vocab)
        tfs += [get_term_frequency(d)]
    df = get_document_frequency(docs)

    from operator import itemgetter
    import numpy as np
    sorted_vocab = sorted(vocab.items(), key=itemgetter(1), reverse=True)

    stats = []
    for v, freq in sorted_vocab:
        tfidfs = []
        for idx in range(len(docs)):
            if tfs[idx].get(v) is not None:
                tfidfs += [tfs[idx][v] * np.log(len(docs) / df[v])]
            else:
                tfidfs += [0]

        stats += [(v, freq, tfidfs, max(tfidfs))]

    sorted_tfidfs = sorted(stats, key=itemgetter(3), reverse=True)[:top_k]
    for v, freq, tfidfs, max_tfidfs in sorted_tfidfs:
        print('%s\t%d\t%s' % (v, freq, '\t'.join(['%.4f' % tfidfs[i] for i in range(len(docs))])))
'''

>>> get_tfidf([doc1, doc2, doc3])

```

위의 코드를 차례대로 실행한 결과는 아래와 같습니다. 첫번째 문서에서 가장 중요한 단어는 '남자'임을 알 수 있고, 마찬가지로 두번째 문서는 '요인'인 것을 알 수 있습니다.

단어( $w$ )	총 출현 횟수	$\text{TF-IDF}(d_1)$	$\text{TF-IDF}(d_2)$	$\text{TF-IDF}(d_3)$
남자	9	9.8875	0.0000	0.0000

단어( $w$ )	총 출현 횟수	TF-IDF( $d_1$ )	TF-IDF( $d_2$ )	TF-IDF( $d_3$ )
요인	6	0.0000	6.5917	0.0000
심리학	5	5.4931	0.0000	0.0000
멀리	4	4.3944	0.0000	0.0000
시험	4	0.0000	4.3944	0.0000
환경	4	0.0000	4.3944	0.0000
성	4	0.0000	4.3944	0.0000
었	4	0.0000	0.0000	4.3944
제	4	0.0000	0.0000	4.3944
대해	3	3.2958	0.0000	0.0000
나	3	3.2958	0.0000	0.0000
간	3	3.2958	0.0000	0.0000
유형	3	0.0000	3.2958	0.0000
됩니다	3	0.0000	3.2958	0.0000
을까요	3	0.0000	3.2958	0.0000
인증	3	0.0000	3.2958	0.0000
탓	3	0.0000	3.2958	0.0000
유전자	3	0.0000	3.2958	0.0000
유전	3	0.0000	3.2958	0.0000
쌍둥이	3	0.0000	3.2958	0.0000
경우	3	0.0000	3.2958	0.0000
공유	3	0.0000	3.2958	0.0000
참여	3	0.0000	0.0000	3.2958
몸짓	3	0.0000	0.0000	3.2958
거짓말	3	0.0000	0.0000	3.2958
에서	8	2.4328	0.8109	0.0000
지수	2	2.1972	0.0000	0.0000
행동	2	2.1972	0.0000	0.0000
나요	2	2.1972	0.0000	0.0000
또	2	2.1972	0.0000	0.0000

## Using Other Features for Similarity

이번엔 다른 방식으로 단어의 유사도를 구하는 방법에 접근 해보겠습니다.  
자체적으로 단어에 대한 특성(feature)들을 모아 feature vector로 만들거나

유사도(similarity)를 계산하는 연산을 통해 단어간의 유사도를 구하는 방법입니다. 지금이야 어렵지않게 단어를 vector 형태로 embedding 할 수 있지만, 딥러닝 이전의 시대에는 쉽지 않은 일이었습니다. 이번 섹션을 통해서 딥러닝 이전의 전통적인 방식의 단어간의 유사도를 구하는 방법에 대해 알아보고, 이 방법의 단점과 한계에 대해서 살펴보겠습니다.

## Collecting Features

먼저, feature vector를 구성하는 방법들에 대해 살펴보고자 합니다. 결국 아래의 방법들이 하고자 하는 일은 같은 조건에 대해서 비슷한 수치를 반환하는 단어는 비슷한 유사도를 갖도록 벡터를 구성하는 것이라고 할 수 있습니다.

### Term-Frequency Matrix

앞서 우리는 TF-IDF에 대해서 살펴 보았습니다. TF-IDF에서 사용되었던 TF(term frequency)는 훌륭한 피쳐(feature)가 될 수 있습니다. 예를 들어 어떤 단어가 각 문서별로 출현한 횟수가 차원별로 구성되면, 하나의 feature vector를 이를 수 있습니다. 물론 각 문서별 TF-IDF 자체를 사용하는 것도 좋습니다.

```
def get_tf(docs):
    vocab = {}
    tfs = []
    for d in docs:
        vocab = get_term_frequency(d, vocab)
        tfs += [get_term_frequency(d)]

    from operator import itemgetter
    import numpy as np
    sorted_vocab = sorted(vocab.items(), key=itemgetter(1), reverse=True)

    stats = []
    for v, freq in sorted_vocab:
        tf_v = []
        for idx in range(len(docs)):
            if tfs[idx].get(v) is not None:
                tf_v += [tfs[idx][v]]
```

```

    else:
        tf_v += [0]

    print('%s\t%d\t%s' % (v, freq, '\t'.join(['%d' % tf for tf in tf_v])))

```

>>> get\_tf([doc1, doc2, doc3])

위의 코드를 사용하여 단어들의 각 문서별 출현횟수를 나타내면 아래와 같습니다.

단어( $w$ )	TF( $w$ )	TF( $w, d_1$ )	TF( $w, d_2$ )	TF( $w, d_3$ )
는	47	15	14	18
을	39	8	10	21
이	32	8	8	16
은	15	6	2	7
가	14	1	7	6
여러분	12	5	6	1
말	11	5	1	5
남자	9	9	0	0
여자	7	5	0	2
차이	7	5	2	0
요인	6	0	6	0
겁니다	5	2	1	2
얼마나	5	4	1	0
심리학	5	5	0	0
학습	5	0	4	1
이야기	5	0	1	4
결과	5	0	4	1
실제로	4	2	1	1
능력	4	3	1	0
멀리	4	4	0	0
시험	4	0	4	0
환경	4	0	4	0
사람	3	1	0	2
동일	3	2	1	0
유형	3	0	3	0
인증	3	0	3	0
유전자	3	0	3	0
수행	3	0	2	1

단어( $w$ )	TF( $w$ )	TF( $w, d_1$ )	TF( $w, d_2$ )	TF( $w, d_3$ )
연구	3	0	2	1
유전	3	0	3	0
쌍둥이	3	0	3	0
경우	3	0	3	0
모두	3	0	1	2
공유	3	0	3	0
인지	3	0	1	2
참여	3	0	0	3
몸짓	3	0	0	3
거짓말	3	0	0	3

여기서 마지막 3개 column이 각 단어별 문서에 대한 출현 횟수를 활용한 feature vector가 될 것 입니다. 지금은 문서가 3개밖에 없기 때문에, 사실 정확한 feature vector를 구성했다고 하기엔 무리가 있습니다. 따라서 문서가 많다면 우리는 지금보다 더 나은 feature vector를 구할 수 있을 것 입니다.

하지만 문서가 너무나도 많을 경우에는 벡터의 차원이 너무 커져버릴 수 있습니다. 예를 들어 문서가 10,000개가 있다고 하면 단어 당 10,000차원의 벡터가 만들어질 것 입니다. 문제는 이 10,000차원의 벡터 대부분은 값이 없이 0으로 채워져 있을 것입니다. 이렇게 벡터의 극히 일부분에만 의미있는 값들로 채워져 있는 벡터를 sparse vector라고 합니다. Sparse vector의 각 차원들은 사실 대부분의 경우 0일 것이기 때문에, 어떤 유의미한 통계를 얻는게 큰 장애물이 될 수 있습니다. 이처럼 희소성 문제는 자연어처리의 고질적인 문제로 작용 합니다.

또한, 단순히 문서에서의 출현 횟수를 가지고 feature vector를 구성하였기 때문에, 많은 정보가 유실되었고, 굉장히 단순화되어 여전히 매우 정확한 feature vector를 구성하였다고 하기엔 무리가 있습니다.

#### Based on Context Window (Co-occurrence)

함께 나타나는 단어들을 활용한 방법입니다. 의미가 비슷한 단어라면 쓰임새가 비슷할 것 입니다. 또한, 쓰임새가 비슷하기 때문에, 비슷한 문장 안에서 비슷한 역할로 사용될 것이고, 따라서 함께 나타나는 단어들이 유사할 것 입니다. 이러한 관점에서 우리는 함께 나타나는 단어들이 유사한 단어들의 유사도를 높게 주도록 만들어 줄 것 입니다.

함께 나타나는 단어들을 조사하기 위해서, 우리는 Context Window를 사용하여 windowing을 실행 합니다. (windowing이란 window를 움직이며 window안에 있는 유닛들의 정보를 취합하는 방법을 이릅니다.) 각 단어별로 window 내에 속해 있는 이웃 단어들을 counting하여 matrix로 나타내는 것 입니다.

이 방법은 좀 전에 다른 문서 내의 단어 출현 횟수(term frequency)를 가지고 feature vector를 구성한 방식보다 좀 더 정확하다고 할 수 있습니다. 하지만, window의 크기라는 하나의 hyper-parameter가 추가되어, 사용자가 그 크기를 정해주어야 합니다. 만약 window의 크기가 너무 크다면, 현재 단어와 너무 관계가 없는 단어들까지 counting 될 수 있습니다. 하지만, 너무 작은 window 크기를 갖는다면, 관계가 있는 단어들이 counting되지 않을 수 있습니다. 따라서, 적절한 window 크기를 정하는 것이 중요 합니다. 또한, window를 문장을 벗어나서도 적용 시킬 것인지도 중요합니다. 문제에 따라 다르지만 대부분의 경우에는 window를 문장 내에만 적용합니다.

python 코드를 통해 아래와 같은 문장들에 대해서 우리는 windowing을 수행 할 수 있습니다.

번호 내용	
1	왜
	냐고요
	?
2	산소
	의
	낭비
	였
	지요
.	.
3	어느
	날
,	,
	저
	는
	요요
	를
	샀
	습니다
.	.

번호 내용

4 저  
는  
회사  
의  
가치  
에  
따른  
가격  
책정  
을  
돕  
습니다

5 하지만  
내게  
매우  
내부  
적  
인  
문제  
가  
생겼  
다

... ...

번호 내용  
9995고독  
은  
여러분  
스스로  
찾  
을  
수  
있  
는  
곳  
에  
있  
어서  
다른  
사람  
들  
에게  
도  
다가  
갈  
수  
있  
습니다  
.

번호 내용  
9996두  
번  
째  
로  
이  
발견  
은  
새로운  
치료  
방법  
의  
아주  
분명  
한  
행로  
를  
제시  
합니다

.

여기  
서부터  
무엇  
을  
해야  
하  
는지  
는  
로켓  
과학자  
가  
아니  
더라도  
알  
수  
있  
잖아요

번호 내용  
9997전쟁  
전  
에  
는  
시리아  
도시  
에서  
그런  
요구  
들  
이  
완전히  
무시  
되  
었  
습니다  
.

번호 내용  
9998세로  
로  
된  
아찔  
한  
암석  
벽  
에  
둘러쌓  
여  
있  
으며  
숲  
에  
숨겨진  
은빛  
폭포  
도  
있  
죠  
.

번호 내용  
9999얼마간  
시간  
이  
지나  
면  
큰  
소리  
는  
더  
이상  
큰  
소리  
가  
아니  
게  
될  
겁니다

.  
10000]러  
한  
마을  
차원  
의  
아이디어  
는  
정말  
훌륭  
한  
아이디어  
입니다

—

```
def read(fn):  
    lines = []  
  
    f = open(fn, 'r')
```

```

for line in f:
    if line.strip() != '':
        lines += [line.strip()]
f.close()

return lines

def get_context_counts(lines, w_size=3):
    co_dict = {}
    for line in lines:
        words = line.split()

        for i, w in enumerate(words):
            for c in words[i - w_size:i + w_size]:
                if w != c:
                    co_dict[(w, c)] = 1 + (0 if co_dict.get((w, c)) is None else 0)

    return co_dict

from operator import itemgetter

fn = 'test.txt'
min_cnt, max_cnt = 0, 100000

lines = read(fn)
co_dict = get_context_counts(lines)
tfs = get_term_frequency(' '.join(lines))
sorted_tfs = sorted(tfs.items(), key=itemgetter(1), reverse=True)

context_matrix = []
row_heads = []
col_heads = [w for w, f in sorted_tfs if f >= min_cnt and f <= max_cnt]
for w, f in sorted_tfs:
    row = []
    if f >= min_cnt and f <= max_cnt:
        row_heads += [w]
        for w_, f_ in sorted_tfs:
            if f_ >= min_cnt and f_ <= max_cnt:
                if co_dict.get((w, w_)) is not None:

```

```

        row += [co_dict[(w, w_)]]
    else:
        row += [0]
context_matrix += [row]

import pandas as pd

p = pd.DataFrame(data=context_matrix, index=row_heads, columns=col_heads)

```

그리고 이 코드를 통해 얻은 결과의 일부는 아래와 같습니다. 이 결과에 따르면 1000개의 문장(문서)에서는 '습니다'의 context window 내에 마침표가 3616번 등장합니다.

	는	이	을	은	하	의	예	,	들	있	...	다다를	유리병	엉터리	켠	기념관	외침	스프를	이야	금성	악상어
는	0	4227	3554	320	13174	1324	6800	4049	315	8655	...	0	1	0	1	0	0	0	0	2	3
이	8466	0	706	3342	494	3450	1423	2797	12935	4470	...	1	0	0	0	0	1	1	0	1	0
을	5496	3031	0	3064	2249	5075	1875	1119	6720	1783	...	0	1	2	1	2	2	0	1	1	0
은	3298	2764	394	0	40	1994	442	2514	9052	59	...	0	2	2	0	0	0	1	0	3	0
하	13364	2407	9382	717	0	112	1945	700	232	336	...	0	0	0	0	0	0	0	0	0	0
의	3289	1965	295	2386	29	0	409	1858	3699	21	...	0	1	1	0	0	6	3	1	2	3
예	5965	2431	625	1574	391	2945	0	1935	1299	3145	...	7	7	0	5	3	1	1	0	0	2
,	2053	2858	969	1541	2899	675	1305	0	1297	1932	...	0	1	1	1	2	0	0	1	2	0
들	3958	13284	6696	10531	23	5851	1447	2311	0	16	...	0	0	3	0	0	1	0	4	0	0
있	8674	4684	2692	169	2997	20	3802	71	932	0	...	3	0	0	0	0	0	0	0	0	0
습니다	465	3532	1406	218	621	1	663	470	167	11875	...	0	0	0	0	0	0	0	0	0	3
가	2788	1517	351	2008	365	1791	1236	2067	211	2880	...	0	0	1	0	1	0	0	0	0	1
를	2274	2219	96	1768	1247	3336	780	1044	641	179	...	0	0	0	0	0	0	0	0	0	3
고	456	3332	5220	250	8667	43	828	6303	291	12122	...	0	0	0	0	0	1	0	0	0	0
것	10318	7502	5762	5080	3916	414	1296	334	1884	2445	...	3	0	0	0	0	0	0	0	0	0

Figure 6: context windowing을 수행한 결과

앞쪽 출현빈도가 많은 단어들은 대부분 값이 잘 채워져 있는 것을 볼 수 있습니다. 하지만 뒤쪽 출현빈도가 낮은 단어들은 많은 부분이 0으로 채워져 있는 것을 볼 수 있습니다. 출현빈도가 낮은 단어들의 row로 갈 경우에는 그 문제가 더욱 심각해집니다.

위의 context windowing 실행 결과 얻은 feature vector들을 tSNE로 시각화 한 모습은 아래와 같습니다. 딱히 비슷한 단어끼리 모이지 않은 것도 많지만, 운좋게 비슷한 단어끼리 붙어 있는 경우도 종종 볼 수 있습니다.

베네수엘라	1	0	0	0	0	3	1	3	0	0	...	0	0	0	0	0	0	0	0	0	0
표하	0	0	2	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
쿠데타	0	0	0	1	0	1	0	0	1	0	...	0	0	0	0	0	0	0	0	0	0
소용없	1	2	0	1	0	0	1	0	0	0	...	0	0	0	0	0	0	0	0	0	0
유실	0	3	1	2	0	2	0	1	2	0	...	0	0	0	0	0	0	0	0	0	0
거장	0	2	0	1	0	5	0	1	2	0	...	0	0	0	0	0	0	0	0	0	0
물어봅니다	1	2	0	0	0	0	0	0	1	0	...	0	0	0	0	0	0	0	0	0	0
자연스레	2	0	3	0	1	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
대포	0	0	0	0	0	1	1	0	1	0	...	0	0	0	0	0	0	0	0	0	0
분당	4	0	0	0	1	0	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0

Figure 7: 대부분의 값이 0으로 채워진 sparse vector

## Get Similarity between Feature Vectors

그럼 이렇게 구해진 feature vector를 어떻게 사용할 수 있을까요? Feature vector는 단어 사이의 유사도를 구할 때 아주 유용하게 쓸 수 있습니다. 그럼 벡터 사이의 유사도 또는 거리는 어떻게 구할 수 있을까요? 아래에서는 두 벡터가 주어졌을 때, 벡터 사이의 유사도 또는 거리를 구하는 방법들에 대해 다루어 보겠습니다. 그리고 PyTorch를 사용하여 해당 수식들을 직접 구현 해 보겠습니다.

```
import torch
```

### Manhattan Distance (L1 distance)

$$d_{L1}(w, v) = \sum_{i=1}^d |w_i - v_i|, \text{ where } w, v \in \mathbb{R}^d.$$

L1 norm을 사용한 Manhattan distance (맨하튼 거리)입니다. 이 방법은 두 벡터의 각 차원별 값의 차이의 절대값을 모두 합한 값입니다.

```
def get_l1_distance(x1, x2):
    return ((x1 - x2).abs()).sum()**.5
```

위의 코드는 torch tensor  $x_1$ ,  $x_2$ 를 입력으로 받아 L1 distance를 리턴해 주는 코드입니다.

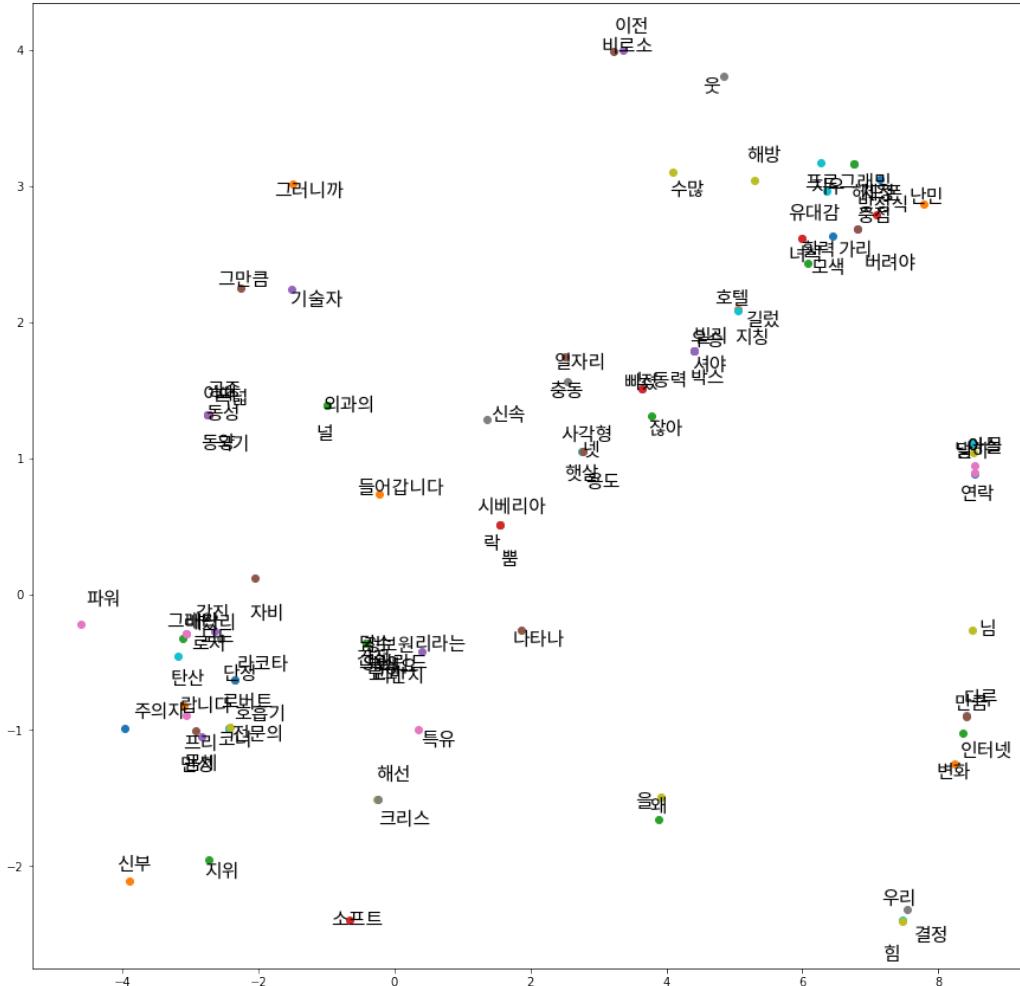


Figure 8: feature vector들을 tSNE로 표현한 모습

### Euclidean Distance (L2 distance)

$$d_{L2}(w, v) = \sqrt{\sum_{i=1}^d (w_i - v_i)^2}, \text{ where } w, v \in \mathbb{R}^d.$$

우리가 가장 친숙한 거리 방법 중의 하나인 Euclidean distance (유클리드 거리)입니다. 각 차원별 값 차이의 제곱의 합에 루트를 취한 형태입니다.

```
def get_l2_distance(x1, x2):
    return ((x1 - x2)**2).sum()**.5
```

L2 distance를 구하기 위해 torch tensor들을 입력으로 받아 계산하는 함수의 코드는 위와 같습니다.

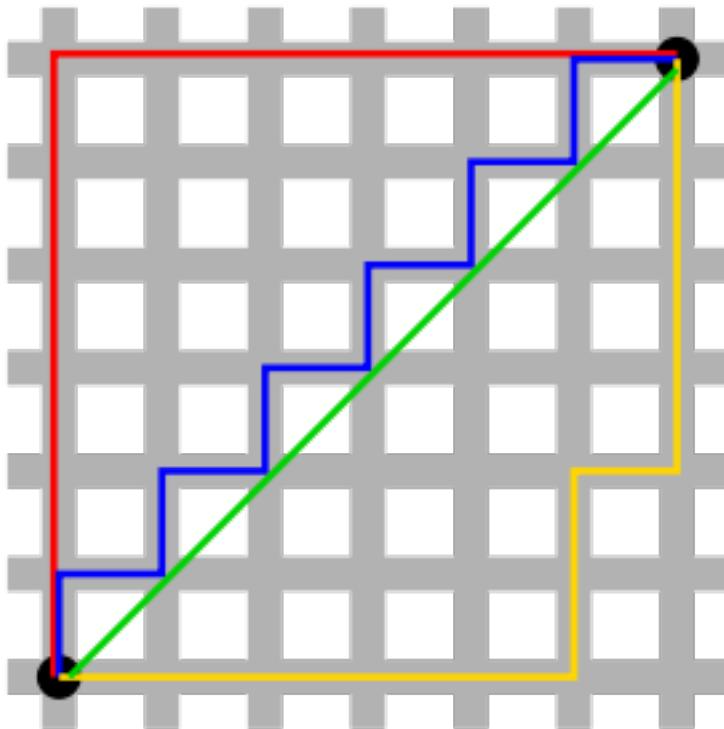


Figure 9: L1 vs L2(초록색) from wikipedia

위의 그림은 Manhattan distance와 Euclidean distance의 차이를 쉽게 나타낸 그림입니다. Euclidean distance를 의미하는 초록색 선을 제외하고 나머지 Manhattan

distance를 나타내는 세가지 선의 길이는 모두 같습니다. Manhattan distance는 그 이름답게 마치 잘 계획된 도시의 길을 지나가는 듯한 선의 형태를 나타내고 있습니다.

### Using Infinity Norm

$$d_\infty(w, v) = \max(|w_1 - v_1|, |w_2 - v_2|, \dots, |w_d - v_d|), \text{ where } w, v \in \mathbb{R}^d$$

$L_1$ ,  $L_2$  distance가 있다면  $L_\infty$  distance도 있습니다. 재미있게도 infinity norm을 이용한 distance는 각 차원별 값의 차이 중 가장 큰 값을 나타냅니다. 위의 수식을 torch tensor에 대해서 계산하는 코드는 아래와 같습니다.

```
def get_infinity_distance(x1, x2):
    return ((x1 - x2).abs()).max()
```

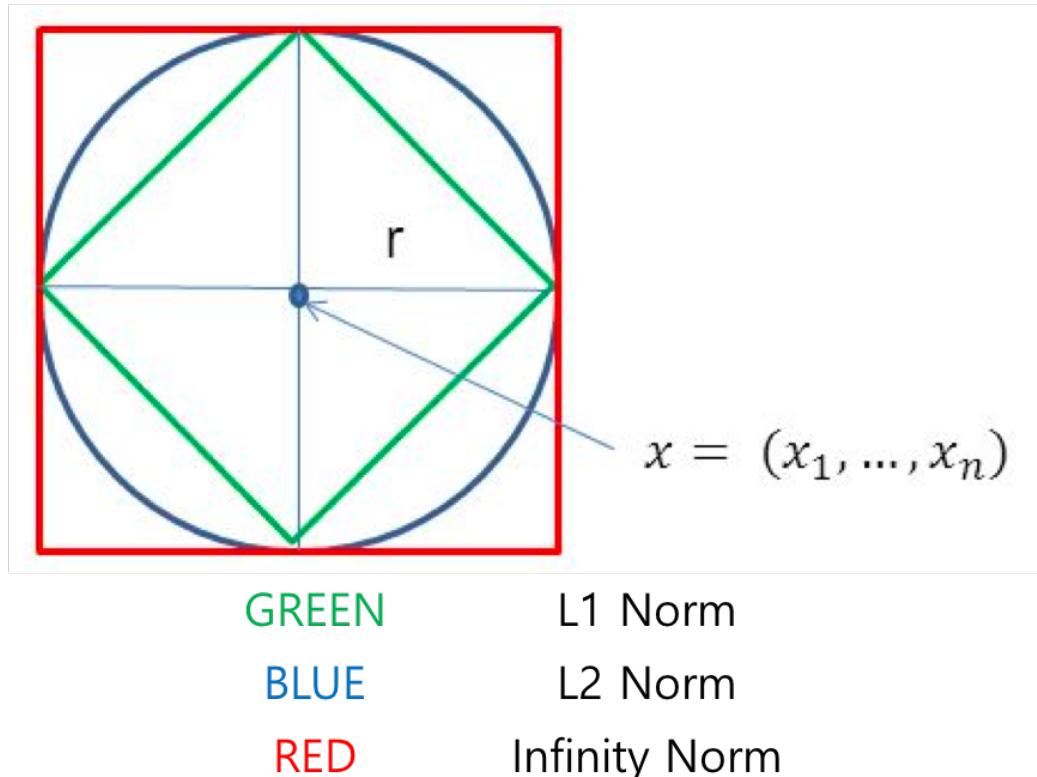


Figure 10: 같은 값  $r$  크기를 갖는  $L_1$ ,  $L_2$ ,  $L_\infty$  거리를 그림으로 나타낸 모습

위의 그림은 각  $L_1$ ,  $L_2$ ,  $L_\infty$  별로 거리의 크기가  $r$ 일때 모습입니다.

## Cosine Similarity

$$\begin{aligned} \text{sim}_{\cos}(w, v) &= \frac{\overbrace{w \cdot v}^{\text{dot product}}}{|w||v|} = \frac{\widehat{w}}{|w|} \cdot \frac{\widehat{v}}{|v|} \\ &= \frac{\sum_{i=1}^d w_i v_i}{\sqrt{\sum_{i=1}^d w_i^2} \sqrt{\sum_{i=1}^d v_i^2}} \\ \text{where } w, v &\in \mathbb{R}^d \end{aligned}$$

위와 같은 수식을 갖는 cosine similarity(코사인 유사도)함수는 두 벡터 사이의 방향과 크기를 모두 고려하는 방법입니다. 수식에서 분수의 윗변은 두 벡터 사이의 element-wise 곱을 사용하므로 벡터의 내적과 같습니다. 따라서 cosine similarity의 결과가 1에 가까울수록 방향은 일치하고, 0에 가까울수록 수직(orthogonal)이며, -1에 가까울수록 반대 방향임을 의미 합니다. 위와 같이 cosine similarity는 크기와 방향 모두를 고려하기 때문에, 자연어처리에서 가장 널리 쓰이는 유사도 측정 방법입니다. 하지만 수식 내 윗변의 벡터 내적 연산이나 밑변 각 벡터의 크기(L2 norm)를 구하는 연산이 비싼 편에 속합니다. 따라서 vector 차원의 크기가 클수록 연산량이 부담이 됩니다.

Cosine similarity는 아래와 같이 PyTorch 코드로 나타낼 수 있습니다.

```
def get_cosine_similarity(x1, x2):
    return (x1 * x2).sum() / ((x1**2).sum()**.5 * (x2**2).sum()**.5)
```

## Jaccard Similarity

$$\begin{aligned} \text{sim}_{\text{jaccard}}(w, v) &= \frac{|w \cap v|}{|w \cup v|} \\ &= \frac{|w \cap v|}{|w| + |v| - |w \cap v|} \\ &\approx \frac{\sum_{i=1}^d \min(w_i, v_i)}{\sum_{i=1}^d \max(w_i, v_i)} \\ \text{where } w, v &\in \mathbb{R}^d. \end{aligned}$$

Jaccard similarity는 두 집합 간의 유사도를 구하는 방법입니다. 수식의 윗변에는 두 집합의 교집합의 크기가 있고, 이를 밑변에서 두 집합의 합집합의 크기로 나누어

줍니다. 이때, Feature vector의 각 차원이 집합의 element가 될 것입니다. 다만, 각 차원에서의 값이 0 또는 0이 아닌 값이 아니라, 수치 자체에 대해서 Jaccard similarity를 구하고자 할 때에는, 두번째 줄의 수식과 같이 두 벡터의 각 차원의 숫자에 대해서 min, max 연산을 통해서 계산 할 수 있습니다. 이를 PyTorch 코드로 나타내면 아래와 같습니다.

```
def get_jaccard_similarity(x1, x2):
    return torch.stack([x1, x2]).min(dim=0)[0].sum() / torch.stack([x1, x2]).max(0)
```

## Appendix: Similarity between Documents

방금은 단어에 대한 feature를 수집하고 유사도를 구하였다면, 마찬가지로 문서에 대해 feature를 추출하여 문서간의 유사도를 구할 수 있습니다. 예를 들어 문서내의 단어들에 대해 출현 빈도(term frequency)나 TF-IDF를 구하여 vector를 구성하고, 이를 활용하여 vector 사이의 유사도를 구할 수도 있을 것 입니다. # Word Sense Disambiguation

이번 세션에서는 단어의 유사도가 아닌, 모호성에 대해 다루어 보겠습니다. 하나의 단어는 여러가지 의미를 지닐 수 있습니다. 비슷한 의미의 동형이의어(homonym)인 경우에는 비교적 그 문제가 크지 않을 수 있습니다. 조금은 어색한 문장 표현이나 이해가 될 수 있겠지만, 큰 틀에서는 벗어나지 않기 때문입니다. 하지만 다의어(polysemy)의 경우에는 문제가 커집니다. 앞서 예를 들었던 '차'의 경우에 'tea'의 의미로 해석되느냐, 'car'의 의미로 해석되느냐에 따라 문장의 의미가 매우 달라질것이기 때문입니다. 따라서 이와 같이 단어가 주어졌을 때, 그 의미의 모호성을 없애고 해석하는 것이 매우 중요합니다. 이를 단어 중의성 해소(word sense disambiguation)이라고 합니다.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

---

G\* I  
was  
kick-  
ing  
her  
in  
the  
car  
that  
went  
to  
the  
park  
for  
tea.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

---

M\* I  
was  
a  
car  
to  
her,  
in  
the  
car  
I  
had  
a  
car  
and  
went  
to  
the  
park.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

---

N\* I  
got  
dumped  
by  
her  
on  
the  
way  
to  
the  
park  
for  
tea.

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

K\* I  
was  
in  
the  
car  
go-  
ing  
to  
the  
park  
for  
tea  
and  
I  
was  
in  
her  
car.

---

차를  
마시러  
공원에  
가던  
차  
안에서  
나는  
그녀에게  
원문 차였다.

---

S\* I  
got  
dumped  
by  
her  
in  
the  
car  
that  
was  
go-  
ing  
to  
the  
park  
for  
a  
cup  
of  
tea.

---

위와 같이 다양한 '차'가 문장에 나타났을 때, 실제 기계번역에 있어서 심각한 성능저하의 원인이 되기도 합니다.

## Thesaurus Based Method: Lesk Algorithm

Lesk 알고리즘은 가장 간단한 사전 기반 중의성 해소 방법입니다. 주어진 문장에서 특정 단어에 대해서 의미를 명확히 하고자 할 때 사용 할 수 있습니다. 이를 위해서 Lesk 알고리즘은 간단한 가정을 하나 만듭니다. 문장 내에 같이 등장하는 단어(context)들은 공통 토픽을 공유한다는 것 입니다.

이 가정을 바탕으로 동작하는 Lesk 알고리즘의 개요는 다음과 같습니다. 먼저, 중의성을 해소하고자 하는 단어에 대해서 사전(주로 WordNet)의 의미별 설명과 주어진 문장 내에 등장한 단어의 사전에서 의미별 설명 사이의 유사도를 구합니다. 유사도를 구하는 방법은 여러가지가 있을테지만, 가장 간단한 방법으로는 겹치는 단어의 갯수를 구하는 것이 될 수 있습니다. 이후, 문장 내 단어들의 의미별 설명과 가장 유사도가 높은 (또는 겹치는 단어가 많은) 의미가 선택 됩니다.

예를 들어 NLTK의 WordNet에 'bass'를 검색해 보면 아래와 같습니다.

```
>>> from nltk.corpus import wordnet as wn
>>> for ss in wn.synsets('bass'):
...     print(ss, ss.definition())
...
Synset('bass.n.01') the lowest part of the musical range
Synset('bass.n.02') the lowest part in polyphonic music
Synset('bass.n.03') an adult male singer with the lowest voice
Synset('sea_bass.n.01') the lean flesh of a saltwater fish of the family Serranidae
Synset('freshwater_bass.n.01') any of various North American freshwater fish with
Synset('bass.n.06') the lowest adult male singing voice
Synset('bass.n.07') the member with the lowest range of a family of musical instruments
Synset('bass.n.08') nontechnical name for any of numerous edible marine and freshwater fishes
Synset('bass.s.01') having or denoting a low vocal or instrumental range
```

Lesk 알고리즘 수행을 위하여 간단하게 래핑(wrapper)하도록 하겠습니다.

```
def lesk(sentence, word):
    from nltk.wsd import lesk

    best_synset = lesk(sentence.split(), word)
    print(best_synset, best_synset.definition())
```

아래와 같이 주어진 문장에서는 'bass'는 물고기의 의미로 뽑히게 되었습니다.

```
>>> sentence = 'I went fishing last weekend and I got a bass and cooked it'
```

```
>>> word = 'bass'  
>>> lesk(sentence, word)
```

Synset('sea\_bass.n.01') the lean flesh of a saltwater fish of the family Serranidae

또한, 아래의 문장에서는 음악에서의 역할을 의미하는 것으로 추정되었습니다.

```
>>> sentence = 'I love the music from the speaker which has strong beat and bass'  
>>> word = 'bass'  
>>> lesk(sentence, word)
```

Synset('bass.n.02') the lowest part **in** polyphonic music

여기까지 보면 Lesk 알고리즘은 비교적 잘 동작하는 것으로 보입니다. 하지만, 비교적 정확하게 예측해낸 위의 두 사례와 달리, 아래의 문장에서는 전혀 다른 의미로 예측하는 것을 볼 수 있습니다.

```
>>> sentence = 'I think the bass is more important than guitar'  
>>> word = 'bass'  
>>> lesk(sentence, word)
```

Synset('sea\_bass.n.01') the lean flesh of a saltwater fish of the family Serranidae

이처럼 Lesk 알고리즘은 명확한 장단점을 지니고 있습니다. WordNet과 같이 잘 분류된 사전이 있다면, 쉽고 빠르게 중의성해소를 해결할 수 있을 것입니다. 또한 WordNet에서 이미 단어별로 몇개의 의미를 갖고 있는지 잘 정의 해 놓았기 때문에, 크게 고민 할 필요도 없습니다. 하지만 보다시피 사전의 단어 및 의미에 대한 설명에 크게 의존하게 되고, 설명이 부실하거나 주어진 문장에 큰 특징이 없을 경우 단어 중의성해소 능력은 크게 떨어지게 됩니다. 그리고 WordNet이 모든 언어에 대해서 존재하는 것이 아니기 때문에, 사전이 존재하지 않는 언어에 대해서는 Lesk 알고리즘 자체의 수행이 어려울 수도 있습니다.

## Monty-hall Problem

$$\begin{aligned}
P(C = 2|A = 0, B = 1) &= \frac{P(A = 0, B = 1, C = 2)}{P(A = 0, B = 1)} \\
&= \frac{P(B = 1|A = 0, C = 2)P(A = 0, C = 2)}{P(A = 0, B = 1)} \\
&= \frac{P(B = 1|A = 0, C = 2)P(A = 0)P(C = 2)}{P(B = 1|A = 0)P(A = 0)} \\
&= \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{2}{3},
\end{aligned}$$

where  $P(B = 1, A = 0) = \frac{1}{2}$ ,  $P(C = 2) = \frac{1}{3}$ , and  $P(B = 1|A = 0, C = 2) = 1$ .

$$\begin{aligned}
P(C = 0|A = 0, B = 1) &= \frac{P(A = 0, B = 1, C = 0)}{P(A = 0, B = 1)} \\
&= \frac{P(B = 1|A = 0, C = 0)P(A = 0, C = 0)}{P(A = 0, B = 1)} \\
&= \frac{P(B = 1|A = 0, C = 0)P(A = 0)P(C = 0)}{P(B = 1|A = 0)P(A = 0)} \\
&= \frac{\frac{1}{2} \times \frac{1}{3}}{\frac{1}{2}} = \frac{1}{3},
\end{aligned}$$

where  $P(B = 1|A = 0, C = 0) = \frac{1}{2}$

### # Selectional Preference

이번 섹션에서는 selectional preference라는 개념에 대해서 다루어 보도록 하겠습니다. 문장은 여러 단어의 시퀀스로 이루어져 있습니다. 따라서 각 단어들은 문장내 주변의 단어들에 따라 그 의미가 정해지기 마련입니다. Selectional preference는 이를 좀 더 수치화하여 나타내 줍니다. 예를 들어 ‘마시다’라는 동사에 대한 목적어는 ‘클래스에 속하는 단어가 올 확률이 매우 높습니다. 따라서 우리는 ‘차’라는 단어가 ‘클래스에 속하는지’ 클래스에 속하는지 쉽게 알 수 있습니다. 이런 성질을 이용하여 우리는 단어 중의성 해소(WSD)도 해결 할 수 있으며, 여러가지 문제들(syntactic disambiguation, semantic role labeling)을 수행할 수 있습니다.

## Selectional Preference Strength

앞서 언급한 것처럼 selectional preference는 단어와 단어 사이의 관계(예: verb-object)가 좀 더 특별한 경우에 대해 수치화 하여 나타냅니다. 술어(predicate) 동사(예: verb)가 주어졌을 때, 목적어(예: object)관계에 있는 headword 단어(보통은 명사가 될 겁니다.)들의 분포는, 평소 문서 내에 해당 명사(예: object로써 noun)가 나올 분포와 다를 것입니다. 그 분포의 차이가 크면 클수록 해당 술어(predicate)는 더 강력한 selectional preference를 갖는다고 할 수 있습니다. 이것을 Philip Resnik은 [Resnik et al.1997]에서 Selectional Preference Strength라고 명명하고 KL-divergence를 사용하여 정의하였습니다.

$$\begin{aligned} S_R(w) &= \text{KL}(P(C|w)||P(C)) \\ &= -\sum_{c \in C} P(c|w) \log \frac{P(c)}{P(c|w)} \\ &= -\mathbb{E}_{C \sim P(C|w)} [\log \frac{P(C)}{P(C|W=w)}] \end{aligned}$$

위의 수식을 해석하면, selectional preference strength  $S_R(w)$ 은  $w$ 가 주어졌을 때의 object class  $C$ 의 분포  $P(C|w)$ 와 그냥 해당 class들의 prior(사전) 분포  $P(C)$ 와의 KL-divergence로 정의되어 있음을 알 수 있습니다. 즉, selectional preference strength는 술어(predicate)가 headword로 특정 클래스를 얼마나 선택적으로 선호(selectional preference)하는지에 대한 수치라고 할 수 있습니다.

예를 들어 “클래스의 단어는” 클래스의 단어보다 나타날 확률이 훨씬 높을 것 입니다. 이때, ‘사용하다’라는 동사(verb) 술어(predicate)가 주어진다면, 동사-목적어(verb-object) 관계에 있는 headword로써의 “클래스의 확률은” 클래스의 확률보다 낮아질 것입니다.

## Selectional Association

이제 그럼 술어와 특정 클래스 사이의 선택 관련도를 어떻게 나타내는지 살펴보겠습니다. Selectional Association,  $A_R(w, c)$ 은 아래와 같이 표현됩니다.

$$A_R(w, c) = -\frac{P(c|w) \log \frac{P(c)}{P(c|w)}}{S_R(w)}$$

Selectional Preference Strength

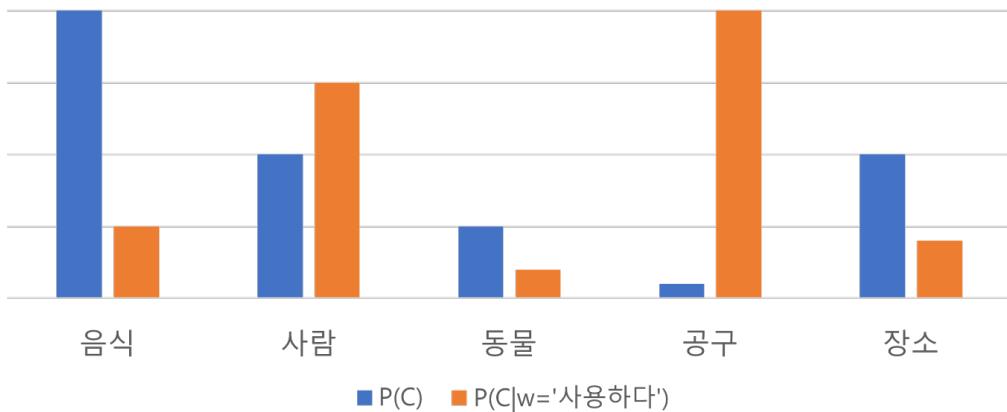


Figure 11: 클래스의 사전 확률 분포와 술어가 주어졌을 때의 확률 분포 변화

위의 수식에 따르면, selectional preference strength가 낮은 술어(predicate)에 대해서 얇변의 값이 클 경우에는 술어와 클래스 사이에 더 큰 selectional association(선택 관련도)를 갖는다고 정의 합니다. 즉, selectional preference strength가 낮아서, 해당 술어(predicate)는 클래스(class)에 대한 선택적 선호 강도가 낮음에도 불구하고, 특정 클래스만 유독 술어에 영향을 받아서 얇변이 커질수록 selectional association의 수치도 커집니다.

예를 들어 어떤 아주 일반적인 동사에 대해서는 대부분의 클래스들이 prior(사전)확률 분포와 비슷하게 여전히 나타날 것입니다. 따라서 selectional preference strength,  $S_R(w)$ 는 0에 가까울것입니다. 하지만 그 가운데 해당 동사와 붙어서 좀 더 나타나는 클래스의 목적어가 있다면, selectional association  $A_R(w, c)$ 는 매우 높게 나타날 것입니다.

## Selectional Preference and WSD

눈치가 빠른 분들은 이미 눈치 채셨겠지만, 우리는 이런 selectional preference의 특성을 이용하여 단어 중의성 해소(Word Sense Disambiguation)에 활용할 수 있습니다. ‘마시다’라는 동사에 ‘차’라는 목적어가 함께 있을 때, 우리는 selectional preference를 통해서 ‘차’는” 클래스에 속한다고 말할 수 있을 것 입니다.

문제는 ‘차’가 “또는” 클래스에 속하는 것을 알아내는 것입니다. 우리가 가지고 있는 코퍼스는 단어들로 표현되어 있지, 클래스로 표현되어 있지는 않습니다. 만약 우리는 단어가 어떤 클래스들에 속하는지 미리 알고 있다면 단어들의 출현 빈도를 세어 클래스의 확률 분포를 estimation(추정)할 수 있을 것 입니다. 결국 이를 위해서는 사전에 정의되어 있는 지식 또는 데이터셋이 필요할 것 입니다.

## Selectional Preference based on WordNet

이때, WordNet이 위력을 발휘합니다. 우리는 WordNet을 통해서 ‘차(car)’의 hypernym을 알 수 있고, 이것을 클래스로 삼으면 우리가 필요한 정보들을 얻을 수 있습니다. 영어 WordNet에서의 예를 들어 ’eat’ 뒤에 있는 ’bass’의 hypernym을 통해 먹는 물고기 ’bass’인지, 악기 ’bass’인지 구분 할 수 있을 것 입니다. [Resnik et al.1997]에서는 술어와 클래스 사이의 확률 분포를 정의하기 위한 출현빈도를 계산하는 수식을 아래와 같이 제안하였습니다.

$$\text{Count}_R(w, c) \approx \sum_{h \in c} \frac{\text{Count}_R(w, h)}{|\text{Classes}(h)|}$$

클래스  $c$ 에 속하는 headword,  $h$ 는 실제 코퍼스(corpus)에 나타난 단어로써, 술어(predicate)  $w$ 와 함께 출현한 headword  $h$ 의 빈도를 세고,  $h$ 가 속하는 클래스들의 set의 크기,  $|\text{Classes}(h)|$ 로 나누어 줍니다. 그리고 이를 클래스  $c$ 에 속하는 모든 단어(headword)에 대해서 수행한 후, 이를 합한 값은  $\text{Count}_R(w, c)$ 를 근사(approximation)합니다.

$$\hat{c} = \underset{c \in \mathcal{C}}{\operatorname{argmax}} A_R(w, c), \text{ where } \mathcal{C} = \text{hypernym}(h).$$

이를 통해 우리는 predicate  $w$ 와 headword  $h$ 가 주어졌을 때,  $h$ 의 클래스  $c$ 를 추정한  $\hat{c}$ 를 구할 수 있습니다.

## Selectional Preference Evaluation using Pseudo Word

단어 중의성 해소(WSD)를 해결할 수 있는 방법 중에 하나로, selectional preference를 살펴 보았습니다. Selectional preference를 잘 해결할 수 있다면 아마도 단어 중의성 해소 문제도 잘 해결 될 것 입니다. 그럼 selectional preference를 어떻게 평가 할 수

있을까요? 정교한 테스트셋을 설계하고 만들어서 selectional preference의 성능을 평가 할 수 있겠지만, 좀 더 쉽고 general한 방법은 없을까요?

Pseudo Word가 하나의 해답이 될 수 있습니다. Pseudo word는 두개의 단어가 인위적으로 합성되어 만들어진 단어를 이릅니다. 실제 일상 생활에서 쓰이기보다는 단순히 두 단어를 합친 것 입니다. 예를 들어 banana와 door를 합쳐 banana-door라는 pseudo word를 만들어 낼 수 있습니다. 이 banana-door라는 단어는 사실 실생활에서 쓰일리 없는 단어입니다. 하지만 우리는 이 단어가 eat 또는 open이라는 동사 술어(verb predicate)와 함께 headword object로써 나타났을 때, eat에 대해서는 banana를 선택해야 하고, open에 대해서는 door를 선택하도록 해야 올바른 selectional preference 알고리즘을 만들거나 구현했음을 확인할 수 있습니다.

[Chambers et al.2010]

### Similarity-based Selectional Preference

위와 같이 selectional preference를 WordNet등의 도움을 받아 구현하는 방법을 살펴보았습니다. 하지만, WordNet이라는 것은 아쉽게도 모든 언어에 존재하지 않으며, 새롭게 생겨난 신조어들도 반영되어 있지 않을 가능성이 매우 높습니다. 따라서 WordNet과 같은 thesaurus에 의존하지 않고 selectional preference를 구할 수 있으면 매우 좋을 것 입니다. [Erk et al.2007]에서는 단어간의 유사도를 통해 thesaurus에 의존하지 않고 간단하게 selectional preference를 구하는 방법을 제시하였습니다.

$(w, h, R)$ , where  $R$  is a relationship, such as verb-object.

이전과 같이 문제 정의는 비슷합니다. 술어(predicate)  $w$ 와 headword  $h$ , 그리고 두 단어 사이의 관계  $R$ 이 tuple로 주어집니다. 이때, selectional association,  $A_R(w, h_0)$ 은 아래와 같이 정의 할 수 있습니다.

$$A_R(w, h_0) = \sum_{h \in \text{Seen}_R(w)} \text{sim}(h_0, h) \cdot \phi_R(w, h)$$

이때, weight  $\phi_R(w, h)$ 은 uniform하게 1로 주어도 되고, 아래와 같이 Inverse Document Frequency (IDF)를 사용하여 정의 할 수도 있습니다.

$$\phi_R(w, h) = \text{IDF}(h)$$

또한, sim함수는 이전에 다루었던 cosine similarity나 jaccard similarity를 포함하여 다양한 유사도 함수를 사용할 수 있습니다.

다만, 이전에 다루었던 selectional association은  $A_R(w, c)$ 을 다루었는데, 지금은  $A_R(w, h_0)$ 을 다루는 것이 차이점이긴 합니다. 하지만 우리는 좀 전에 selectional preference의 문제를 선호하는 클래스를 알아내는 것이 아닌, pseudo word와 같이 두 개의 단어가 주어졌을 때 선호하는 단어를 고르는 문제로 만들었습니다. 따라서 큰 문제가 되지 않습니다.

이 방법은 쉽게 WordNet과 같은 thesaurus 없이 쉽게 selectional preference를 계산할 수 있게 합니다. 하지만 유사도를 비교하기 위해서  $Seen_R(w)$  함수를 통해 대상 단어를 선정하기 때문에, 코퍼스에 따라서 유사도를 구할 수 있는 대상이 달라지게 됩니다. 따라서 이에 따뜻 커버리지 문제가 발생할 수 있습니다.

## Example

```
from konlpy.tag import Mecab

def count_seen_headwords(lines, predicate='VV', headword='NNG'):
    mecab = Mecab()
    seen_dict = {}

    for line in lines:
        pos_result = mecab.pos(line)

        word_h = None
        word_p = None
        for word, pos in pos_result:
            if pos == predicate or pos[:3] == predicate + '+':
                word_p = word
                break
            if pos == headword:
                word_h = word

        if word_h is not None and word_p is not None:
            seen_dict[word_p] = [word_h] + ([] if seen_dict.get(word_p) is None else seen_dict[word_p])

    return seen_dict
```

```

def get_selectional_association(predicate, headword, lines, dataframe, metric):
    v1 = torch.FloatTensor(dataframe.loc[headword].values)
    seens = count_seen_headwords(lines)[predicate]

    total = 0
    for seen in seens:
        try:
            v2 = torch.FloatTensor(dataframe.loc[seen].values)
            total += metric(v1, v2)
        except:
            pass

    return total

def wsdl(predicate, headwords):
    selectional_associations = []
    for h in query_h:
        selectional_associations += [get_selectional_association(query_p, h, lines[h])]

    print(selectional_associations)

>>> wsdl(' ', [' ', ' ', ' ', ' '])
[tensor(6.1853), tensor(3.9723), tensor(3.8503)]

```

## Conclusion

이번 챕터에서는 단어의 의미에 대해서 다뤄보고, 의미의 유사성이나 모호성에 대해서 다루어보았습니다. 단어는 곁의 discrete한 형태와 달리 내부적으로는 non-discrete한 '의미(sense)'를 갖고 있습니다. 따라서 우리는 단어의 곁 모양이 다르더라도 의미가 유사할 수 있음을 알고 있습니다. 이렇게 의미가 유사한 단어들간의 유사도를 계산 할 수 있다면, 코퍼스(corpus)로부터 분포나 특징(feature)들을 훈련 할 때 좀 더 많은 정보를 얻고 정확한 훈련을 할 수 있습니다. 예를 들어 꼭 source 단어가 아니더라도 그 source 단어와 유사한 단어들로부터 정보를 얻어올 수 있고, 그 정보를 source 단어와 유사한 만큼만 사용하면 될 것입니다.

따라서, 이러한 자연어처리의 염원 아래, 사람이 직접 한땀한땀 정성들여 만든

WordNet이라는 사전이 등장하게 되었고, WordNet을 활용하여 단어 사이의 유사도(거리)를 계산할 수도 있게 되었습니다. 하지만 이렇게 WordNet과 같은 사전(thesaurus)을 구축하는 것은 너무나도 엄청난 일이므로, 사전이 없이 유사도를 구할 수 있으면 더 좋을 것 입니다.

비록 사전이 없이 코퍼스만 가지고 특징(feature)을 추출하여 단어를 벡터로 만든다면, WordNet의 정교한 지식은 이용할 수 없겠지만, 훨씬 더 간단한 작업이 될 것입니다. 더욱이 코퍼스의 크기가 커질수록 추출된 특징들은 점점 더 정확해 질 것이고, feature vector는 더욱 정확해 질 것입니다. Feature vector가 추출 된 이후에는 cosine similarity나 L2 distance등의 metric을 통해서 유사도를 계산할 수 있습니다.

하지만, 이전에 보았듯이, 단어 사전의 크기가 30,000~50,000이 넘는 현실에서 이렇게 추출된 feature vector의 차원은 단어 사전의 크기와 맞먹습니다. 단어 대신 문서를 feature로 사용하더라도 주어진 문서의 숫자만큼 vector의 차원이 만들어질 것입니다. vector가 큰 문제는 차치하고서라도, 더 큰 문제는 그 차원의 대부분의 값들이 0으로 채워져 있다는 것입니다. 즉, 각 차원에 숫자가 나타나는 경우는 0이 나타나는 경우에 비해서 현저히 적습니다. 따라서 이런 0으로 가득한 sparse vector를 통해 무엇인가 배우고자 할 때에 큰 장애로 작용합니다.

이러한 sparsity(희소성)문제는 자연어처리의 가장 큰 특징입니다. 단어는 discrete한 심볼로 이루어져 있기 때문입니다. 따라서 전통적인 자연어처리에서는 이러한 희소성 문제로 인해서 정말 큰 어려움을 겪었습니다. 사실 많은 분들이 잘 알고 있듯이, 요즘 딥러닝에서는 단어의 feature vector를 이런 sparse vector로 만들기보단 embedding 기법을 통해 dense vector로 만들어 사용 합니다. 이런 차이점이 바로 딥러닝에서의 자연어처리가 전통적인 방식의 자연어처리에 비해서 성능이 월등한 이유입니다. 이제 우리는 앞으로 다룰 챕터에서 딥러닝을 통해 이런 자연어처리의 문제를 잘 해결하는 방법들을 소개해보고자 합니다.

## Preprocessing

### Preprocessing

자연어처리에 있어서 전처리는 매우 중요합니다. 사실 어떻게 보면 image를 다루는 computer vision이나 눈에 보이지 않는 signal을 다루는 audio 또는 speech 분야에 비하면 데이터의 성격이 다루기 쉬워 보일 수 있습니다. 하지만, 오히려 discrete한 symbol로 이루어져 있기 때문에 continuous한 값들로 이루어진 다른 데이터의 형태에 비해 훨씬 더 정제의 중요성이 강조됩니다. 예를 들어 이미지는 1000x1000 image의 픽셀 하나의 색깔이 살짝 바뀌어도 그 이미지의 의미는 변화가 없다고 할 수 있지만, 문장에서의 단어 하나의 변화는 문장의 뉘앙스를 바꿀 수도 있고, 아예 문장의 의미 자체를 완전히 다른 것으로 만들어 버릴 수도 있습니다. 따라서, NLP에서의 전처리는 매우 중요한 의미를 가지고 있습니다.

## Corpus란?

Corpus(코퍼스)는 말뭉치라고 불리우기도 합니다. (복수 표현은 corpora) 보통 문장은 여러 단어들로 이루어져 있고, 이러한 문장들의 집합을 corpus라고 합니다. NLP분야의 machine learning을 수행하기 위해서는 훈련 데이터가 필요한데 보통 이런 다수의 문장들로 구성된 corpus가 필요합니다.

한가지 언어로 구성된 corpus는 monolingual corpus라고 부르며, 두개의 언어로 구성된 corpus는 bilingual corpus라고 부릅니다. 더 많은 숫자의 언어로 구성되면 multilingual corpus가 됩니다. 이때, 예를 들어 아래와 같이 언어간에 쌍으로 구성된 corpus를 parallel corpus라고 부릅니다.

English	Korean
I love to go to school.	나는 학교에 가는 것을 좋아한다.
I am a doctor.	나는 의사입니다.

이러한 corpus가 많을 수록, 오류가 없을 수록 우리는 NLP machine learning을 더욱 정교하게 할 수 있고, model의 정확도를 높힐 수 있습니다. 우리은 이 chapter에서 corpus를 수집하고 정제하여 효율성을 높힐 수 있도록하는 preprocessing에 대해서 다루어 보도록 하겠습니다.



Figure 1: Noam Chomsky – Image from Wikipedia

## Preprocess Overview

NLP에서의 전처리는 목적에 따라 약간씩 다르지만 대체로 아래와 같이 비슷한 과정을 지니고 있습니다. 이번 챕터에서는 이러한 과정에 대해서 다루고 넘어갈 것입니다.

1. 말뭉치(corpus) 수집
2. 말뭉치(corpus) 정제(normalization)
  1. Cleaning
  2. Sentence Tokenization
  3. Tokenization
  4. Subword segmentation

## Collecting Corpus

Corpus를 구하는 방법은 여러가지가 있습니다. Open된 데이터를 사용할 수도 있고, 구매를 할 수도 있습니다. Open된 데이터는 주로 각종 Task(e.g. Sentiment Analysis and Machine Translation)들의 대회 또는 논문을 위한 데이터입니다. 여기서는 crawling을 통한 수집을 주로 다루도록 하겠습니다. 다양한 웹사이트에서 crawling을 수행 할 수 있는만큼, 다양한 domain의 corpus를 모을 수 있습니다. 만약 특정 domain만을 위한 NLP task가 아니라면, 특정 domain에 편향(biased)되지 않도록 최대한 다양한 domain에서 corpus를 수집하는 것이 중요합니다.

하지만 무작정 웹사이트로부터 corpus를 crawling하는 것은 법적인 문제가 될 수 있습니다. 저작권 뿐만 아니라, 불필요한 traffic을 웹서버에 가중시킴으로써, 문제가 생길 수 있습니다. 따라서 올바른 방법으로 적절한 웹사이트에서 상업적인 목적이 아닌 경우에 제한된 crawling을 할 것을 권장합니다. 해당 웹사이트의 Crawling에 대한 허용 여부는 그 사이트의 robots.txt를 보면 확인 할 수 있습니다. 예를 들어 TED의 robot.txt는 다음과 같이 확인 할 수 있습니다.

```
$ wget https://www.ted.com/robots.txt
$ cat robots.txt
User-agent: *
Disallow: /latest
Disallow: /latest-talk
Disallow: /latest-playlist
Disallow: /people
```

```

Disallow: /profiles
Disallow: /conversations

User-agent: Baiduspider
Disallow: /search
Disallow: /latest
Disallow: /latest-talk
Disallow: /latest-playlist
Disallow: /people
Disallow: /profiles

```

모든 User-agent에 대해서 일부 경우에 대해서 disallow 인 것을 확인 할 수 있습니다. robots.txt에 대한 좀 더 자세한 내용은 <http://www.robotstxt.org/>에서 확인 할 수 있습니다.

Crawling 할 때에는 selenium이라는 package를 사용하여 web-browser driver를 직접 control하여 crawling을 수행하곤 합니다. 이때, phantomJS나 chrome과 같은 headless browser를 사용하면, 실제 사용자가 화면에서 보던 것과 같은 page를 다운로드 받을 수 있습니다. (예를 들어 wget과 같이 그냥 다운로드 할 경우, 브라우저와 다른 모양의 페이지가 다운로드 되는 경우도 많습니다.) 이후, beautiful-soup이라는 package를 사용하여 HTML 코드를 쉽고 간단하게 parsing할 수 있습니다.

## Monolingual Corpora

사실 가장 손 쉽게 구할 수 있는 종류의 corpus입니다. 경우에 따라서 Wikipedia나 각종 Wiki에서는 dump 데이터를 제공하기도 합니다. 따라서 해당 데이터를 다운로드 및 수집하는 것은 손쉽게 대량의 corpus를 얻을 수 있는 방법 중에 하나입니다. 아래는 domain에 따른 대표적인 corpus의 수집 방식입니다. 또한 Kaggle에서도 많은 종류의 dataset이 대량으로 upload되어 있으니 필요에 따라 다운로드 받아 사용하면 매우 유용합니다. Crawling을 수행 할 때에는 해당 사이트의 robots.txt 등을 확인하여 적법한 절차를 통해 crawling을 수행하길 권장 합니다.

문체	domain	수집처	정제 난이도
대화체	일반	채팅 로그	높음
대화체	일반	블로그	높음
문어체	시사	뉴스 기사	낮음

문체	domain	수집처	정제 난이도
문어체	과학, 교양, 역사 등	Wikipedia	중간
문어체	과학, 교양, 역사, 서브컬쳐 등	나무위키	중간
대화체	일반(각 분야별 게시판 존재)	클리앙	중간
문어체	일반, 시사 등	PGR21	중간
대화체	일반	드라마, 영화 자막	낮음

자막은 저작권이 있는 경우가 많기 때문에 저작권 관련 정보를 잘 확인 하는 것도 중요합니다.

## Multilingual Corpora

Machine Translation을 위한 parallel corpus를 구하는 것은 monolingual corpus에 비해서 상당히 어렵습니다. Crawling을 수행 할 때에는 해당 사이트의 robots.txt 등을 확인하여 적법한 절차를 통해 crawling을 수행하길 권장 합니다. 자막은 저작권이 있는 경우가 많기 때문에 저작권 관련 정보를 잘 확인 하는 것도 중요합니다.

문체	domain	수집처	정제 난이도	정렬 난이도
문어체	시사, 과학 등	OPUS	낮음	중간
대화체	시사, 교양, 과학 등	TED	낮음	중간
문어체	시사	중앙일보영자신문	낮음	중간
문어체	시사	동아일보영자신문	낮음	중간
문어체	일반	Korean Parallel Data	낮음	낮음
대화체	일반	드라마, 영화 자막	낮음	중간

자막을 parallel corpus로 사용할 경우 몇가지 문제점이 있습니다. 번역 품질의 저하로 인한 문제는 둘째치고, 'he'나 'she'같은 대명사가 사람 이름과 같은 고유명사로 표현되어 있는 경우도 많습니다. 따라서, 이러한 문제점들을 두루 고려해야 할 필요성이 있습니다.

## Speech with Transcript

Speech 데이터와 그와 함께 annotated된 transcript 데이터를 구하는 것은 정말 어렵습니다. 주로 연구용으로 공개 된 데이터들을 이용하거나 자막을 활용하여 영상에서 추출 하는 방법도 생각 해 볼 수 있습니다. # Cleaning

정제(normalization)는 텍스트를 사용하기에 앞서 필수적인 과정입니다. 원하는 Task에 따라, 또는 application에 따라서 필요한 정제의 수준 또는 깊이가 다를 수 있습니다. 예를 들어 음성인식을 위한 언어모델의 경우에는 사람의 음성을 그대로 받아적어야 하기 때문에, 괄호 또는 별표와 같은 기호나 특수문자들은 포함되어서는 안됩니다. 또한, 전화번호나 이메일 주소, 신용카드 번호와 같은 개인정보나 민감한 정보들은 제거되거나 변조된 채로 모델링 되야 할 수도 있습니다. 각 case에 따라서 필요한 형태를 얻어내기 위해서는 효과적인 정제 방법을 사용해야 합니다.

## 전각문자 제거

대부분의 중국어와 일본어 문서, 그리고 일부 한국어 문서들은 숫자, 영자, 기호가 전각문자로 되어 있는 경우가 있습니다. 이러한 경우에 일반적으로 사용되는 반각문자로 변환해 주는 작업이 필요합니다. 대표적으로 반각/전각문자로 혼용되는 문자들은 아래와 같습니다. 아래의 문자들을 각 문자에 해당하는 반각문자로 바꾸어주는 작업이 필요합니다.

" " ,

## 대소문자 통일

일부 영어 corpus에서는 또는 corpus마다 약자 등에서의 대소문자 표현이 통일되지 않은 경우가 있습니다. 예를 들어 New York City의 줄임말(약자)인 NYC의 경우에 아래와 같은 다양한 표현이 가능합니다.

번호	New York City
1	NYC
2	nyc
3	N.Y.C.
4	N.Y.C

따라서 이러한 다양한 표현을 일원화 시켜주는 것은 한개의 의미를 지니는 여러 단어의 형태를 하나로 통일시켜 줌으로써, Sparsity를 감소시키는 효과를 거둘 수 있습니다. 하지만, deep learning에 접어들어 word embedding을 통한 효과적인 표현이 가능해지면서, 다양한 표현을 비슷한 값의 vector로 나타낼 수 있게 되어, 대소문자 통일과 같은 작은 문제(전체 corpus에서 차지하는 비율이 적은 문제)에 대한 해결 필요성이 줄어들었습니다.

## Regular Expression

또한, crawling을 통해 얻어낸 다양한 corpus는 보통 특수문자, 기호 등에 의해서 noise가 섞여 있는 경우가 많습니다. 또한, 웹사이트의 성격에 따라 일정한 패턴을 띠고 있는 경우도 많습니다. 이러한 noise들을 효율적으로 감지하고 없애기 위해서는 regular expression (정규식)의 사용은 필수적입니다. 따라서, 이번 section은 regular expression(regex)에 대해서 살펴 봅니다.

### [ ]의 사용

[2345cde]

(2|3|4|5|c|d|e)

### -의 사용

[2-5c-e]

### [^ ]의 사용

[^2-5c-e]

### ( )의 사용

(x)(yz)

One of:

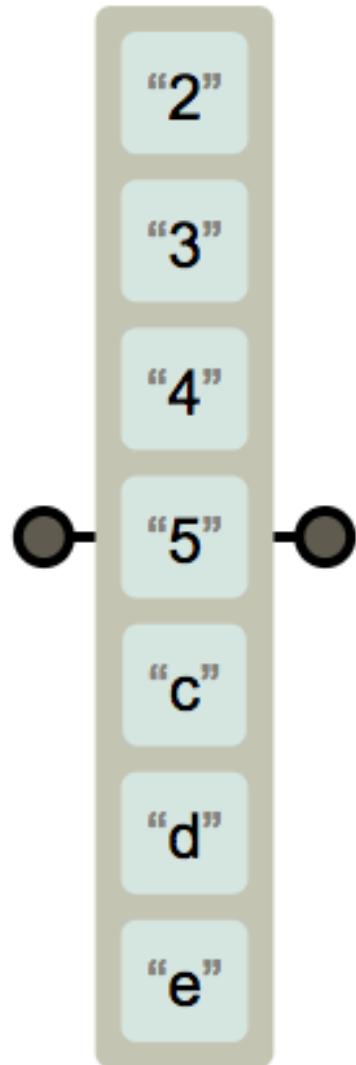


Figure 2:

**One of:**

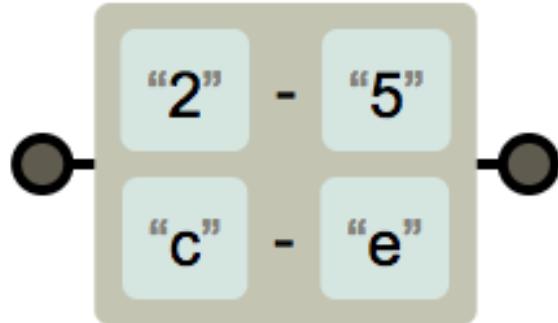


Figure 3:

**None of:**

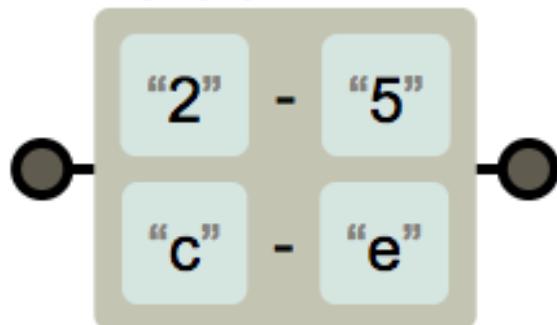


Figure 4:

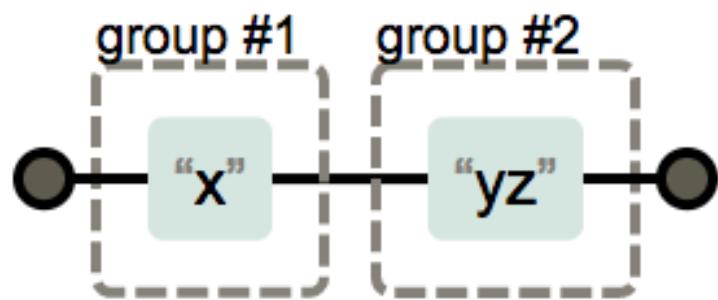


Figure 5:

|의 사용

$(x|y)$

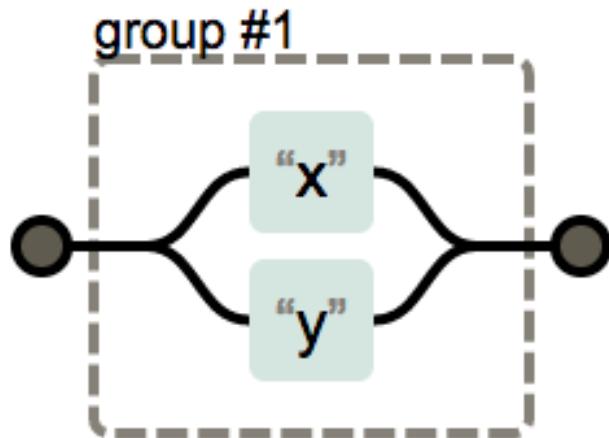


Figure 6:

?, \*, +의 사용

?는 앞의 수식하는 부분이 나타나지 않거나 한번만 나타날 경우 사용 합니다.

$x?$

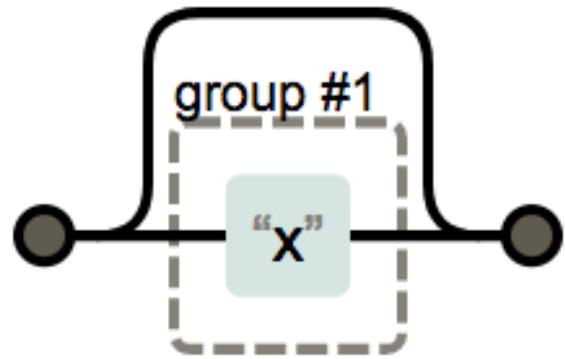


Figure 7:

$+$ 는 앞의 수식하는 부분이 한 번 이상 나타날 경우 사용 합니다.

$x^+$

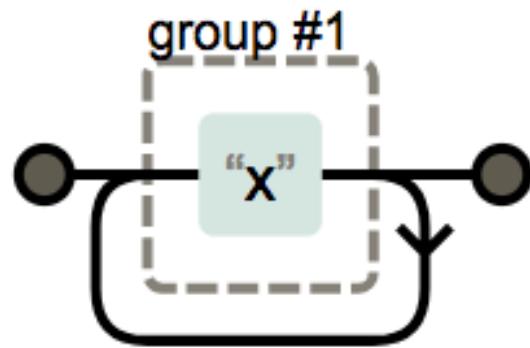


Figure 8:

$*$ 는 앞의 수식하는 부분이 나타나지 않거나 여러번 나타날 경우 사용 합니다.

$x^*$

$\{n\}$ ,  $\{n,\}$ ,  $\{n,m\}$ 의 사용

$x^{\{n\}}$

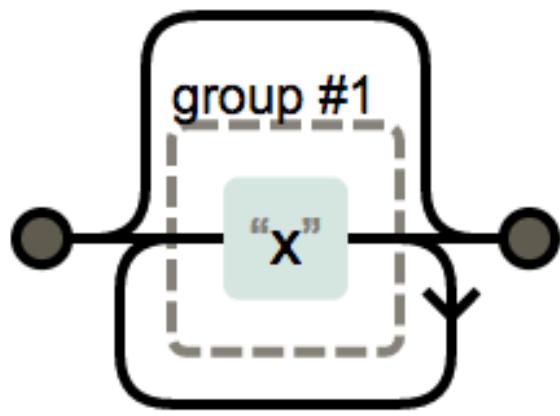


Figure 9:

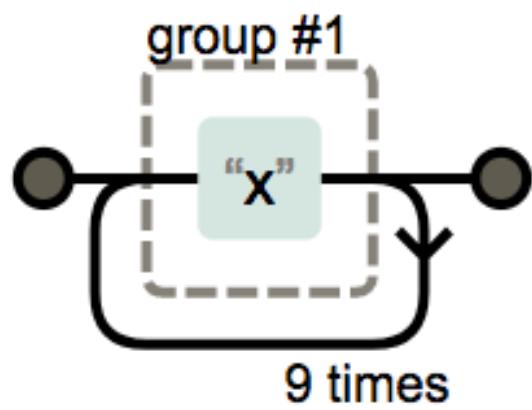


Figure 10:

$x\{n,\}$

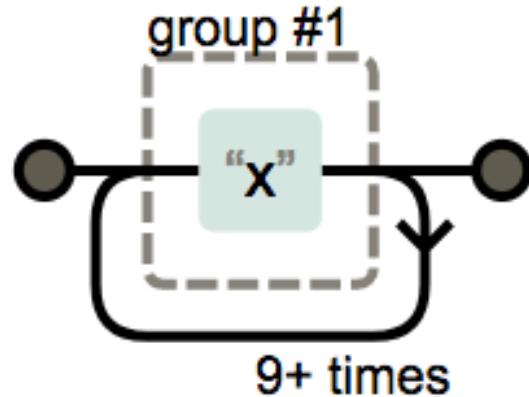


Figure 11:

$x\{n,m\}$

. 의 사용

.

^와 \$의 사용

$^x\$$

지정문자의 사용

Meta Characters	Description
Ws	공백문자(white space)
WS	공백문자를 제외한 모든 문자
Ww	alphanumeric(알파벳 + 숫자) + '_' ([A-Za-z0-9_]와 같음)
WW	non-alphanumeric 문자 '_'도 제외 ([^A-Za-z0-9_]와 같음)
Wd	숫자 ([0-9]와 같음)

Meta Characters	Description
WD	숫자를 제외한 모든 문자 ([^0-9]와 같음)

### Example

실제 예를 들어 보겠습니다. NLP 문제를 풀고 있는 중에, 문서의 마지막 줄에 종종 아래와 같은 개인의 전화번호 정보가 포함되어 있는 문서를 dataset으로 사용하려 할 때, 해당 정보를 제외하고 사용하고 싶다고 가정해 보겠습니다.

Hello Ki,  
I would like to introduce regular expression in this section.  
~~  
Thank you!  
Sincerely,  
Ki: +82-10-1234-5678

무턱대고 마지막 줄을 지우기에는 마지막 줄에 전화번호 정보가 없는 경우도 많기 때문에 선택적으로 지워야 할 것 같습니다. 따라서 데이터를 쭈욱 훑어가며 살펴보니, 마지막 줄은 아래와 같은 규칙을 따르는 것 같습니다.

- 이름이 전화번호 앞에 나올 수도 있다.
- 이름 뒤에는 콜론(:)이 나올 수도 있다.
- 콜론 앞/뒤로는 공백(tab 포함)이 다수가 존재할 수도 있다.
- 전화번호는 국가번호를 포함할 수도 있다.
- 국가번호는 최대 3자리이다.
- 국가번호의 앞에는 '+'가 붙을 수도 있다.
- 전화번호 사이에 '-'가 들어갈 수도 있다.
- 전화번호는 빈칸이 없이 표현 된다.
- 전화번호의 맨 앞과 지역번호(또는 010)의 다음에는 괄호가 들어갈 수도 있다.
- 괄호는 한쪽만 나올 수도 있다.
- 지역번호 자리의 맨 처음 나오는 0은 빠질 수도 있다. 즉, 2자리가 될 수도 있다.
- 지역번호 다음 번호 그룹은 3에서 4자리 숫자이다.
- 마지막은 항상 4자리 숫자이다.

위의 규칙을 따르는 regular expression을 표현하면 아래와 같습니다.

([\w]+|\s\*:\?\s\*)?\(\?\+\?([0-9]{1,3})\)?\-\?[0-9]{2,3}(\)|\-\?)?[0-9]{3,4}\-\?[0-9]{4}

위의 수식을 그림으로 표현하면 아래와 같습니다.

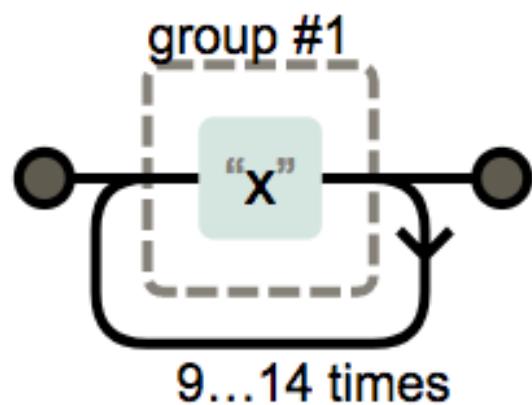


Figure 12:



Figure 13:

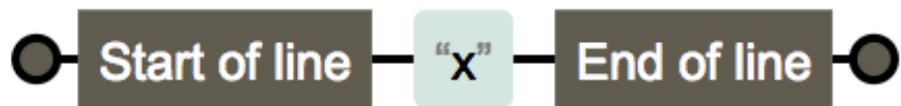
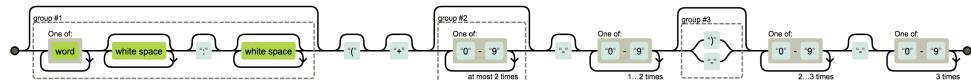


Figure 14:



[Image from [regexpex.com](http://regexpex.com)]

## Python에서의 Regular Expression

```
>>> import re
>>> regex = r"([\w]+\s*:?\s*)?\((\?|\+|([0-9]{1,3}))?\-?[0-9]{2,3}(\)|\-)?[0-9]{3,4}\-?"
>>> x = "Ki: +82-10-9420-4104"
>>> re.sub(regex, "REMOVED", x)
'REMOVED'
>>> x = "CONTENT jiu 02)9420-4104"
>>> re.sub(regex, "REMOVED", x)
'CONTENT REMOVED'
```

`re.sub`

`r""`

## ₩1, ₩2, … 치환자의 사용

이제까지 다른 정규식 표현만으로도 많은 부분을 cover할 수 있지만, 아직 2% 부족함이 남아 있습니다. 예를 들어 아래와 같은 case를 다루어 보겠습니다.

알파벳(소문자) 사이에 있는 숫자를 제거하라.

```
abcdefg
12345
ab12
a1bc2d
12ab
a1b
1a2
a1
1a
hijklmnop
```

만약 그냥  $[0-9]^+$ 으로 숫자를 찾아서 없애면 두번째 줄의 숫자만 있는 경우와 숫자가 가장자리에 있는 경우도 사라지게 됩니다. 그럼 어떻게 해야 할까요? 이때 유용한 방법이 치환자를 사용하는 것입니다.

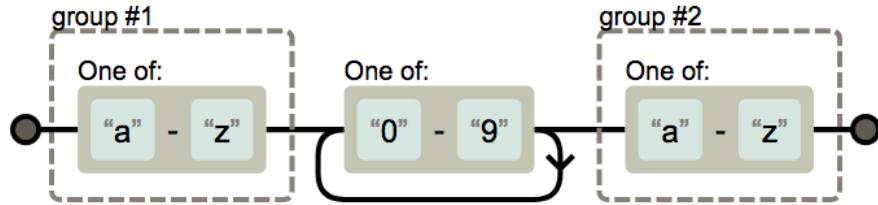


Figure 15:

괄호로 묶인 부분은 group으로 지정이 되고, 바뀔 문자열 내에서 역슬래시(\)와 함께 숫자로 가리킬 수 있습니다. 예를 들어 아래와 같이 구현 할 수 있습니다.

```
x = '''abcdefg
12345
ab12
a1bc2d
12ab
a1b
1a2
a1
1a
hijklmnop'''
```

```
regex = r'([a-z])[0-9]+([a-z])'
to = r'\1\2'
```

```
y = '\n'.join([re.sub(regex, to, x_i) for x_i in x.split('\n')])
```

위의 방법은 굳이 python과 같은 programming 언어가 아니더라도 sublime text와 같이 지원되는 text editor들이 있으니 editor 상에서의 정제를 할 때에도 유용하게 사용할 수 있습니다.

## Sentence Tokenization

우리가 다루고자 하는 task들은 입력 단위가 문장단위인 경우가 많습니다. 즉, 한 line에 한 문장만 있어야 합니다. 여러 문장이 한 line에 있거나, 한 문장이 여러 line에 걸쳐 있는 경우에 대해서 sentence tokenization이 필요합니다. 단순이 마침표만을 기준으로 sentence tokenization을 수행하게 되면, U.S. 와 같은 영어 약자나 3.141592와 같은 소숫점 등 여러가지 문제점에 마주칠 수 있습니다. 따라서 직접 tokenization해주기 보다는 NLTK에서 제공하는 tokenizer를 이용하기를 권장합니다. – 물론, 이 경우에도 완벽하게 처리하지는 못하며, 일부 추가적인 전/후 처리가 필요할 수도 있습니다.

## Tokenization

한 line에 여러 문장이 들어 있는 경우에 대한 Python script 예제.

before:

```
TED      1,000      TED      .      --      .  
  
import sys, fileinput, re  
from nltk.tokenize import sent_tokenize  
  
if __name__ == "__main__":  
    for line in fileinput.input():  
        if line.strip() != "":  
            line = re.sub(r'([a-z])\.( [A-Z])', r'\1. \2', line.strip())  
  
            sentences = sent_tokenize(line.strip())  
  
            for s in sentences:  
                if s != "":  
                    sys.stdout.write(s + "\n")
```

after:

```
TED      1,000      TED      .  
--      .  
.      .
```

```
1,000      ,      1,000  
1,000      ?
```

## Combine and Tokenization

여러 line에 한 문장이 들어 있는 경우에 대한 Python script 예제.

before:

```
TED      1,000      TED      .  
--  
.      .  
1,000      ,  
1,000      .
```

```
1,000      ?  
1,000  TED
```

```
import sys, fileinput  
from nltk.tokenize import sent_tokenize  
  
if __name__ == "__main__":  
    buf = []  
  
    for line in fileinput.input():  
        if line.strip() != "":  
            buf += [line.strip()]  
            sentences = sent_tokenize(" ".join(buf))  
  
            if len(sentences) > 1:  
                buf = sentences[1:]  
  
                sys.stdout.write(sentences[0] + '\n')  
  
    sys.stdout.write(" ".join(buf) + "\n")
```

after:

```

TED      1,000    TED
--      .
.

1,000 , 1,000 .
1,000 ? 
1,000 TED

```

## Part of Speech Tagging, Tokenization (Segmentation)

우리가 하고자 하는 task에 따라서 Part-of-speech (POS) tagging 또는 단순한 segmentation을 통해 normalization을 수행합니다. 주로 띠어쓰기에 대해서 살펴보도록 하겠습니다. 아래에 정리한 프로그램들이 기본적으로 띠어쓰기만 제공하는 프로그램이 아니더라도, tag정보 등을 제외하게 되면 띠어쓰기(segmentation)를 수행하는 것과 동일합니다.

띠어쓰기(tokenization or segmentation)에 대해서 살펴 보겠습니다. 한국어, 영어와 달리 일본어, 중국어의 경우에는 띠어쓰기가 없는 경우가 없습니다. 또한, 한국어의 경우에도 띠어쓰기가 도입된 것은 근대에 이르러서이기 때문에, 띠어쓰기의 표준화가 부족하여 띠어쓰기가 중구난방인 경우가 많습니다. 특히나, 한국어의 경우에는 띠어쓰기가 문장의 의미 해석에 큰 영향을 끼치지 않기 때문에 더욱이 이런 현상은 가중화 됩니다. 따라서, 한국어의 경우에는 띠어쓰기가 이미 되어 있지만, 제각각인 경우가 많기 때문에 normalization을 해주는 의미로써 다시한번 적용 됩니다. 또한, 교착어로써 접사를 어근에서 분리해주는 역할도 하여, sparseness를 해소하는 역할도 합니다. 이러한 한국어의 띠어쓰기 특성은 앞서 Why Korean NLP is Hell에서 다루었습니다.

일본어, 중국어의 경우에는 띠어쓰기가 없기 때문에, 애초에 모든 문장들이 같은 띠어쓰기 형태(띠어쓰기가 없는 형태)를 띠고 있지만, 적절한 language model 구성을 위하여 띠어쓰기가 필요합니다. 다만, 중국어의 경우에는 character 단위로 문장을 실제 처리하더라도 크게 문제가 없기도 합니다.

영어의 경우에는 기본적으로 띠어쓰기가 있고, 대부분의 경우 매우 잘 표준을 따르고 있습니다. 다만 language model을 구성함에 있어서 좀 더 용이하게 해 주기 위한 일부 처리들을 해주면 더 좋습니다. NLTK를 사용하여 이러한 전처리를 수행합니다.

각 언어별 주요 프로그램들을 정리하면 아래와 같습니다.

## 언어프로세서개발

한국 Mecab + 일본어

Mecab을  
wrap-  
ping하였으며,  
속도가  
가장  
빠름.  
설치가  
종종  
까다로움

한국 KoNLPyParser을

Wrap하는(복합)  
설치  
가능하며  
사용이  
쉬우나,  
일부

tag-  
ger의  
경우에는  
속도가  
느림

일본 MeCab + 속도가

빠르다

중국 Stanford 미국

Parser 스탠포드에서  
개발

중국 JKU Java 북경대에서

Parser 개발.  
Stan-  
ford  
Parser와  
성능  
차이가  
거의  
없음

## 언어프로세서개발

중국JiebaPython장

최근에

개발됨.

Python으로

제작되어

시스템

구성에

용이

사실 일반적인 또는 전형적인 쉬운 문장(표준어를 사용하며 문장 구조가 명확한 문장)의 경우에는 대부분의 프로그램들의 성능이 비슷합니다. 하지만 각 프로그램 성능의 차이를 만드는 것은 바로 신조어나 보지 못한 고유명사를 처리하는 능력입니다. 따라서 어떤 한 프로그램을 선택하고자 할때에, 그런 부분에 초점을 맞추어 성능을 평가하고 선택해야 할 필요성이 있습니다.

## Korean

### Mecab

한국어 tokenization에 가장 많이 사용되는 프로그램은 Mecab입니다. 원래 Mecab은 일본어 tokenizer 오픈소스로 개발되었지만, 이를 한국어 tokenizing에 성공적으로 적용시켜 널리 사용되고 있습니다. 아래와 같이 설치 가능합니다. – 참고사이트: <http://konlpy-ko.readthedocs.io/ko/v0.4.3/install/#ubuntu>

```
$ sudo apt-get install curl  
$ bash <(curl -s https://raw.githubusercontent.com/konlpy/konlpy/master/scripts/me
```

위의 instruction을 따라 정상적으로 설치 된 Mecab의 경우에는, KoNLPy에서 불러 사용할 수 있습니다. – 참고사이트: <http://konlpy-ko.readthedocs.io/ko/v0.4.3/api/konlpy.tag/#mecab-class>

또는 아래와 같이 bash상에서 mecab명령어를 통해서 직접 실행 가능하며 standard input/output을 사용하여 tokenization을 수행할 수 있습니다.

```
$ echo " , ! " | mecab  
NNG,* ,T, ,*,*,*,*
```

```

XSV,*,F, ,*,*,*,*
EP+EF,*,F, ,Inflect,EP,EF, /EP/*+ /EF/*
, SC,*,*,*,*,*,*
VA,*,T, ,*,*,*,*
EF,*,F, ,*,*,*,*
! SF,*,*,*,*,*,*
EOS

```

만약, 단순히 띄어쓰기만 적용하고 싶을 경우에는 -O wakati 옵션을 사용하여 실행시키면 됩니다. 마찬가지로 standard input/output을 통해 수행 할 수 있습니다.

```
$ echo " , ! " | mecab -O wakati
,
```

위와 같이 기본적으로 어근과 접사를 분리하는 역할을 수행하며, 또는 잘못된 띄어쓰기에 대해서도 올바르게 교정을 해 주는 역할 또한 수행 합니다. 어근과 접사를 분리함으로써 어휘의 숫자를 효과적으로 줄여 sparsity를 해소할 수 있습니다.

## KoNLPy

KoNLPy는 여러 한국어 tokenizer 또는 tagger들을 모아놓은 wrapper를 제공합니다. 이름에서 알 수 있듯이 Python wrapper를 제공하므로 시스템 연동 및 구성이 용이할 수 있습니다. 하지만 내부 라이브러리들은 각기 다른 언어(Java, C++ 등)로 이루어져 있어 호환성에 문제가 생기기도 합니다. 또한, 일부 library들은 상대적으로 제작/업데이트가 오래되었고, java로 구현되어 있어 C++로 구현되어 있는 Mecab에 비해 속도면에서 불리함을 갖고 있습니다. 따라서 대용량의 corpus를 처리할 때에 불리하기도 합니다. 하지만 설치 및 사용이 쉽고 다양한 library들을 모아놓았다는 장점 때문에 사랑받고 널리 이용되고 있습니다.

참고사이트: <http://konlpy-ko.readthedocs.io/>

## English

### Moses

영어의 경우에는 위에서 언급했듯이, 기본적인 띄어쓰기가 잘 통일되어 있는 편이기 때문에, 띄어쓰기 자체에 대해서는 큰 normalization 이슈가 없습니다. 다만 comma, period, quotation 등을 띄어주어야 합니다. Moses에서 제공하는 tokenizer를

통해서 이런 처리를 수행하게 됩니다. NLTK 예전 버전(3.2.5)에서는 Moses를 랩핑(wrapping)하여 tokenizer를 제공하였습니다. 3.2.5버전의 NLTK를 설치하기 위한 pip 명령어는 다음과 같습니다.

```
$ pip install nltk==3.2.5
```

아래는 NLTK를 사용한 Python script 예제 입니다.

before: >North Korea's state mouthpiece, the Rodong Sinmun, is also keeping mum on Kim's summit with Trump while denouncing ever-tougher U.S. sanctions on the rogue state.

```
import sys, fileinput
from nltk.tokenize.moses import MosesTokenizer

t = MosesTokenizer()

if __name__ == "__main__":
    for line in fileinput.input():
        if line.strip() != "":
            tokens = t.tokenize(line.strip(), escape=False)

            sys.stdout.write(" ".join(tokens) + "\n")
```

after: >North Korea 's state mouthpiece , the Rodong Sinmun , is also keeping mum on Kim 's summit with Trump while denouncing ever-tougher U.S. sanctions on the rogue state .

## Chinese

### Stanford Parser

스탠포드 대학교에서 제작한 중국어 parser입니다. Java로 제작되었습니다.

참고사이트: <https://nlp.stanford.edu/software/lex-parser.shtml>

## JIEBA

비교적 가장 늦게 개발/업데이트가 이루어진 Jieba library는, 사실 성능상에서 다른 parser들과 큰 차이가 없지만, python으로 구현되어 있어 우리가 실제 상용화를 위한 배치 시스템을 구현 할 때에 매우 쉽고 용이할 수 있습니다. 특히, 딥러닝을 사용한 머신러닝 시스템은 대부분 Python위에서 동작하기 때문에 일관성 있는 시스템을 구성하기 매우 좋습니다.

참고사이트: <https://github.com/fxsjy/jieba # Align Parallel Corpus>

대부분의 parallel corpora는 여러 문장 단위로 align이 되어 있는 경우가 많습니다. 이러한 경우에는 한 문장씩에 대해서 align을 해주어야 합니다. 또한, 이러한 과정에서 일부 parallel 하지 않은 문장들을 걸러내야 하고, 문장 간 align이 잘 맞지 않는 경우 align을 재정비 해 주거나 아예 걸러내야 합니다. 이러한 과정에 대해서 살펴 봅니다.

## Process Overview for Parallel Corpus Alignment

Alignment를 수행하기 위한 전체 과정을 요약하면 아래와 같습니다.

1. Building a dictionary between source language to target language.
  1. Collect and normalize (clean + tokenize) corpus for each language.
  2. Get word embedding vector for each language.
  3. Get word-level-translator using MUSE.
2. Align collected semi-parallel corpus using Champollion.
  1. Sentence-tokenize for each language.
  2. Normalize (clean + tokenize) corpus for each language.
  3. Align parallel corpus using Champollion.

## Building Dictionary

만약 기존에 단어 (번역) 사전을 구축해 놓았다면 그것을 이용하면 되지만, 단어 사전을 구축하는 것 또한 비용이 들기 때문에, 일반적으로는 쉽지 않습니다. 따라서, 단어 사전을 구축하는 것 또한 자동으로 할 수 있습니다.

Facebook의 MUSE는 parallel corpora가 없는 상황에서 사전을 구축하는 방법을 제시하고, 코드를 제공합니다. 각 Monolingual corpus를 통해 구축한 언어 별 word embedding vector에 대해서 다른 언어의 embedding vector와 mapping하도록 함으로써, 단어 간 번역을 할 수 있습니다. 이는 각 언어 별 corpus가 많을 수록

embedding vector가 많을수록 더욱 정확하게 수행 됩니다. MUSE는 parallel corpora가 없는 상황에서도 수행 할 수 있기 때문에 unsupervised learning이라고 할 수 있습니다.

아래는 MUSE를 통해 구한 영한 사전의 일부로써, 꽤 정확한 단어 간의 번역을 볼 수 있습니다. 이렇게 구성한 사전은 champollion의 입력으로 사용되어, champollion은 이 사전을 바탕으로 parallel corpus의 sentence alignment를 수행합니다. ◇을 delimiter로 사용하여 한 라인에 source 언어의 단어와 target 언어의 단어를 표현 합니다.

```
stories <>
stories <>
contact <>
contact <>
contact <>
green <>
green <>
green <>
dark <>
dark <>
dark <>
song <>
song <>
song <>
salt <>
```

## Align via Champollion

Chapollion Toolkit(CTK)는 parallel corpus의 sentence alignment를 수행하는 open-source입니다. Perl을 사용하여 구현되었으며, 이집트 상형문자를 처음으로 해독해낸 역사학자 Champollion의 이름을 따서 명명되었습니다.



(Jean-François Champollion, Image from Wikipedia)

기-구축된 단어 (번역) 사전을 이용하거나, 위와 같이 자동으로 구축한 단어 사전을 참고하여 Champollion은 sentence alignment를 수행합니다. 여러 line으로 구성된 각 언어 별 하나의 document에 대해서 sentence alignment를 수행한 결과는 아래와 같습니다.

```
omitted <=> 1
omitted <=> 2
omitted <=> 3
1 <=> 4
2 <=> 5
3 <=> 6
4,5 <=> 7
6 <=> 8
7 <=> 9
8 <=> 10
9 <=> omitted
```

위의 결과를 해석해 보면, target 언어의 1, 2, 3번째 문장은 짹을 찾지 못하고 버려졌고, source 언어의 1, 2, 3번째 문장은 각각 target 언어의 4, 5, 6번째 문장과 mapping 된 것을 알 수 있습니다. 또한, source 언어의 4, 5번째 두 문장은 target 언어의 7번 문장에 동시에 mapping 된 것을 볼 수 있습니다.

이와 같이 어떤 문장들은 버려지기도 하고, 일대일 mapping이 이루어지기도 하며,

일대다, 다대일 mapping이 이루어지기도 합니다.

아래는 champollion을 쉽게 사용하기 위한 Python script 예제입니다. CTK\_ROOT에 Chapollion Toolkit의 위치를 지정하여 사용할 수 있습니다.

```
import sys, argparse, os

BIN = CTK_ROOT + "/bin/champollion"
CMD = "%s -c %f -d %s %s %s %s"
 OMIT = "omitted"
INTERMEDIATE_FN = "./tmp/tmp.txt"

def read_alignment(fn):
    aligns = []

    f = open(fn, 'r')

    for line in f:
        if line.strip() != "":
            srcs, tgts = line.strip().split(' <=> ')
            if srcs == OMIT:
                srcs = []
            else:
                srcs = list(map(int, srcs.split(',')))
            if tgts == OMIT:
                tgts = []
            else:
                tgts = list(map(int, tgts.split(',')))
            aligns += [(srcs, tgts)]
    f.close()

    return aligns

def get_aligned_corpus(src_fn, tgt_fn, aligns):
    f_src = open(src_fn, 'r')
```

```

f_tgt = open(tgt_fn, 'r')

for align in aligns:
    srcs, tgts = align

    src_buf, tgt_buf = [], []
    for src in srcs:
        src_buf += [f_src.readline().strip()]
    for tgt in tgts:
        tgt_buf += [f_tgt.readline().strip()]

    if len(src_buf) > 0 and len(tgt_buf) > 0:
        sys.stdout.write("%s\t%s\n" % (" ".join(src_buf), " ".join(tgt_buf)))

f_tgt.close()
f_src.close()

def parse_argument():
    p = argparse.ArgumentParser()

    p.add_argument('--src', required = True)
    p.add_argument('--tgt', required = True)
    p.add_argument('--src_ref', default = None)
    p.add_argument('--tgt_ref', default = None)
    p.add_argument('--dict', required = True)
    p.add_argument('--ratio', type = float, default = 1.2750)

    config = p.parse_args()

    return config

if __name__ == "__main__":
    config = parse_argument()

    if config.src_ref is None:
        config.src_ref = config.src
    if config.tgt_ref is None:

```

```

config.tgt_ref = config.tgt

cmd = CMD % (BIN, config.ratio, config.dict, config.src_ref, config.tgt_ref, )
os.system(cmd)

aligns = read_alignment(INTERMEDIATE_FN)
get_aligned_corpus(config.src, config.tgt, aligns)

```

특기할 점은 ratio parameter의 역할입니다. 이 parameter는 실제 champollion의 -c 옵션으로 mapping되어 사용되는데, champollion 상에서의 설명은 다음과 같습니다.

```

$ ./champollion
usage: ./champollion [-hdscn] <X token file> <Y token file> <alignment file>

-h      : this (help) message
-d dictf : use dictf as the translation dictionary
-s xstop : use words in file xstop as X stop words
-c n    : number of Y chars for each X char
-n      : disallow 1-3, 3-1, 1-4, 4-1 alignments
          (faster, lower performance)

```

즉, source language의 character 당 target language의 character 비율을 의미합니다. 이를 기반하여 champollion은 문장 내 모든 단어에 대해서 번역 단어를 모르더라도 sentence alignment를 수행할 수 있게 됩니다. # Segmentation using Subword (Byte Pair Encoding, BPE)

Byte Pair Encoding (BPE) 알고리즘을 통한 subword segmentation은 [Sennrich et al. 2015]에서 처음 제안되었고, 현재는 주류 전처리 방법으로 자리잡아 사용되고 있습니다.

Subword segmentation은 기본적으로 단어는 여러 sub-word들의 조합으로 이루어진다는 가정 하에 적용 되는 알고리즘입니다. 실제로 영어나 한국어의 경우에도 latin어와 한자를 기반으로 형성 된 언어이기 때문에, 많은 단어들이 sub-word들로 구성되어 있습니다. 따라서 적절한 subword detection을 통해서 subword 단위로 쪼개어 주면 어휘수를 줄일 수 있고 sparsity를 효과적으로 감소시킬 수 있습니다.

---

언어	단어	조합
영어	Concentrate	con(=together) + centr(=center) + ate(=make)

---

언어	단어	조합
한국어	집중(□□)	□(모을 집) + □(가운데 중)

Sparsity 감소 이외에도, 가장 대표적인 subword segmentation으로 얻는 효과는 unknown(UNK) token에 대한 효율적인 대처입니다. Natrual Language Generation (NLG)를 포함한 대부분의 딥러닝 NLP 알고리즘들은 문장을 입력으로 받을 때 단순히 word sequence로써 받아들이게 됩니다. 따라서 UNK가 나타나게 되면 이후의 language model의 확률은 매우 망가져버리게 됩니다. 따라서 적절한 문장의 encoding 또는 generation이 어렵습니다. – 특히 문장 generation의 경우에는 이전 단어를 기반으로 다음 단어를 예측하기 때문에 더더욱 어렵습니다.

하지만 subword tokenization을 통해서 신조어나 typo(오타) 같은 UNK에 대해서 subword나 character 단위로 쪼개줌으로써 known token들의 조합으로 만들어버릴 수 있습니다. 이로써, UNK 자체를 없앰으로서 효율적으로 UNK에 대처할 수 있고, NLP 결과물의 품질을 향상시킬 수 있습니다.

## Example

아래와 같이 BPE를 적용하면 원래의 띠어쓰기 공백 이외에도 BPE의 적용으로 인한 공백이 추가됩니다. 따라서 원래의 띠어쓰기와 BPE로 인한 띠어쓰기를 구분해야 할 필요성이 있습니다. 그래서 특수문자(기존의 \_가 아닌 □)를 사용하여 기존의 띠어쓰기(또는 단어의 시작)를 나타냅니다. 이를 통해 후에 설명할 detokenization에서 문장을 BPE이전의 형태로 원상복구 할 수 있습니다.

한글 Mecab에 의해 tokenization 된 원문 현재 TED 웹 사이트에는 1,000 개 가 넘는 TED 강연들이 있습니다. 여기 계신 여러분의 대다수는 정말 대단한 일이라고 생각하시겠죠 – 전 다릅니다. 전 그렇게 생각하지 않아요. 저는 여기 한 가지 문제점이 있다고 생각합니다. 왜냐하면 강연이 1,000 개라는 것은, 공유할 만한 아이디어들이 1,000 개 이상이라는 뜻이 되기 때문이죠. 도대체 무슨 수로 1,000 개나 되는 아이디어를 널리 알릴 건가요? 1,000 개의 TED 영상 전부를 보면서 그 모든 아이디어들을 머리 속에 집어 넣으려고 해도, 250 시간 이상의 시간이 필요할 겁니다. 250 시간 이상의 시간이 필요할 겁니다. 간단한 계산을 해봤는데요. 정말 그렇게 하는 경우 1인당 경제적 손실은 15,000 달러 정도가 됩니다.

한글 BPE 적용 □현재 □TED □웹 □사이트 □에 □는 □1 □, □000 □개 □가 □넘 □는 □TED □강 연 □들 □이 □있 □습니다 □. □여기 □계 신 □여러분

□의 □대 다 수 □는 □정말 □대단 □한 □일 □이 □라고 □생각 □하 □시 □겠  
□죠 □- □전 □다 릅니다 □. □전 □그렇게 □생각 □하 □지 □않 □아요 □.  
□저 □는 □여기 □한 □가지 □문 제 점 □이 □있 □다고 □생각 □합니다 □.  
□왜냐하면 □강 연 □이 □1 □, □000 □개 □라는 □것 □은 □, □공유 □할 □만  
□한 □아이디어 □들 □이 □1 □, □000 □개 □이상 □이 □라는 □뜻 □이 □되  
□기 □때문 □이 □죠 □. □도 대체 □무슨 □수로 □1 □, □000 □개 □나 □되  
□는 □아이디어 □를 □널 리 □알 릴 □건 가요 □? □1 □, □000 □개 □의 □TED  
□영상 □전부 □를 □보 □면서 □그 □모든 □아이디어 □들 □을 □머리 □속 □에  
□집 □어 □넣 □으 려고 □해도 □, □2 50 □시간 □이상 □의 □시간 □이 □필요  
□할 □겁니다 □. □2 50 □시간 □이상 □의 □시간 □이 □필요 □할 □겁니다  
□. □간단 □한 □계산 □을 □해 □봤 □는데요 □. □정말 □그렇게 □하 □는  
□경우 □1 □인 □당 □경제 □적 □손 실 □은 □15 □, □000 □달러 □정도 □가  
□됩니다 □.

영어 NLTK에 의해 tokenization이 된 원문 There's currently over a thousand TED Talks on the TED website . And I guess many of you here think that this is quite fantastic , except for me , I don't agree with this . I think we have a situation here . Because if you think about it , 1,000 TED Talks , that's over 1,000 ideas worth spreading . How on earth are you going to spread a thousand ideas ? Even if you just try to get all of those ideas into your head by watching all those thousand TED videos , it would actually currently take you over 250 hours to do so . And I did a little calculation of this . The damage to the economy for each one who does this is around \$ 15,000 . So having seen this danger to the economy , I thought , we need to find a solution to this problem . Here's my approach to it all .

영어 BPE 적용 □There □'s □currently □over □a □thous and □TED □T al ks  
□on □the □TED □we b site □. □And □I □guess □many □of □you □here  
□think □that □this □is □quite □f ant as tic □, □ex cept □for □me □, □I  
□don □'t □agree □with □this □. □I □think □we □have □a □situation  
□here □. □Because □if □you □think □about □it □, □1 ,000 □TED □T al  
ks □, □that □'s □over □1 ,000 □ideas □worth □sp reading □. □How □on  
□earth □are □you □going □to □spread □a □thous and □ideas □? □Even  
□if □you □just □try □to □get □all □of □those □ideas □into □your □head  
□by □watching □all □those □thous and □TED □vide os □, □it □would  
□actually □currently □take □you □over □2 50 □hours □to □do □so □.  
□And □I □did □a □little □cal cu lation □of □this □. □The □damage □to  
□the □economy □for □each □one □who □does □this □is □around □\$ □15  
,000 □. □So □having □seen □this □dang er □to □the □economy □, □I  
□thought □, □we □need □to □find □a □solution □to □this □problem □.

□Here □'s □my □approach □to □it □all □.

원문	subword segmentation
대다수	대 + 다 + 수
문제점	문 + 제 + 점
건가요	건 + 가요
website	we + b + site
except	ex + cept
250	2 + 50
15,000	15 + ,000

위와 같이 subword segmentation에 의해서 추가적으로 쪼개진 단어들은 적절한 subword의 형태로 나누어진 것을 볼 수 있습니다. 특히, 숫자 같은 경우에는 자주 쓰이는 숫자 50이나, 000의 경우 성공적으로 하나의 subword로 지정된 것을 볼 수 있습니다.  
# Detokenization

아래는 preprocessing에서 tokenization을 수행하고, 다시 복원(detokenization)하는 과정을 설명 한 것입니다. 하나씩 따라가보도록 하겠습니다.

아래와 같은 영어 원문이 주어지면,

There's currently over a thousand TED Talks on the TED website.

각 언어 별 tokenizer(영어의 경우 NLTK)를 통해 tokenization을 수행하고, 기존의 띄어쓰기와 tokenization에 의해 수행된 공백과의 구분을 위해 □을 원래의 공백 위치에 삽입합니다.

There 's currently over a thousand TED Talks on the TED website .

여기서 subword segmentation을 수행하며 이전 step까지의 공백과 subword segmentation으로 인한 공백을 구분하기 위한 □를 삽입합니다.

There 's currently over a thou sand TED T al ks on the TED we b site

이렇게 preprocessing 과정이 종료되었습니다. 이런 형태의 tokenized 문장을 NLP 모델에 훈련시키면 같은 형태로 tokenized된 문장을 생성해 낼 것입니다. 그럼 이런 문장을 복원(detokenization)하여 사람이 읽기 좋은 형태로 만들어 주어야 합니다.

먼저 whitespace를 제거합니다.

There 's currently over a thousand TED Talks on the TED website .

그리고 □가 2개가 동시에 있는 문자열 □□을 white space로 치환합니다. 그럼 한 개 짜리 □만 남습니다.

There's currently over a thousand TED Talks on the TED website.

마지막 □를 제거합니다. 그럼 문장 복원이 완성 됩니다.

There's currently over a thousand TED Talks on the TED website.

## Post Tokenization

아래는 기존의 공백과 전처리 한 단계로 인해 생성된 공백을 구분하기 위한 □을 삽입하는 python script 예제 입니다.

```
import sys

STR = ' '

if __name__ == "__main__":
    ref_fn = sys.argv[1]

    f = open(ref_fn, 'r')

    for ref in f:
        ref_tokens = ref.strip().split(' ')
        tokens = sys.stdin.readline().strip().split(' ')

        idx = 0
        buf = []

        # We assume that stdin has more tokens than reference input.
        for ref_token in ref_tokens:
            tmp_buf = []

            while idx < len(tokens):
                tmp_buf += [tokens[idx]]
                idx += 1

                if ''.join(tmp_buf) == ref_token:
```

```

break

if len(tmp_buf) > 0:
    buf += [STR + tmp_buf[0]] + tmp_buf[1:]

sys.stdout.write(' '.join(buf) + '\n')

f.close()

```

위 script의 사용 방법은 아래와 같습니다. 주로 다른 tokenizer 수행 후에 바로 붙여 사용하여 좋습니다.

```
$ cat [before filename] | python tokenizer.py | python post_tokenize.py [before fi
```

## Detokenization

아래는 앞서 설명한 detokenization을 bash에서 sed 명령어를 통해 수행 할 경우에 대한 예제입니다.

```
sed "s/ //g" | sed "s/ / /g" | sed "s//g" | sed "s/^\\s//g"
```

또는 아래의 python script 예제 처럼 처리 할 수도 있습니다.

```

import sys

if __name__ == "__main__":
    for line in sys.stdin:
        if line.strip() != "":
            line = line.strip().replace(' ', '').replace(' ', '').replace(' ', '')

    sys.stdout.write(line + '\n')

```

## TorchText

사실 딥러닝 코드를 작성하다 보면, 신경망 모델 자체를 코딩하는 시간보다 그 모델을 훈련하도록 하는 코드를 짜는 시간이 더 오래걸리기 마련입니다. 데이터 입력을 준비하는 부분도 이에 해당 합니다. TorchText는 NLP 또는 텍스트와 관련된

기계학습 또는 딥러닝을 수행하기 위한 데이터를 읽고 전처리 하는 코드를 모아놓은 라이브러리입니다.

이번 섹션에서는 TorchText를 활용한 기본적인 데이터 로딩 방법에 대한 실습을 해보도록 하겠습니다. 좀 더 자세한 내용은 TorchText 문서를 참조해 주세요.

## How to install

TorchText는 pip을 통해서 쉽게 설치 가능 합니다. 아래와 같이 명령어를 수행하여 설치 합니다.

```
$ pip install torchtext
```

## Example

사실 NLP분야에서 주로 사용되는 학습 데이터의 형태는 크게 3가지로 분류할 수 있습니다. 주로 우리의 신경망이 하고자 하는 바는 입력  $x$ 를 받아 알맞은 출력  $y$ 를 반환해 주는 함수의 형태이므로,  $x, y$ 의 형태에 따라서 분류 해 볼 수 있습니다.

x 데이터	y 데이터	어플리케이션
corpus	클래스(class)	텍스트 분류(text classification), 감성분석(sentiment analysis)
corpus	-	언어모델(language modeling)
corpus	corpus	기계번역(machine translation), 요약(summarization), QnA

따라서 우리는 이 3가지 종류의 데이터 형태를 다루는 방법에 대해서 실습합니다. 사실 TorchText는 훨씬 더 복잡하고 정교한 함수들을 제공합니다. 하지만 글쓴이가 느끼기에는 좀 과도하고 직관적이지 않은 부분들이 많아, 제공되는 함수들의 사용을 최소화 하여 복잡하지 않고 간단한 방식으로 구현해보고자 합니다.

## Reading Corpus with Labeling

첫번째 예제는 한 줄 안에서 클래스(class)와 텍스트(text)가 tab(탭, ‘\t’)으로 구분되어 있는 데이터의 입력을 받기 위한 예제 입니다. 주로 이런 예제는 텍스트 분류(text classification)을 위해 사용 됩니다.

```

class TextClassificationDataLoader():

    def __init__(self, train_fn, valid_fn, tokenizer,
                 batch_size = 64,
                 device = 'cpu',
                 max_vocab = 9999999,
                 fix_length = None,
                 use_eos = False,
                 shuffle = True):

        super(TextClassificationDataLoader, self).__init__()

        self.label = data.Field(sequential = False, use_vocab = False)
        self.text = data.Field(tokenize = tokenizer,
                              use_vocab = True,
                              batch_first = True,
                              include_lengths = True,
                              fix_length = fix_length,
                              eos_token = '<EOS>' if use_eos else None
                             )

        train, valid = data.TabularDataset.splits(path = '',
                                                train = train_fn,
                                                validation = valid_fn,
                                                format = 'tsv',
                                                fields = [('label', self.label),
                                                          ('text', self.text)])
        self.train_iter, self.valid_iter = data.BucketIterator.splits((train, valid),
                                                                    batch_size = batch_size,
                                                                    device = device,
                                                                    shuffle = shuffle)

        self.label.build_vocab(train)
        self.text.build_vocab(train, max_size = max_vocab)

```

## Reading Corpus

이 예제는 한 라인에 텍스트로만 채워져 있을 때를 위한 코드입니다. 주로 언어모델(language model)을 훈련 시키는 상황에서 사용 될 수 있습니다.

```
from torchtext import data, datasets

PAD = 1
BOS = 2
EOS = 3

class DataLoader():

    def __init__(self, train_fn, valid_fn,
                 batch_size = 64,
                 device = 'cpu',
                 max_vocab = 99999999,
                 max_length = 255,
                 fix_length = None,
                 use_bos = True,
                 use_eos = True,
                 shuffle = True
                 ):

        super(DataLoader, self).__init__()

        self.text = data.Field(sequential = True,
                              use_vocab = True,
                              batch_first = True,
                              include_lengths = True,
                              fix_length = fix_length,
                              init_token = '<BOS>' if use_bos else None,
                              eos_token = '<EOS>' if use_eos else None
                             )

    train = LanguageModelDataset(path = train_fn,
                                 fields = [('text', self.text)],
                                 max_length = max_length
```

```

        )
valid = LanguageModelDataset(path = valid_fn,
                            fields = [('text', self.text)],
                            max_length = max_length
                           )

self.train_iter = data.BucketIterator(train,
                                      batch_size = batch_size,
                                      device = 'cuda:%d' % device if dev else None,
                                      shuffle = shuffle,
                                      sort_key=lambda x: -len(x.text),
                                      sort_within_batch = True
                                     )
self.valid_iter = data.BucketIterator(valid,
                                      batch_size = batch_size,
                                      device = 'cuda:%d' % device if dev else None,
                                      shuffle = False,
                                      sort_key=lambda x: -len(x.text),
                                      sort_within_batch = True
                                     )

self.text.build_vocab(train, max_size = max_vocab)

class LanguageModelDataset(data.Dataset):

    def __init__(self, path, fields, max_length=None, **kwargs):
        if not isinstance(fields[0], (tuple, list)):
            fields = [('text', fields[0])]

        examples = []
        with open(path) as f:
            for line in f:
                line = line.strip()
                if max_length and max_length < len(line.split()):
                    continue
                if line != '':
                    examples.append(data.Example.fromlist(
                        [line], fields))

```

```

super(LanguageModelDataset, self).__init__(examples, fields, **kwargs)

if __name__ == '__main__':
    import sys
    loader = DataLoader(sys.argv[1], sys.argv[2])

    for batch_index, batch in enumerate(loader.train_iter):
        print(batch.text)

        if batch_index >= 1:
            break

```

### Reading Parallel(Bi-lingual) Corpus

아래의 코드는 텍스트로만 채워진 두개의 파일을 동시에 입력 데이터로 읽어들이는 예제입니다. 이때, 두 파일의 corpus는 병렬(parallel) 데이터로 취급되어 같은 라인 수로 채워져 있어야 합니다. 주로 기계번역(machine translation)이나 요약(summarization) 등에 사용 할 수 있습니다.

```

import os
from torchtext import data, datasets

PAD = 1
BOS = 2
EOS = 3

class DataLoader():

    def __init__(self, train_fn = None,
                 valid_fn = None,
                 exts = None,
                 batch_size = 64,
                 device = 'cpu',
                 max_vocab = 99999999,
                 max_length = 255,
                 fix_length = None,
                 use_bos = True,

```

```

        use_eos = True,
        shuffle = True
    ):

super(DataLoader, self).__init__()

self.src = data.Field(sequential = True,
                      use_vocab = True,
                      batch_first = True,
                      include_lengths = True,
                      fix_length = fix_length,
                      init_token = None,
                      eos_token = None
                    )

self.tgt = data.Field(sequential = True,
                      use_vocab = True,
                      batch_first = True,
                      include_lengths = True,
                      fix_length = fix_length,
                      init_token = '<BOS>' if use_bos else None,
                      eos_token = '<EOS>' if use_eos else None
                    )

if train_fn is not None and valid_fn is not None and exts is not None:
    train = TranslationDataset(path = train_fn, exts = exts,
                               fields = [('src', self.src), ('tgt', self.tgt)],
                               max_length = max_length
                             )
    valid = TranslationDataset(path = valid_fn, exts = exts,
                               fields = [('src', self.src), ('tgt', self.tgt)],
                               max_length = max_length
                             )

self.train_iter = data.BucketIterator(train,
                                      batch_size = batch_size,
                                      device = 'cuda:%d' % device if device is not None else None,
                                      shuffle = shuffle,

```

```

        sort_key=lambda x: len(x.tgt)
        sort_within_batch = True
    )
    self.valid_iter = data.BucketIterator(valid,
                                         batch_size = batch_size,
                                         device = 'cuda:%d' % device if
                                         shuffle = False,
                                         sort_key=lambda x: len(x.tgt)
                                         sort_within_batch = True
                                         )

    self.src.build_vocab(train, max_size = max_vocab)
    self.tgt.build_vocab(train, max_size = max_vocab)

def load_vocab(self, src_vocab, tgt_vocab):
    self.src.vocab = src_vocab
    self.tgt.vocab = tgt_vocab

class TranslationDataset(data.Dataset):
    """Defines a dataset for machine translation."""

    @staticmethod
    def sort_key(ex):
        return data.interleave_keys(len(ex.src), len(ex.trg))

    def __init__(self, path, exts, fields, max_length=None, **kwargs):
        """Create a TranslationDataset given paths and fields.
        Arguments:
            path: Common prefix of paths to the data files for both languages.
            exts: A tuple containing the extension to path for each language.
            fields: A tuple containing the fields that will be used for data
                    in each language.
        Remaining keyword arguments: Passed to the constructor of
            data.Dataset.
        """
        if not isinstance(fields[0], (tuple, list)):
            fields = [(['src', fields[0]], ('trg', fields[1]))]

```

```
if not path.endswith('.'):
    path += '.'

src_path, trg_path = tuple(os.path.expanduser(path + x) for x in exts)

examples = []
with open(src_path) as src_file, open(trg_path) as trg_file:
    for src_line, trg_line in zip(src_file, trg_file):
        src_line, trg_line = src_line.strip(), trg_line.strip()
        if max_length and max_length < max(len(src_line.split()), len(trg_line)):
            continue
        if src_line != '' and trg_line != '':
            examples.append(data.Example.fromlist(
                [src_line, trg_line], fields))

super(TranslationDataset, self). __init__(examples, fields, **kwargs)
```

# Word Embedding Vector

## Word Embedding

이전 챕터에서 단어의 의미와 모호성에 대해 다루었습니다. 이전에 다루었듯이 사람의 언어는 discrete한 형태의 단어로 이루어져 있습니다. 내부의 의미는 단어와 단어끼리 연관성이 있을 수 있지만, 형태가 다른 경우에는 곁에서 보기에는 연관성이 얼마나 있는지 (특히 컴퓨터의 경우에는) 알기 쉽지 않습니다. 이런 자연어의 특성 때문에, 자연어처리에서 단어 또는 문장(문서)를 벡터(vector)로 나타내는 것은 매우 큰 숙명이었습니다. 비록 자연어의 형태와 속성이 사람이 처리하기는 쉬운 형태로 발전해왔지만, 컴퓨터가 이해하고 처리하기에는 어려운 형태이기 때문입니다. 즉, 사람이 사용하는 자연어의 형태와 컴퓨터가 이해하는 벡터로 변환이 가능한 함수 또는 맵핑 테이블(mapping table)을 만들어 내는 과정은 매우 중요합니다.

따라서 우리는 이전 챕터에서 단어의 의미에 대해 다루면서, 코퍼스로부터 단어의 특징(feature)를 추출하여 벡터로 만드는 과정을 다루어보았습니다. 예를 들어 우리는 코퍼스에서 정해진 크기의 윈도우 내에 함께 나타난 출현 빈도를 세어 매트릭스로 나타내기도 하였습니다. 하지만 우리는 차원의 저주(curse of dimensionality)로 인해서, 그 결과물은 여전히 sparse한 벡터가 나타나는 것을 보았습니다. 같은 데이터를 표현하는데 있어서 가능한 낮은 차원으로 표현할수록 쉽게 모델링하고 학습할 수 있기 때문에, 이러한 sparse한 벡터로 나타나는 것 보단 dense한 벡터로 표현해 주는 것이 훨씬 좋을 것입니다.

이번 챕터에서는 이처럼 단어를 컴퓨터가 이해하고 처리하기 쉬운 형태로 변환하는 과정인 word embedding에 대해 다루어 보겠습니다. # One-hot encoding

단어는 discrete한 심볼(symbol)로 그 내부의 의미는 유사성이 있을 수 있지만, 곁 형태는 다른 경우가 많습니다. (동형이의어 제외) 따라서 가장 기본적으로 단어를 벡터로 나타내는 방법은 one-hot 인코딩(encoding)이라는 방식입니다. 이 방식은 말 그대로 단 한개의 1과 나머지 수많은 0들로 표현된 인코딩 방식을 뜻합니다. One-hot 인코딩 벡터의 차원은 보통 전체 어휘(vocabulary)의 갯수가 되며, 당연히 보통 그 숫자는 매우 큰 숫자가 됩니다. (대략 30,000~100,000)

$$v \in \mathbb{R}^{|V|}, \text{ where } v \text{ is one-hot vector and } |V| \text{ is vocabulary size.}$$

따라서 전체 단어에 대해서 one-hot 벡터를 구성한다면 아래와 같은 형태가 될 것입니다.



Figure 1: Tomas Mikolov – Image from web

단어	사전에서의 순서(index)	One-hot 벡터
...		
강아지	8	0, 0, 0, 0, 0, 0, 1, 0, 0, 0, ..., 0
개	9	0, 0, 0, 0, 0, 0, 0, 1, 0, 0, ..., 0
고양이	10	0, 0, 0, 0, 0, 0, 0, 0, 1, 0, ..., 0
구렁이	11	0, 0, 0, 0, 0, 0, 0, 0, 0, 1, ..., 0
...		
하마	20,567	0, ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
...		

위처럼 사전(dictionary)내의 각 단어를 one-hot encoding 방식을 통해 벡터로 나타낼 수 있습니다. 문제는 표현하는 정보에 비해 벡터의 차원이 너무 커진 것입니다. 수만개의 단어 중 하나의 단어를 표현하기 위해서 단어수 만큼의 차원의 벡터에서 하나의 차원을 제외하고 모두 0으로 채웠기 때문입니다. 즉, 하나의 차원을 제외한 모든 차원이 0으로 채워진 벡터입니다. 이러한 벡터를 우리는 sparse vector라고 부릅니다.

## Curse of dimensionality

문제는 이런 sparse vector는 기계학습에 있어서 매우 큰 장벽으로 작용한다는 점입니다. 예를 들어, 점보를 표현하는데 훨씬 큰 차원이 사용되었다면 작은 차원으로 같은 정보를 표현한 것에 비해서, 상대적으로 같은 크기의 공간에 표현되는 정보는 훨씬 더 적을 것이기 때문입니다.

차원의 저주: 차원이 높을 수록 같은 정보를 표현하는데 불리합니다.

우리는 이런 문제를 차원의 저주(curse of dimensionality)라고 부릅니다.

## Similarity

One-hot encoding을 통해 sparse vector로 표현된 워드 임베딩 벡터의 경우에는 또 다른 어려움이 있습니다. 바로 one-hot encoding을 통해서는 단어간의 유사도를 표현할 수 없다는 것입니다.

$$[0, 0, \dots, 1, 0] \times [0, 1, 0, \dots, 0]^T = 0$$

우리는 임베딩 벡터간의 유사도 비교를 통해, 부족한 정보를 비슷한 다른 단어로부터 가져올 수 있을 것 입니다. 하지만 one-hot encoding을 통해 구한 임베딩 벡터의 경우에는 코사인 유사도(cosine similarity)가 항상 0일 수 밖에 없습니다. 또한, 유클리디안 거리(eucleadian distance)의 경우에도 모두가 같을 것 입니다. 따라서, one-hot encoding에 따르면, 컴퓨터는 ‘고양이’와 ‘강아지’, 그리고 ‘개’ 사이의 유사도를 구할 수 없을 것 입니다. 이것은 데이터를 통해서 학습을 수행하는 기계학습의 특성상 데이터가 많을 수록 유리한데 반해, 불리하게 작용할 수 밖에 없습니다. ‘강아지’에 대한 데이터가 적어 일반화(generalization)하기 어려울 때, ‘개’와 관련된 데이터로부터 도움을 받을 수 없을 것이기 때문입니다.

따라서 우리는 위와 같이 one-hot encoding의 한계를 실감할 수 밖에 없습니다. 따라서 우리는 차원의 저주로부터 벗어나기 위해 차원을 축소하여 단어를 표현해야 할 필요성을 느낍니다. # Dimension Reduction

이전 장에서 우린 높은 차원에서 데이터를 표현하였기 때문에 어려움이 많은 것을 살펴보았습니다. 따라서 우리는 같은 정보를 표현하기 위해서 보다 낮은 차원을 사용하는 것이 중요합니다.

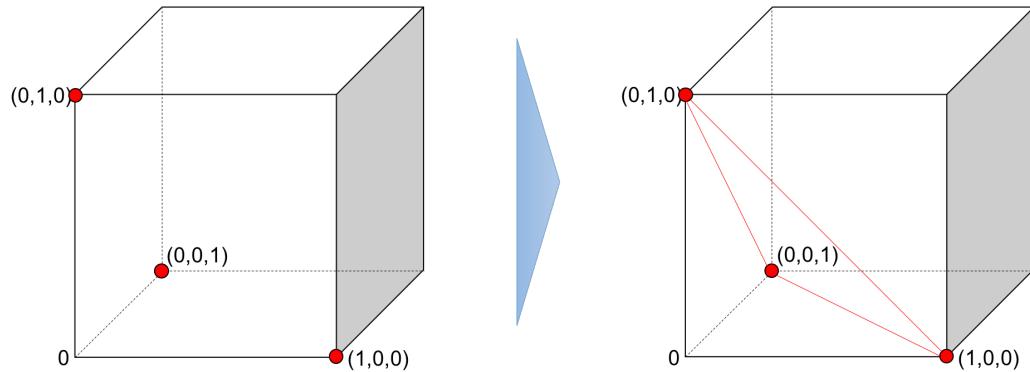


Figure 2: One-hot 벡터의 경우에는 아래와 같이 차원 축소가 가능할 것입니다.

이번 장에서는 좀 더 작은 차원으로 효율적으로 정보를 표현하는 방법에 대해서 다루어 보겠습니다.

## Principal Component Analysis (PCA)

대표적인 차원축소 방법으로는 PCA(Principal Component Analysis, 주성분분석)가 있습니다.

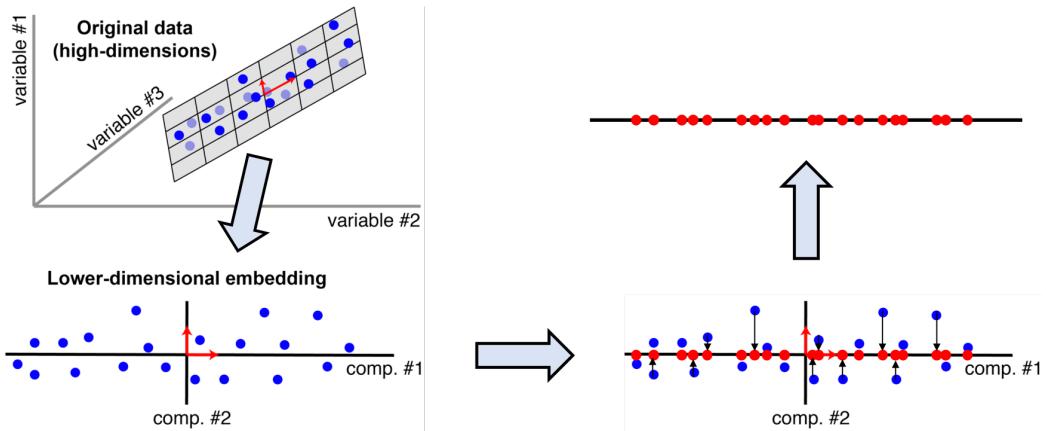


Figure 3: 3차원에서 2차원, 다시 1차원으로 PCA를 수행하는 예

위와 같이 고차원(high-dimension)의 데이터를 보다 낮은 차원으로 표현하는 것이 가능합니다. 주로 Singular Vector Decomposition (SVD)를 통해 PCA를 수행 할 수 있습니다.

이때 축소를 위한 주성분(principal component)는 위와 같은 조건을 만족합니다. 고차원에서 주어진 데이터들을 임의의 주성분 hyper-plane에 투사(projection)하였을 때, 투사점 사이가 최대한 멀어지도록 되어야 합니다. 또한, hyper-plane으로 투사할 때, 원래 벡터와 hyper-plane상의 거리가 최소가 되도록 하여야 합니다.

PCA를 통해 우리는 효과적으로 고차원의 데이터를 보다 낮은 차원으로 압축할 수 있습니다. 하지만 위에서 언급했듯이, 실제 데이터(점)의 위치와 hyper-plane에 투사된 점의 거리가 생길 수 밖에 없습니다. 이는 곧 정보의 손실을 의미합니다. 특히 주성분은 직선 또는 평면일 수 밖에 없기 때문에, 이러한 손실을 최소화 하는 것은 생각보다 어렵습니다. 너무 많은 정보가 손실된다면, 효율적으로 정보를 학습하거나 복구할 수 없기 때문입니다. 따라서 높은 차원에 표현되어 있는 정보를 지나치게 낮은 차원으로 축소하여 표현하는 것은 어려움이 따릅니다. 특히 데이터가 비선형적으로 표현되어 있을수록 이는 점점 더 어려워집니다.

## Manifold Hypothesis

이때 하나의 가설을 통해 우리는 좀 더 차원축소를 효율적으로 접근해 볼 수 있습니다. 사실 one-hot encoding과 같이 대부분의 높은 차원에 존재하는 데이터들은 비록 높은 차원에서 표현되어 있지만, 해당 데이터들을 아우르는 낮은

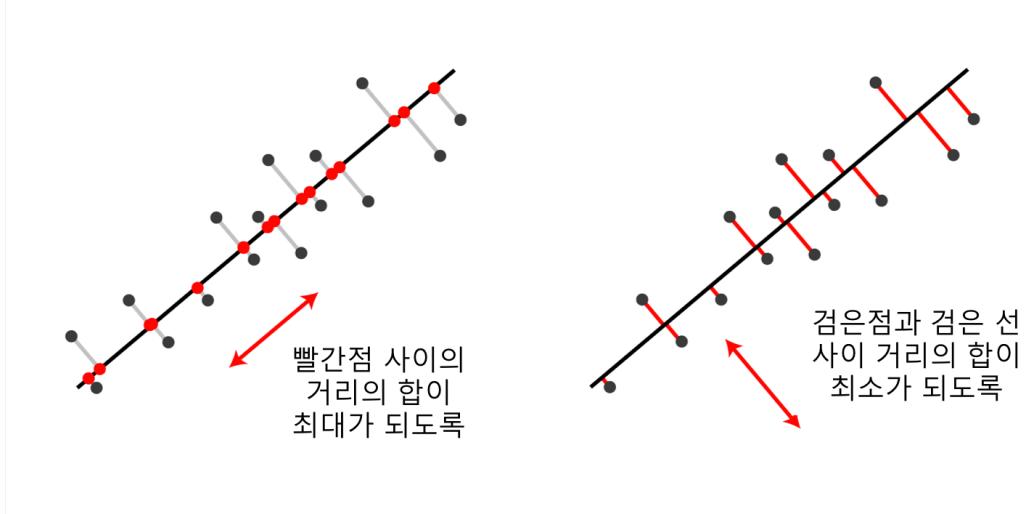


Figure 4: 주성분의 조건

차원의 매니폴드(manifold, 다양체)가 존재한다는 가설입니다.

위와 같이 3차원상에 분포한 데이터를 아우르는 소용돌이 모양의 구부려진 2차원 매니폴드가 존재할 수도 있을 것 입니다. 그럼 우리는 이런 매니폴드를 찾아 2차원의 평면 상에 데이터 포인트들을 표현할 수 있을 것 입니다. 만약 이 매니폴드를 찾을 수 있다면 이전 PCA처럼 데이터를 hyper-plane에 투사(projection)하며 생긴 손실을 더욱 최소화 할 수 있을 것 입니다.

또한, 매니폴드 가설에 따르면 또 하나의 흥미로운 특성이 있습니다. 비록 고차원상에서는 가까운 거리에 있던 벡터들일지라도, 매니폴드를 저차원 hyper-plane으로 표현하였을 때는 오히려 거리가 멀어질 수 있다는 것 입니다. 그리고 매니폴드 표면(surface)상에서 가까운 점들끼리는 실제 의미적으로 가까운 데이터를 표현하고 있다는 것 입니다.

## Why deep learning is so working well

바로 딥러닝이 훌륭한 성능을 내는 이유가 여기 있습니다. 거의 모든 문제에 있어서 딥러닝이 문제를 풀기위해 수행하는 것은 바로 차원 축소이며, 그 과정은 데이터가 존재하는 고차원(high-dimension)상에서 매니폴드(manifold)를 찾는 과정 그 자체이기 때문입니다. 다른 선형적인 방식에 비해서 딥러닝은 비선형적인 방식으로 차원축소를 수행하며, 그 과정에서 해당 문제를 가장 잘 해결하기 위한 매니폴드를

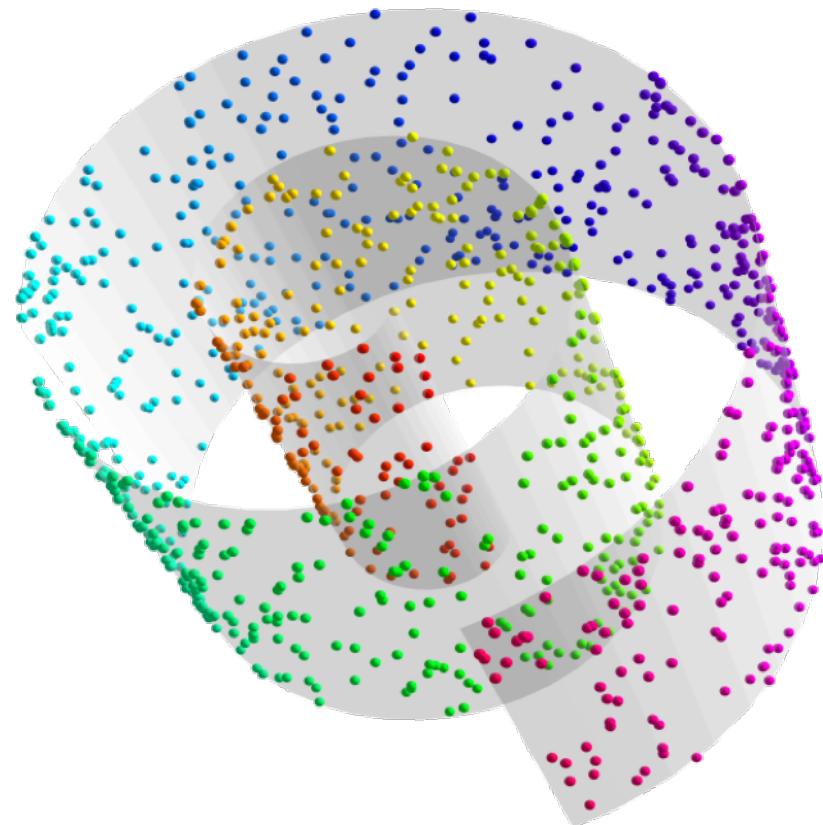


Figure 5:

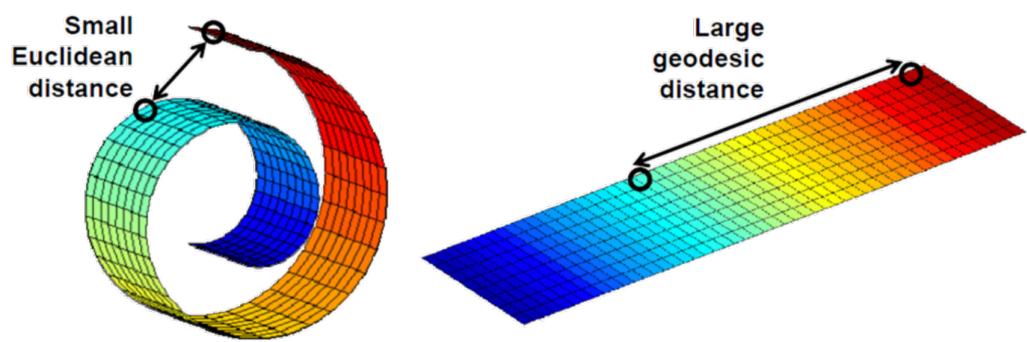


Figure 6:

자연스럽게 찾아냅니다. 이것이 바로 딥러닝이 이토록 성공적으로 동작하는 이유입니다.

## Auto-encoder

그럼 본격적으로 자연어처리에서 단어를 표현하기 위한 차원 축소를 다루기에 앞서, 오토인코더(auto-encoder)에 대해 다루고 넘어가도록 하겠습니다. 오토인코더는 아래와 같은 구조를 가진 딥러닝 모델입니다.

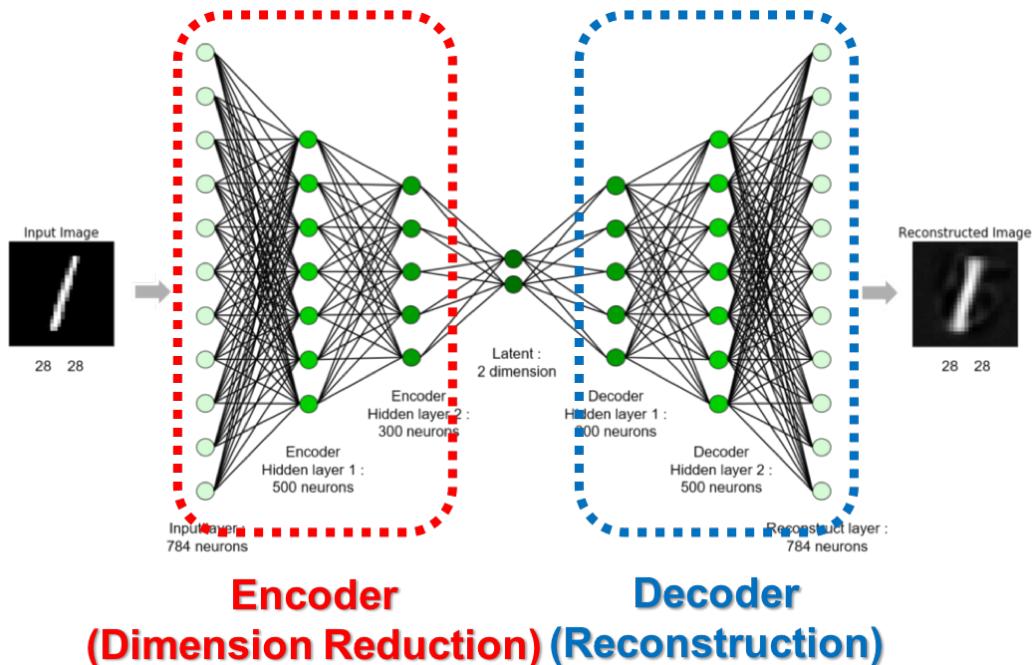


Figure 7:

고차원의 입력을 받아 저차원으로 축소하는 인코더를 거쳐 매니폴드인 bottle-neck(병목)에서의 숨겨진(latent, hidden) 데이터로 표현 합니다. 그리고 디코더는 숨겨진 저차원에서의 벡터를 받아, 다시 원래 입력 데이터가 존재하던 고차원으로 데이터를 복원하는 작업을 수행 합니다.

따라서 이 구조의 모델을 훈련할 때엔, 복원된 데이터와 실제 입력 데이터 사이의 차이를 최소화하도록 손실함수(loss function)을 구성합니다.

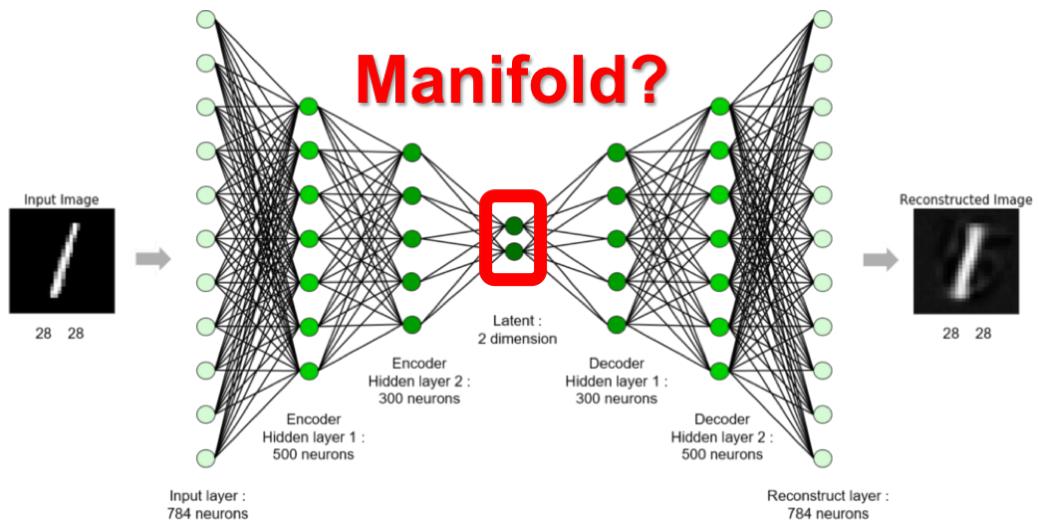


Figure 8:

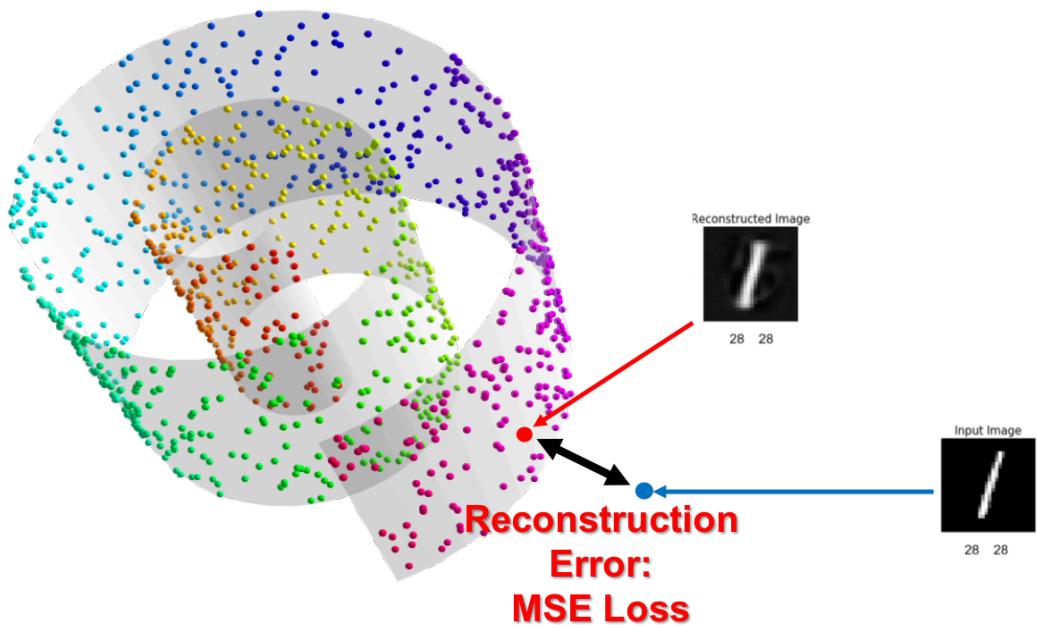


Figure 9:

하지만 고차원에서 저차원으로 데이터를 표현하는 것에 손실이 따를 수 있기 때문에, 훈련이 완료된 모델일지라도 복원된 데이터는 실제 입력과 차이가 있을 수 있습니다.

우리는 이 오토인코더를 사용하여 이전 챕터에서 구해진 sparse word feature 벡터를 dense word embedding 벡터로 변환할 수 있을 것 입니다. # Myth

우리는 이번 챕터를 통해 단어를 벡터로 표현하는 방법에 대해서 살펴보고 있습니다. 이어지는 섹션에서 skip-gram 또는 GloVe를 사용하여 One-hot encoding의 sparse 벡터를 차원 축소(dimension reduction)하여 훨씬 작은 차원의 dense 벡터로 표현하는 방법에 대해 다룰 것 입니다.

이에 앞서, 하지만 많은 분들이 헷갈려 하는 부분이 있다면, 이렇게 훈련한 단어 임베딩 벡터를 추후 우리가 다룰 텍스트 분류, 언어모델, 번역 등의 딥러닝 모델들의 입력으로 사용할 것이라고 생각한다는 점입니다.

Word2Vec을 통해 얻은 단어 임베딩 벡터는 훌륭하게 단어의 특성을 잘 반영하고 있지만, 텍스트 분류, 언어모델, 번역의 문제 해결을 위한 최적의 벡터 임베딩이라고는 볼 수 없습니다.

예를 들어 긍정/부정 분류를 하는 텍스트 분류 문제의 경우에는 ‘행복’이라는 단어가 매우 중요한 특징(feature)가 될 것이고, 이를 표현하기 위한 임베딩 벡터가 존재할 것입니다. 하지만 영화 시놉시스의 장르를 분류하기 위한 문제에서는 ‘행복’이라는 단어는 다른 특징으로 작용 될 것이며, 이 분류 문제를 위한 ‘행복’ 단어의 임베딩 벡터의 값은 이전 긍정/부정 분류 문제의 값과 당연히 달라지게 될 것입니다. 따라서 문제의 특성을 고려하지 않은 단어 임베딩 벡터는 그다지 좋은 방법이 될 수 없습니다.

## How to get word embedding instead of using Word2Vec

우리는 Word2Vec을 사용하지 않더라도, 문제의 특성에 맞는 단어 임베딩을 구할 수 있습니다. PyTorch를 비롯한 여러 딥러닝 프레임워크는 Embedding Layer라는 레이어 아키텍처를 제공합니다.

이 레이어는 아래와 같이 bias가 없는 Linear Layer와 같은 형태를 갖고 있습니다.

$$y = \text{emb}(x) = Wx,$$

where  $W \in \mathbb{R}^{d \times |V|}$  and  $|V|$  is size of vocabulary.

쉽게 생각하면  $W$ 는  $d \times |V|$  크기의 2차원의 매트릭스입니다. 따라서 입력으로 one-hot vector가 주어지게 되면,  $W$ 의 특정 column(구현 방법에 따라 또는 row)만

반환하게 됩니다.

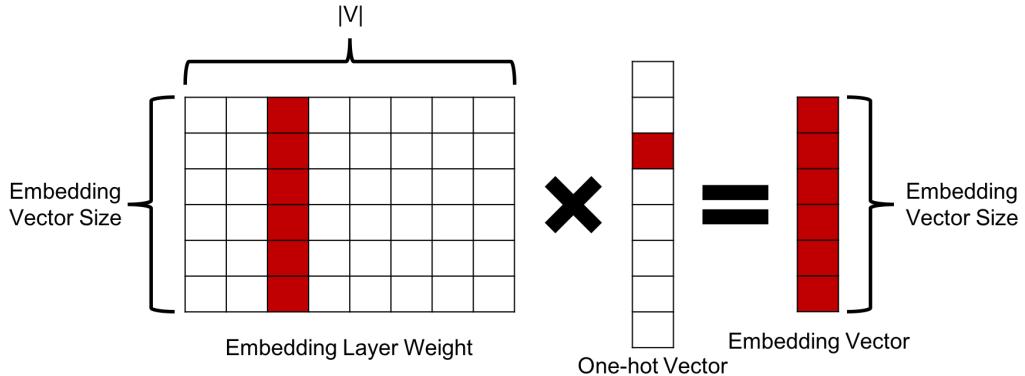


Figure 10:

따라서 최종적으로 모델으로부터 구한 손실값(loss)에 따라 back-propagation 및 gradient descent를 수행하게 되면, 자동적으로 embedding layer의 weight  $W$ 의 값을 구할 수 있게 될 것 입니다.

물론 실제 구현에 있어서는 이렇게 큰 embedding layer weight와 one-hot 벡터를 곱하는 것은 매우 비효율적이므로, 단순히 테이블에서 lookup하는 작업을 수행 합니다. 따라서 우리는 단어를 나타냄에 있어 (embedding layer의 입력으로) one-hot 벡터를 굳이 넘겨줄 필요 없이, 1이 존재하는 단어의 index 정수 값만 입력으로 넘겨주면 embedding vector를 얻을 수 있습니다.

추후 실제 앞으로 우리가 다룰 텍스트 분류나 기계번역 챕터에서 구현된 것을 살펴보면, 위에서 말한대로 embedding layer를 사용하여 구현한 것을 알 수 있습니다. # GloVe

이전 섹션에서는 Word2Vec에 대해서 다루어 보았습니다. 이번 섹션에서는 또 하나의 대표적인 word embedding 방법인 GloVe (Global Vectors for Word Representation) [Pennington et al.2014]에 대해 다루어보고자 합니다.

## Arhitecture

이전에 다루었던 skip-gram은 대상 단어를 통해 주변 단어를 예측하도록 네트워크를 구성하여 word embedding 벡터를 학습하였습니다. GloVe는 대신에 대상 단어에 대해서 코퍼스에 함께 나타난 각 단어별 출현 빈도를 예측하도록 합니다. GloVe를 구하기 위한 파라미터를 구하는 수식은 아래와 같습니다.

$$\hat{\theta} = \operatorname{argmin}_{\theta} \sum_{x \in \mathcal{X}} f(x) \times \|W'Wx - \log C_x\|_2$$

where  $C_x$  is vector of co-occurrences with  $x$

Also,  $x \in \{0, 1\}^{|V|}$ ,  $W \in \mathbb{R}^{d \times |V|}$  and  $W' \in \mathbb{R}^{|V| \times d}$ .

Skip-gram을 위한 네트워크와 거의 유사한 형태임을 알 수 있습니다. 다만, 여기서는 classification이 아닌, 출현 빈도에 근사(approximation)하는 regression에 가깝기 때문에, Mean Square Error (MSE)를 사용한 것을 볼 수 있습니다.

마찬가지로 one-hot 인코딩 벡터  $x$ 를 입력으로 받아 한 개의 hidden layer  $W$ 를 거쳐 output layer  $W'$ 를 통해 출력 벡터를 반환 합니다. 이 출력 벡터는 단어  $x$ 와 함께 코퍼스에 출현했던 모든 단어들의 각 동시 출현 빈도들을 나타낸 벡터인  $C_x$ 를 근사해야 합니다. 따라서 이 둘의 차이를 손실(loss)로 삼아 back-propagation 및 gradient descent를 통해 학습을 할 수 있습니다.

이때 단어  $x$  자체의 출현빈도 또는 사전확률(prior probability)에 따라서 MSE loss의 값이 매우 달라질 것 입니다. 예를 들어,  $C_x$ 가 클수록 손실(loss)값은 커질것이기 때문입니다. 따라서  $f(x)$ 는 단어의 빈도에 따라 아래와 같이 손실 함수에 가중치를 부여합니다.

$$f(x) = \begin{cases} \left(\text{Count}(x)/\text{thres}\right)^{\alpha} & \text{Count}(x) < \text{thres} \\ 1 & \text{otherwise.} \end{cases}$$

이 논문에서는 실험에 의해  $\text{thres} = 100$ ,  $\alpha = 3/4$  일때 가장 좋은 결과가 나온다고 언급하였습니다.

## Pros

코퍼스 내에서 주변단어를 예측하고자 하는 Skip-gram과 달리, GloVe는 처음에 코퍼스를 통해 각 단어별 동시 출현 빈도를 조사하여 이에 대한 매트릭스를 만들고, 이후엔 해당 매트릭스를 통해 동시 출현 빈도를 근사(approximate)하고자 합니다. 따라서 계속하여 코퍼스 전체를 훑으며 대상 단어와 주변 단어를 가져와 학습해야 하는 skip-gram과 달리 훨씬 학습이 빠릅니다.

또한, 코퍼스를 훑으며 학습하는 skip-gram의 특성상, 사전확률(prior probability)이 작은 (즉, 출현 빈도 자체가 작은) 단어에 대해서는 학습 기회가 적을 수 밖에

없습니다. 따라서 출현 빈도가 작은 단어들은 비교적 부정확한 word embedding 벡터를 학습하게 됩니다. 하지만, GloVe는 skip-gram에 비해서 이러한 단점이 어느정도 보완되었습니다.

## Conclusion

비록 GloVe의 저자는 GloVe가 가장 뛰어난 embedding 방식임을 주장하였지만, 사실 skip-gram도 파라미터 튜닝 여부에 따라 GloVe와 큰 성능 차이가 없습니다. 따라서 실제 구현하고자 할때에는 주어진 상황(예를 들어 시스템 구성)에 따라 적절한 방법을 선택하는 것도 한가지 방법입니다. # Word2Vec Practice

## FastText

## Visualization

## Sequence Modeling



Figure 1: Andrey Markov – Image from Wikipedia

## Sequential Modeling

우리는 3차원의 공간과 1차원의 시간을 합친 시공간(spacetime)에 살고 있습니다. 그리고 기계학습(machine learning) 또는 인공지능(artificial intelligence)라는 방법을 이용하여 우리의 삶을 개선하고자 합니다. 따라서 많은 문제들은 시공간 상에서 정의되어 있기 마련이고, 이러한 문제들을 풀기 위해 접근함에 있어서 시간의 개념(순서 정보)을 적용하여 문제를 해결하는 것은 중요합니다. 예를 들어 주식시장의 주가 예측이나, 일기예보부터 음성인식이나 번역까지 수많은 문제들이 시간 개념이 큰 영향을 끼칩니다.

우리가 다루고자 하는 자연어(natural language), 즉 텍스트(text)의 경우에도 단어들이 모여(sequence) 문장이 되고, 문장이 모여 문서가 됩니다. 문장 내의 단어들은 앞뒤 위치에 따라 서로에게 영향을 주고, 문서 내의 문장들도 위치에 따라 서로에게 영향을 주고 받습니다. 따라서 우리는 단순히  $y = f(x)$ 와 같이 시간의 개념이 없이 입력을 넣으면 출력이 나오는 함수의 형태가 아닌, 시간에 따라서 순차적(sequential)으로 입력을 넣고, 입력에 따라 모델의 상태(hidden state)가 변하며, 출력(observation)이 상태에 따라 반환되는 그러한 함수가 필요합니다.

이런 시간 개념 또는 순서 정보를 사용하여 입력을 학습하는 것을 Sequential Modeling이라고 합니다. 신경망(neural network) 뿐만이 아니라 다양한 방법(Hidden Markov Model, Conditional Random Fields 등)을 통해 이런 문제들에 접근 할 수 있으며, 신경망에서는 Recurrent Neural Network(순환신경망, RNN)이라는 아키텍처를 사용하여 효과적으로 문제를 해결할 수 있습니다.

## (Vanilla) Recurrent Neural Network

기존 신경망은 정해진 입력  $x$ 를 받아  $y$ 를 출력해 주는 형태였습니다.

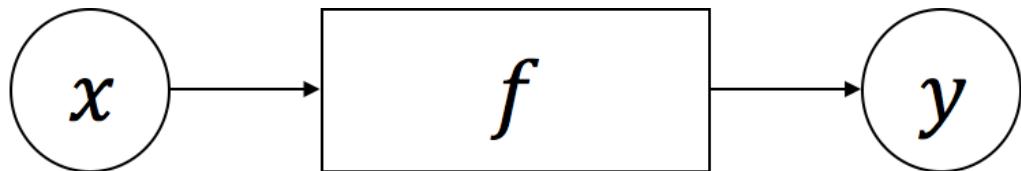


Figure 2: 기존의 뉴럴넷 구조

$$y = f(x)$$

하지만 recurrent neural network (순환신경망, RNN)은 입력  $x_t$ 와 직전 자신의 상태(hidden state)  $h_{t-1}$ 를 참조하여 현재 자신의 상태  $h_t$ 를 결정하는 작업을 여러 time-step에 걸쳐 수행 합니다. 각 time-step별 RNN의 상태는 경우에 따라 출력이 되기도 합니다.

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

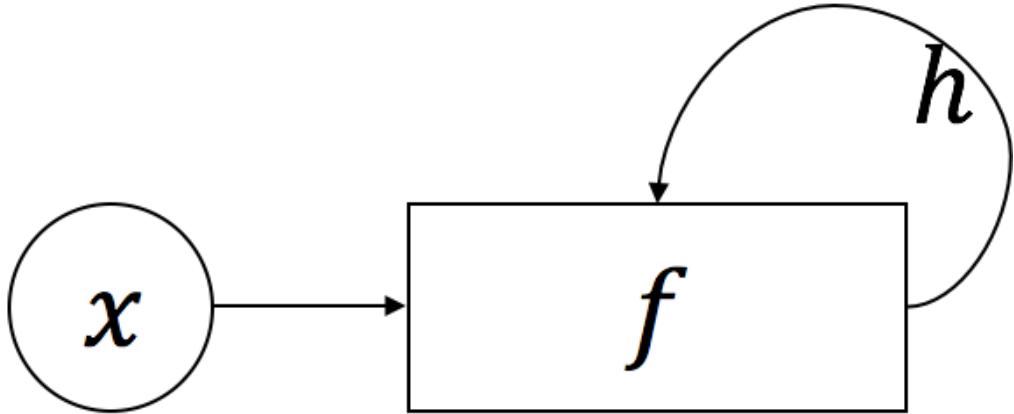


Figure 3: Recursive한 속성이 부여된 뉴럴넷 구조

### Feed-forward

기본적인 RNN을 활용한 feed-forward 계산의 흐름은 아래와 같습니다. 아래의 그림은 각 time-step 별로 입력  $x_t$ 와 이전 time-step의  $h_t$ 가 RNN으로 들어가서 출력으로  $h_t$ 를 반환하는 모습입니다. 이렇게 얻어낸  $h_t$ 들을  $\hat{y}_t$ 로 삼아서 정답인  $y_t$ 와 비교하여 손실(loss)  $\mathcal{L}$ 을 계산 합니다.

위 그림을 수식으로 표현하면 아래와 같습니다. 함수  $f$ 는  $x_t$ 와  $h_{t-1}$ 을 입력으로 받아서 파라미터  $\theta$ 를 통해  $h_t$ 를 계산 합니다. 이때, 각 입력과 출력 그리고 내부 파라미터의 크기는 다음과 같습니다.  $- x_t \in \mathbb{R}^w, h_t \in \mathbb{R}^d, W_{ih} \in \mathbb{R}^{d \times w}, b \in \mathbb{R}^d, W_{hh} \in \mathbb{R}^{d \times d}, b_{hh} \in \mathbb{R}^d$

$$\begin{aligned}\hat{y}_t &= h_t = f(x_t, h_{t-1}; \theta) \\ &= \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \\ \text{where } \theta &= [W_{ih}; b_{ih}; W_{hh}; b_{hh}].\end{aligned}$$

위의 수식에서 나타나듯이 RNN에서는 ReLU나 다른 활성함수(activation function)을 사용하기보단 tanh를 주로 사용합니다. 최종적으로 각 time-step별로  $y_t$ 를 계산하여 아래의 수식처럼 모든 time-step에 대한 손실(loss)  $\mathcal{L}$ 을 구합니다.

$$\mathcal{L} = \frac{1}{n} \sum_{t=1}^n loss(y_t, \hat{y}_t)$$

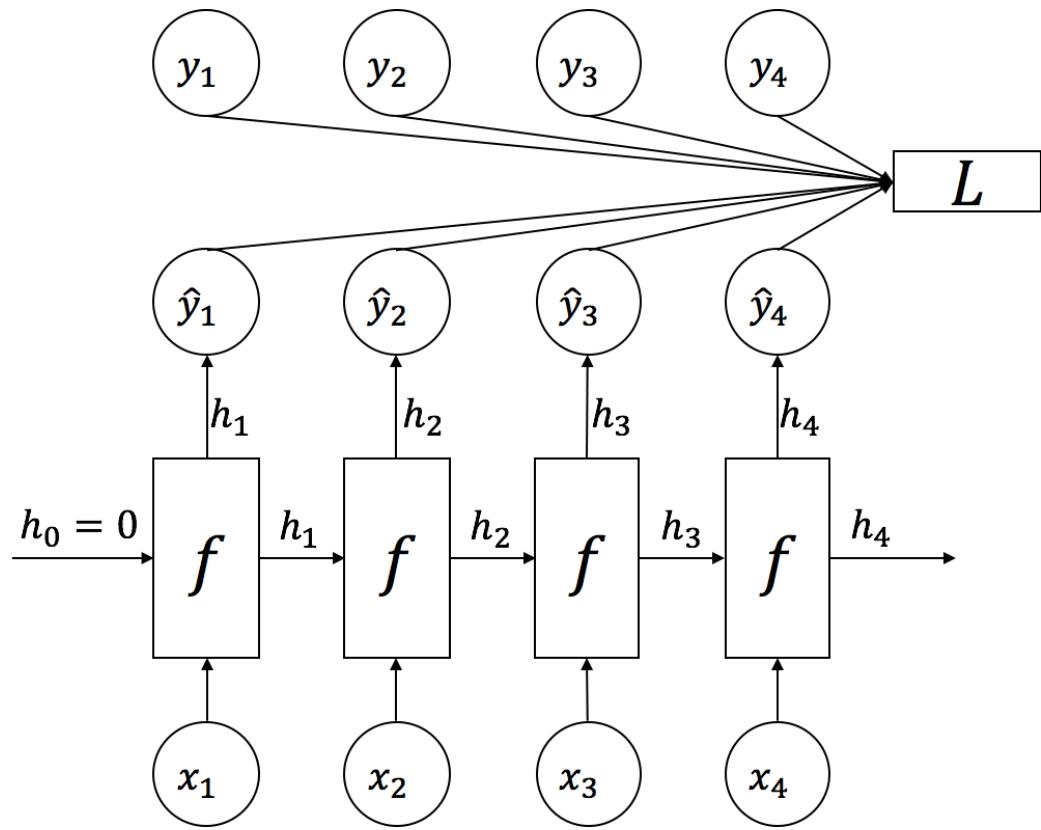


Figure 4: 기본적인 RNN의 feed-forward 형태

## Back-propagation Through Time (BPTT)

그럼 이렇게 feed-forward 된 이후에 오류의 back-propagation(역전파)은 어떻게 될까요? 우리는 수식보다 좀 더 개념적으로 접근해 보도록 하겠습니다.

각 time-step의 RNN에 사용된 파라미터  $\theta$ 는 모든 시간에 공유되어 사용 되는 것을 기억 해 봅시다. 따라서, 앞서 구한 손실  $\mathcal{L}$ 에 미분을 통해 back-propagation 하게 되면, 각 time-step 별로 뒤( $t$ 가 큰 time-step)로부터  $\theta$ 의 gradient가 구해지고, 이전 time-step ( $t - 1$ )  $\theta$ 의 gradient에 더해지게 됩니다. 즉,  $t$ 가 0에 가까워질수록 RNN 파라미터  $\theta$ 의 gradient는 각 time-step 별 gradient가 더해져 점점 커지게 됩니다.

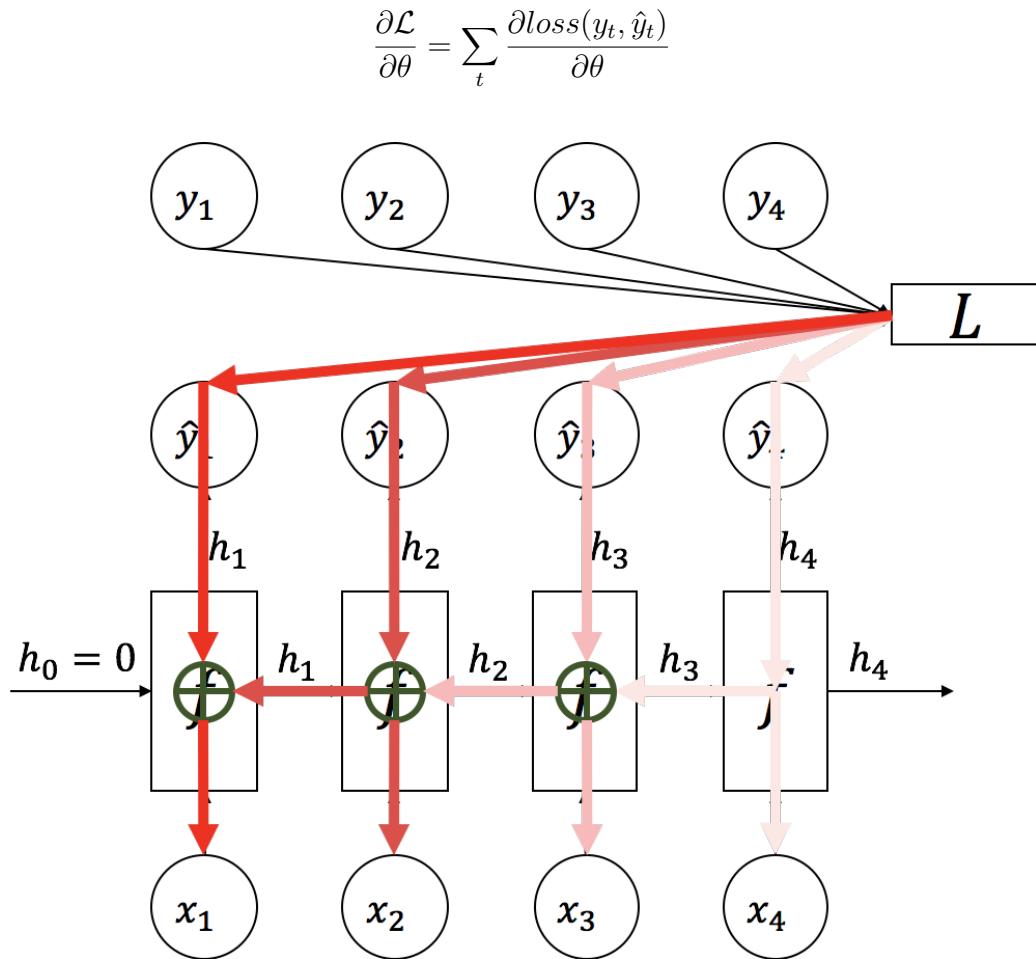


Figure 5: RNN에서 BPTT가 되는 모습

위 그림에서는 붉은색이 점점 짙어지는 것으로 그런 RNN back-propagation의 속성을 나타내었습니다. 이 속성을 back-propagation through time(BPTT)이라고 합니다.

이런 RNN back-propagation의 속성으로 인해, 마치 RNN은 time-step의 수 만큼 layer(계층)이 있는 것이나 마찬가지가 됩니다. 따라서 time-step이 길어짐에 따라, 매우 깊은 신경망과 같이 동작 합니다.

## Gradient Vanishing

상기 했듯이, BPTT로 인해 RNN은 마치 time-step 만큼의 layer가 있는 것과 비슷한 속성을 띠게 됩니다. 그런데 위의 RNN의 수식을 보면, 활성함수(activation function)으로  $\tanh$ (Hyperbolic Tangent, '탄에이치'라고 읽기도 합니다.)가 사용 된 것을 볼 수 있습니다.  $\tanh$ 은 아래와 같은 형태를 띠고 있습니다.

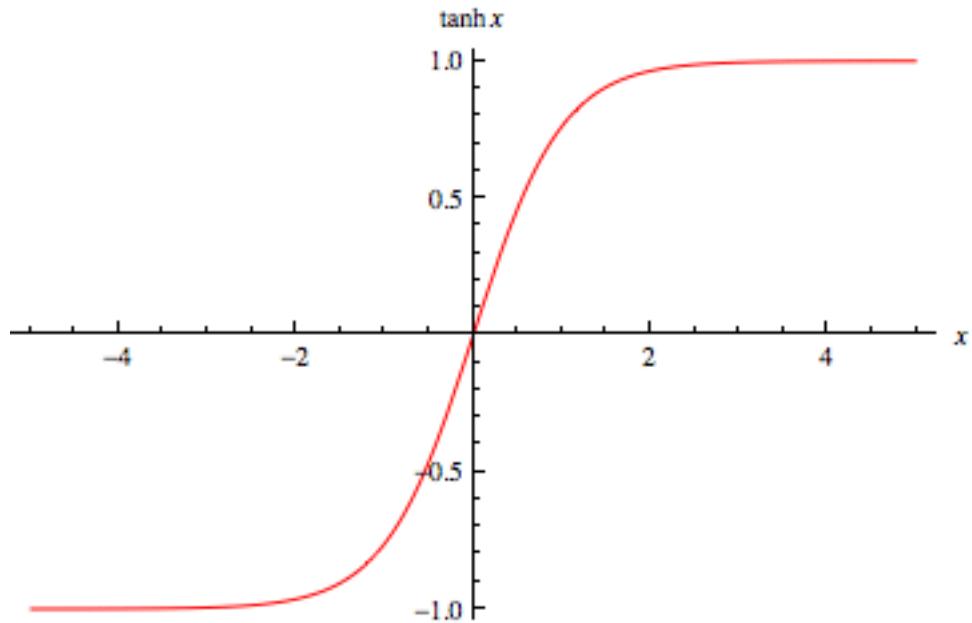


Figure 6: Hyperbolic Tangent의 형태

$\tanh$ 의 양 끝은 수평에 가깝게되어 점점  $-1$  또는  $1$ 에 근접하는 것을 볼 수 있습니다. 문제는 이렇게 되면,  $\tanh$  양 끝의 gradient는  $0$ 에 가까워진다는 것입니다. 따라서  $\tanh$  양 끝의 값을 반환하는 layer의 경우에는 gradient가  $0$ 에 가깝게 되어, 그

다음으로 back-propagation 되는 layer는 제대로 된 gradient를 전달 받을 수가 없게 됩니다. 이를 gradient vanishing이라고 합니다.

따라서, time-step이 많거나 여러층으로 되어 있는 신경망의 경우에는 이 gradient vanishing 문제가 쉽게 발생하게 되고, 이는 딥러닝 이전의 신경망 학습에 큰 장애가 되곤 하였습니다.

## Multi-layer RNN

기본적으로 Time-step별로 RNN이 동작하지만, 아래의 그림과 같이 한 time-step 내에서 RNN을 여러 층을 쌓아올릴 수 있습니다. 그림상으로 시간의 흐름은 왼쪽에서 오른쪽으로 간다면, 여러 layer를 아래에서 위로 쌓아 올릴 수 있습니다. 따라서 여러개의 RNN layer가 쌓여 하나의 RNN을 이루고 있을 때, 가장 위층의 hidden state가 전체 RNN의 출력값이 됩니다.

당연히 각 층 별로 파라미터  $\theta$ 를 공유하지 않고 따로 갖습니다. 보통은 각 layer 사이에 dropout을 끼워 넣기도 합니다.

기존의 단층 RNN의 경우에는 hidden state와 출력값이 같은 값이었지만, 여러 층이 쌓여 이루어진 RNN의 경우에는 각 time-step의 출력값이 맨 윗층의 hidden state가 됩니다.

## Bi-directional RNN

여러 층을 쌓는 방법에 대해 이야기 했다면, 이제 RNN의 방향에 대해서 이야기 할 차례입니다. 이제까지 다른 RNN은  $t$ 가 1에서부터 마지막 time-step 까지 차례로 입력을 받아 진행 하였습니다. 하지만, bi-directional(양방향) RNN을 사용하게 되면, 기존의 정방향과 추가적으로 마지막 time-step에서부터 거꾸로 역방향으로 입력을 받아 진행 합니다. Bi-directional RNN의 경우에도 당연히 정방향과 역방향의 파라미터  $\theta$ 는 공유되지 않습니다.

보통은 여러 층의 bi-directional RNN을 쌓게 되면, 각 층마다 두 방향의 각 time-step 별 출력(hidden state)값을 이어붙여(concatenate) 다음 층(layer)의 각 방향 별 입력으로 사용하게 됩니다. 경우에 따라서 전체 RNN layer들 중에서 일부 층만 bi-directional을 사용하기도 합니다.

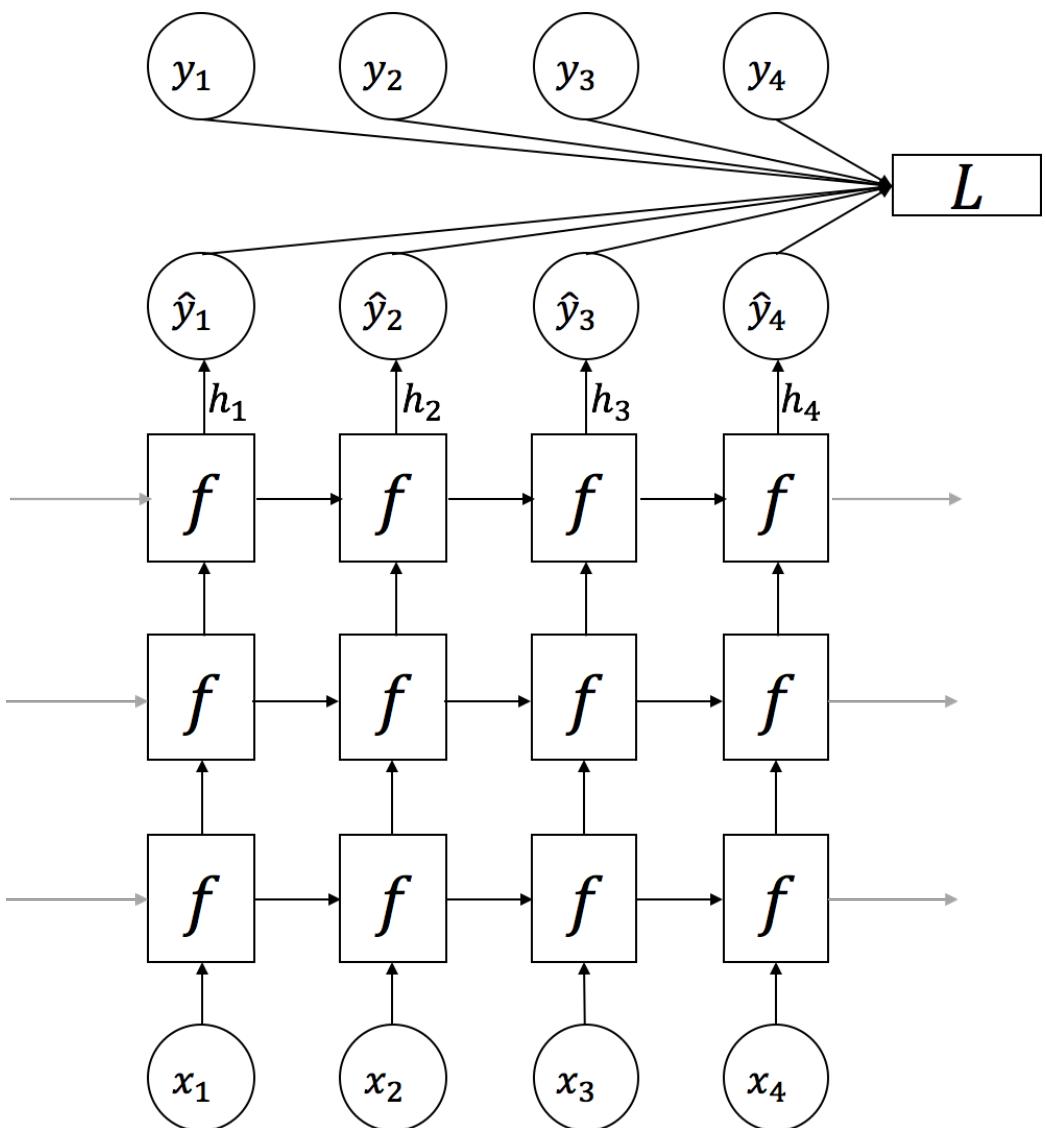


Figure 7: 여러 층이 쌓인 RNN의 형태

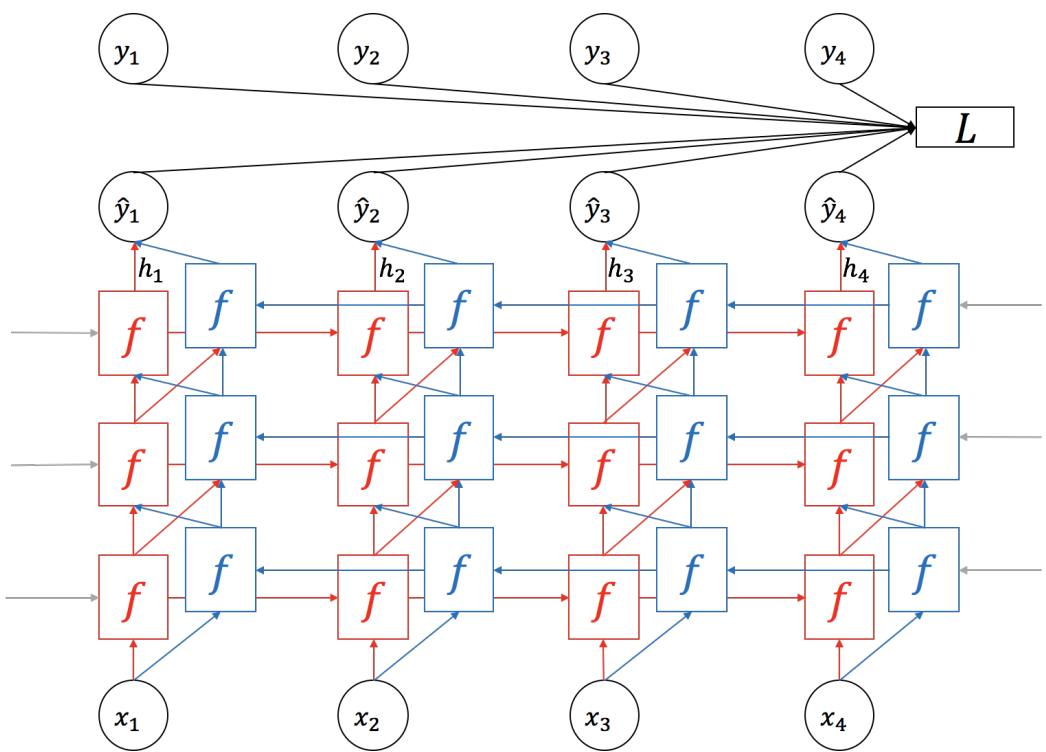


Figure 8: 두 방향으로 hidden state를 전달 및 계산하는 RNN의 형태

## How to Apply to NLP

그럼 위에서 다른 내용을 바탕으로 RNN을 NLP를 비롯한 실무에서는 어떻게 적용하는지 알아보도록 하겠습니다. 여기서는 RNN을 한개 층만 쌓아 정방향으로만 다른 것 처럼 묘사하였지만, 여러 층을 양방향으로 쌓아 사용하는 것도 대부분의 경우 가능 합니다.

### Use only last hidden state as output

가장 쉬운 사용케이스로 마지막 time-step의 출력값만 사용하는 경우입니다.

가장 흔한 예제로 그림의 감성분석과 같이 텍스트 분류(text classification)의 경우에 단어(토큰)의 갯수 만큼 입력이 RNN에 들어가고, 마지막 time-step의 결과값을 받아서 softmax 함수를 통해 해당 입력 텍스트의 클래스(class)를 예측하는 확률 분포를 근사(approximate)하도록 동작하게 됩니다.

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

이때, 각 time-step 별 입력 단어  $x_t$ 는 one-hot vector로 표현(encoded)되고 embedding layer를 거쳐 정해진 dimension의 word embedding vector로 표현되어 RNN에 입력으로 주어지게 됩니다. 마찬가지로 정답 클래스 또한 one-hot vector가 되어 cross entropy 손실함수(loss function)를 통해 softmax 결과값인 각 클래스 별 확률을 나타낸 (multinoulli) 확률 분포 vector와 비교하여 손실(loss)값을 구하게 됩니다.

$$\text{CrossEntropy}(y_{1:n}, \hat{y}_{1:n}) = \frac{1}{n} \sum_{i=1}^n y_i^T \hat{y}_i$$

### Use all hidden states as output

그리고 또 다른 많이 이용되는 방법은 모든 time-step의 출력값을 모두 사용하는 것입니다. 우리는 이 방법을 언어모델(language modeling)이나 기계번역(machine translation)으로 실습 해 볼 것이지만, 굳이 그런 방법이 아니어도, 문장을 입력으로 주고, 각 단어 별 형태소를 분류(classification)하는 문제라던지 여러가지 방법으로 응용이 가능합니다.

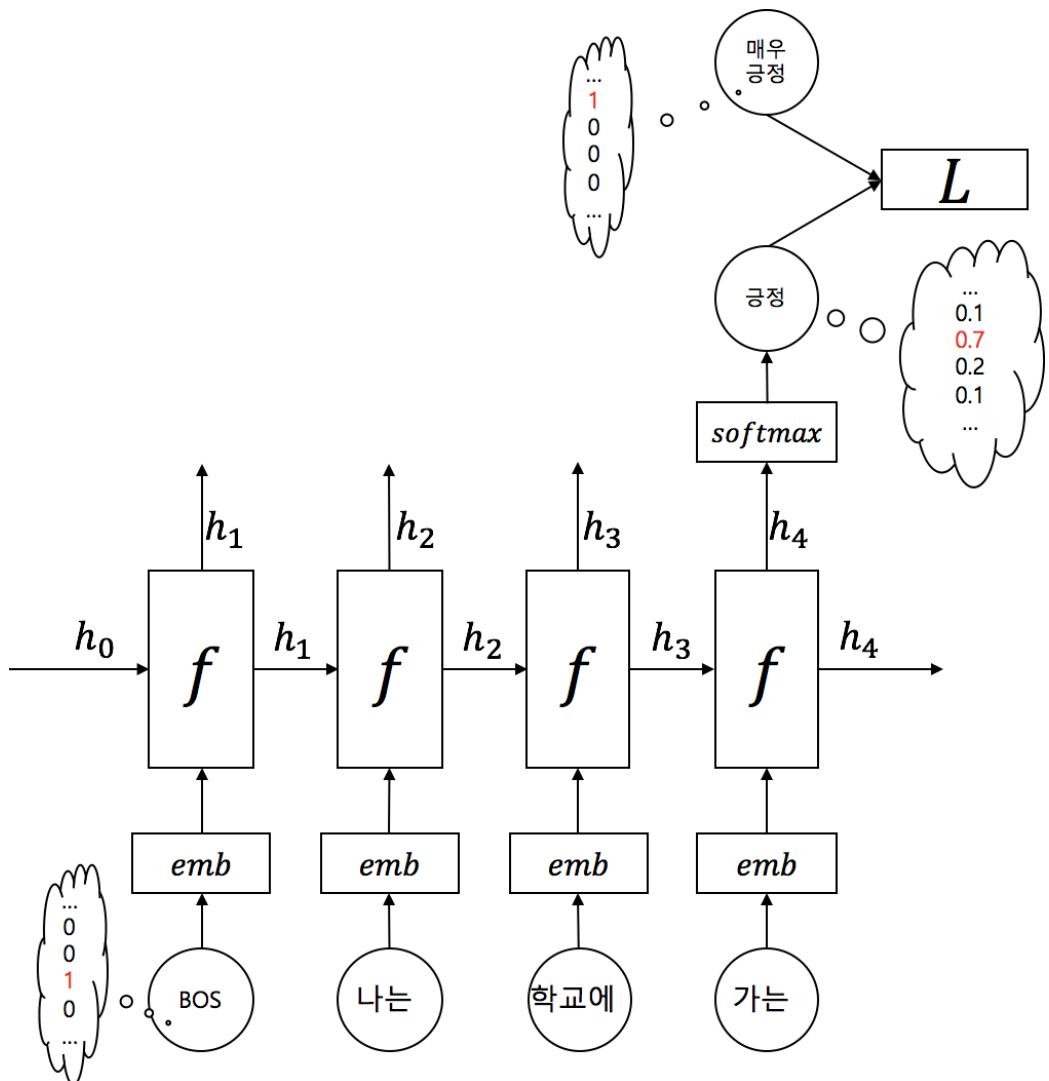


Figure 9: RNN의 마지막 time-step의 출력을 사용 하는 경우

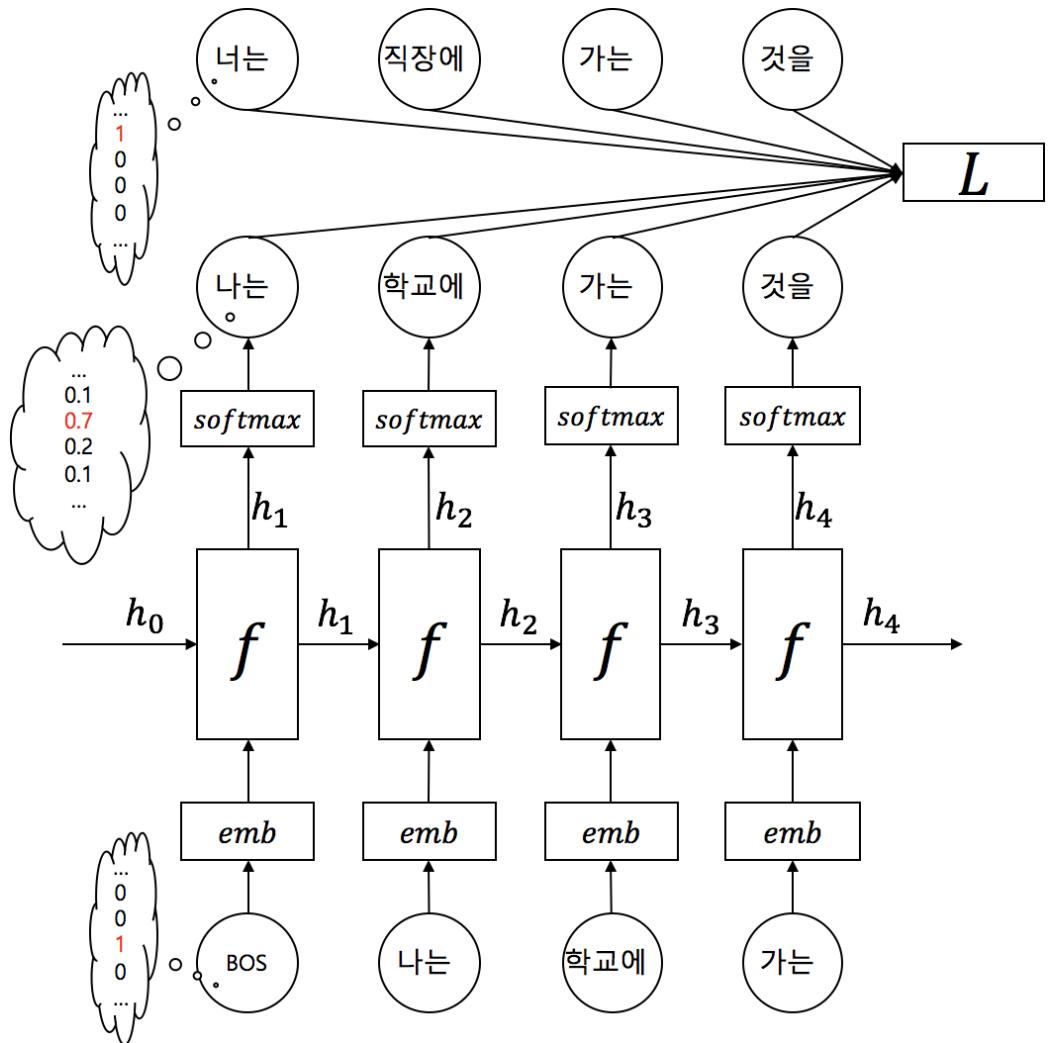


Figure 10: 모든 time-step의 출력을 사용 하는 경우

그림과 같이 각 time-step 별로 입력을 받아 RNN을 거치고 나서, 각 time-step 별로 어떠한 결과물을 출력 하여, 각 time-step 별 정답과 비교하여 손실(loss)를 구합니다. 이때에도 각 단어는 one-hot vector로 표현 될 수 있으며, 그 경우에는 embedding layer를 거쳐 word embedding vector로 변환 된 후, RNN에 입력으로 주어지게 됩니다.

대부분의 경우 RNN은 여러 층(layer)과 양방향(bi-directional)으로 구현 될 수 있습니다. 하지만 입력과 출력이 같은 데이터를 공유 하는 경우에는 bi-directional RNN을 사용할 수 없습니다. 좀 더 구체적으로 설명하면 이전 time-step이 현재 time-step의 입력으로 사용되는 모델 구조의 경우에는 bi-directional RNN을 사용할 수 없습니다. 위의 그림도 그 경우에 해당 합니다. 하지만 형태소 분류기와 같이 출력이 다음 time-step에 입력에 영향을 끼치지 않는 경우에는 bi-directional RNN을 사용할 수 있습니다.

## Conclusion

위와 같이 RNN은 가변길이의 입력을 받아 가변길이의 출력을 내어줄 수 있는 모델입니다. 하지만 기본(Vanilla) RNN은 time-step이 길어질 수록 앞의 데이터를 기억하지 못하는 치명적인 단점이 있습니다. 이를 해결하기 위해서 Long Short Term Memory (LSTM)이나 Gated Recurrent Unit (GRU)와 같은 응용 아키텍쳐들이 나왔고 훌륭한 개선책이 되어 널리 사용되고 있습니다. # Long Short Term Memory (LSTM)

RNN은 가변길이의 입력에 대해서 훌륭하게 동작하지만, 그 길이가 길어지면 오래된 데이터를 잊어버리는 치명적인 단점이 있습니다. 하지만 Long Short Term Memory(LSTM)의 등장으로 RNN을 단점을 보완할 수 있게 되었습니다. LSTM은 기존 RNN의 hidden state 이외에도 별도의 cell state라는 변수(variable)를 두어, 그 기억력을 증가시켰을 뿐만 아니라, 여러가지 게이트(gate)를 두어 기억하거나, 잊어버리거나, 출력하고자 하는 데이터의 양을 상황에 따라서, 마치 수도꼭지를 잠궜다 열듯이, 효과적으로 제어하여 긴 길이의 데이터에 대해서도 효율적으로 대처할 수 있게 되었습니다.

하지만 덕분에 LSTM의 수식은 덕분에 RNN에 비하면 굉장히 복잡해지게 되었고, 더 많아진 파라미터들을 훈련시키기 위해서는 상대적으로 더 많은 데이터들로 더 오래 훈련 해야 합니다. 다행히, 빅데이터의 시대를 맞이하여 범람하는 정보들과, 그래픽카드의 발달로 인한 GPGPU의 빨라진 속도로 인해, 지금은 아무런 어려움 없이 당연하게 LSTM을 사용하는 시대가 되었습니다.

아래는 LSTM의 수식입니다.

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}$$

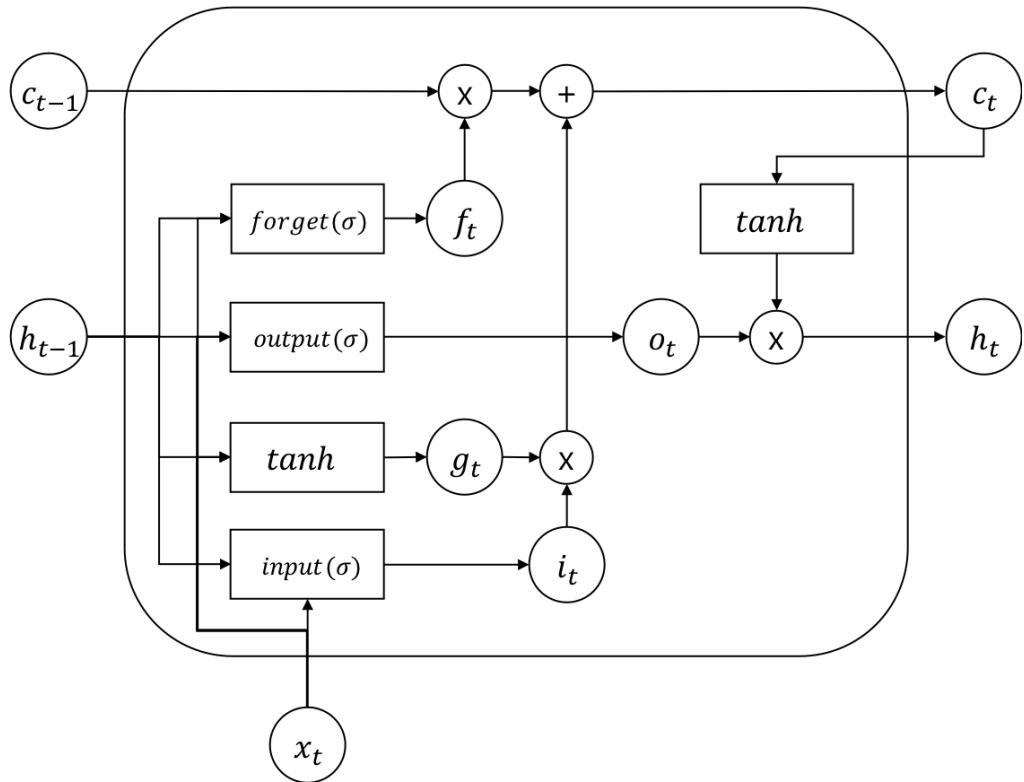


Figure 11: LSTM의 구조

각 게이트(gate) 앞에는 sigmoid( $\sigma$ )가 붙어 0에서 1 사이의 값으로 얼마나 게이트를 열고 닫을지를 결정합니다. 그럼 그 결정된 값에 따라서 cell state  $c_{t-1}$ 와  $g_t$ ,  $c_t$ 가 새롭게 인코딩(encoding) 됩니다.

RNN과 마찬가지로 LSTM 또한 여러층으로 쌓거나, 양방향(bi-directional)으로 구현할 수 있습니다. 더 길어진 길이에 대해서도 RNN보다 훨씬 훌륭하게

대처하지만, 무한정 길어지는 길이에 대처 할 수 있는 것은 아닙니다. 따라서 여전히 길이의 데이터에 대해서 기억하지 못하는 문제점은 아직 남아 있습니다. # Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU)는 뉴욕대학교 조경현 교수가 제안한 방법입니다. 기존 LSTM이 너무 복잡한 모델이므로 좀 더 간단하면서 성능이 비슷한 방법을 제안하였습니다. 마찬가지로 GRU 또한 sigmoid  $\sigma$ 로 구성 된 리셋(reset) 게이트(gate)  $r_t$ 와 업데이트(update) 게이트  $z_t$ 가 있습니다. 마찬가지로 sigmoid 함수로 인해서 게이트의 출력값은 0과 1 사이가 나오므로, 데이터의 흐름을 게이트를 열고 닫아 제어할 수 있습니다. 기존 LSTM 대비 게이트의 숫자가 줄어든 것(4 → 2)을 볼 수 있고, 따라서 게이트에 딸려있던 파라미터들이 그만큼 줄어든 것입니다.

$$\begin{aligned} r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\ z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\ n_t &= \tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{(t-1)} + b_{hn})) \\ h_t &= (1 - z_t)n_t + z_th_{(t-1)} \end{aligned}$$

사실 LSTM 대비 GRU는 더 가벼운 몸집을 자랑하지만, 아직까지는 LSTM을 사용하는 빈도가 더 높은 것이 사실이긴 합니다. 사실 특별히 성능의 차이가 있다고 하기보단, LSTM과 GRU의 learning-rate나 hidden size등의 하이퍼파라미터(hyper-parameter)가 다르므로 사용 모델에 따라서 다시 파라미터 셋팅을 연구 해야 하므로, 쉽게 이것저것 사용하기 힘든 부분도 있을 뿐더러, 연구자의 취향에 가까운 것 같습니다. # Gradient Clipping

RNN은 BPTT를 통해서 gradient를 구합니다. 따라서 출력의 길이에 따라서 gradient의 크기가 달라지게 됩니다. 즉, 길이가 길수록 자칫 gradient가 너무 커질 수 있기 때문에, learning rate를 조절하는 일이 필요 합니다. 너무 큰 learning rate를 사용하게 되면 gradient descent에서 step의 크기가 너무 커져버려 잘못된 방향으로 학습 및 발산(gradient exploding) 해 버릴 수 있기 때문입니다. 이 경우 가장 쉬운 대처 방법은 learning rate를 아주 작은 값을 취하는 것입니다. 하지만 작은 learning rate를 사용할 경우, 평소 상황에서 너무 적은 양만 배우므로 훈련 속도가 매우 느려질 것입니다. 즉, 길이는 가변이므로 learning rate를 그때그때 알맞게 최적의 값을 찾아 조절 해 주는 것은 매우 어려운 일이 될 것입니다. 이때 gradient clipping이 큰 힘을 발휘합니다.

Gradient clipping은 신경망 파라미터  $\theta$ 의 norm (보통 L2 norm)을 구하고, 이 norm의 크기를 제한하는 방법입니다. 따라서 gradient vector는 방향은 유지하되,

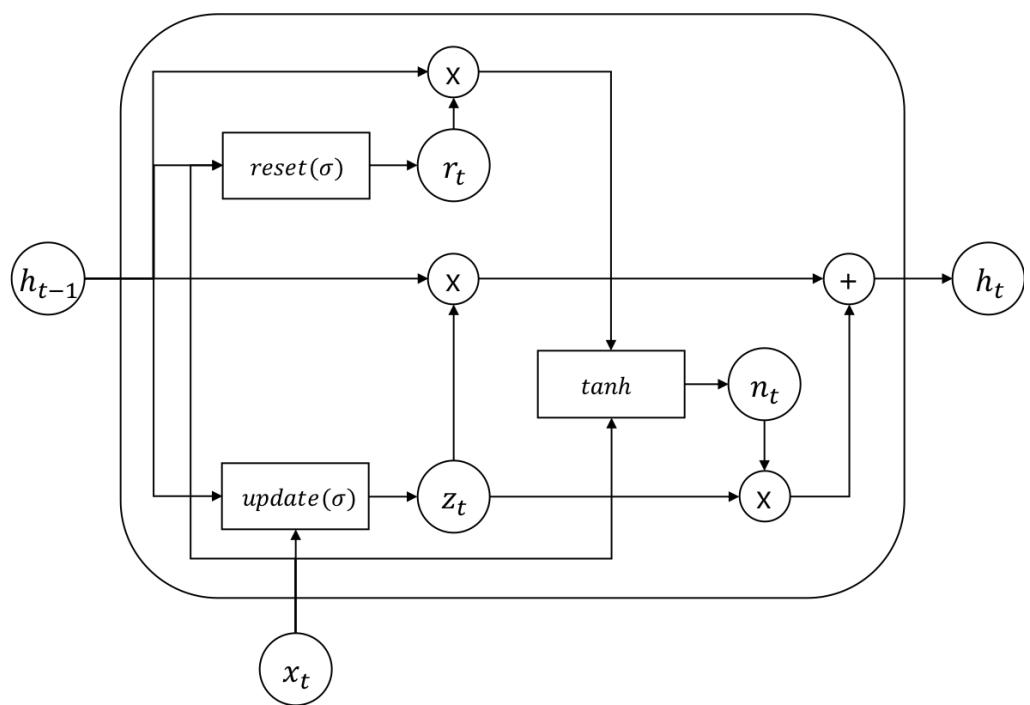


Figure 12: GRU의 구조

그 크기는 학습이 망가지지 않은 정도로 줄어들 수 있게 됩니다. 물론 norm의 maximum value를 바로 사용자가 정의 해 주어야 하기 때문에, 또 하나의 hyper-parameter가 생기게 되지만, 큰 norm을 가진 gradient vector의 경우에만 gradient clipping을 수행하기 때문에, 능동적으로 learning rate를 조절하는 것과 비슷한 효과를 가질 수 있습니다. 따라서 gradient clipping은 RNN 계열의 학습 및 훈련을 할 때 널리 사용되는 방법입니다.

$$\frac{\partial \epsilon}{\partial \theta} \leftarrow \begin{cases} \frac{\text{threshold}}{\|\hat{g}\|} \hat{g} & \text{if } \|\hat{g}\| \geq \text{threshold} \\ \hat{g} & \text{otherwise} \end{cases}$$

where  $\hat{g} = \frac{\partial \epsilon}{\partial \theta}$ .

수식을 보면, gradient norm이 정해진 threshold(역치)보다 클 경우에, gradient 벡터를 threshold 보다 큰 만큼의 비율로 나누어 줍니다. 따라서 gradient는 항상 threshold 보다 작으면 이는 gradient exploding을 방지함과 동시에, gradient의 방향을 유지해주기 때문에 모델 파라미터  $\theta$ 가 학습해야 하는 방향은 잃지 않습니다.

PyTorch에서도 기능을 `torch.nn.utils.clip_grad_norm_` 이라는 함수를 제공하고 있으므로 매우 쉽게 사용 할 수 있습니다.

## Text Classification



Figure 1: Thomas Bayes – Image from Wikipedia

## Text Classification

텍스트 분류(text classification)는 텍스트, 문장 또는 문서(문장들)를 입력으로 받아 사전에 정의된 클래스(class)들 중에서 어떤 클래스에 속하는지 분류 하는 과정을 의미합니다. 따라서 텍스트 분류는 어쩌면 이 책에서 (그 난이도에 비해서) 독자들에게 가장 쓸모가 있는 챕터가 될 수도 있습니다. 아래와 같이 그 응용 분야가 매우 다양하기 때문입니다.

문제	클래스 예
감성분석(Sentiment Analysis)	긍정(positive), 중립(neutral), 부정(negative)
스팸 메일 탐지(Spam E-mail Detection)	정상(normal), 스팸(spam)
사용자 의도 분류(User Intent Classification)	명령, 질문, 응답 등
주제 분류	각 주제
카테고리 분류	각 카테고리

이처럼 무언가 분류해야 하는 문제가 있다면 대부분 텍스트 분류 문제에 속한다고 볼 수 있습니다. 사실 자연어처리에서 텍스트 분류의 문제일뿐, 다른 기계학습 분야에 적용도 얼마든지 가능합니다. 예를 들어 주식 가격과 같은 시계열(time-series) 데이터를 입력으로 받아 주식의 오름세를 예측 해 볼 수도 있을 것 입니다.

딥러닝 이전에는 Naive Bayes, SVM(Support Vector Machine) 등 다양한 방법이 존재하였습니다. 이번 챕터에서는 딥러닝 이전의 가장 간단한 방식인 naive bayes 방식과 딥러닝 방식들을 소개 하도록 하겠습니다. # Naive Bayes for Text Classification

Naive Bayes는 매우 간단하지만 정말 강력한 방법입니다. 의외로 기대 이상의 성능을 보여줄 때가 많습니다. 물론 단어를 여전히 discrete한 심볼로 다루기 때문에, 여전히 아쉬운 부분이 많습니다. 이번 섹션에서는 Naive Bayes를 통해서 텍스트를 분류하는 방법을 살펴 보겠습니다.

## Maximum A Posterior

Naive Bayes를 소개하기에 앞서 Bayes Theorem(베이즈 정리)을 짚고 넘어가지 않을 수 없습니다. Thomas Bayes(토마스 베이즈)가 정립한 이 정리에 따르면 조건부 확률은 아래와 같이 표현 될 수 있으며, 각 부분은 명칭을 갖고 있습니다. 이 이름들에 대해서는 앞으로 매우 친숙해져야 합니다.

$$\underbrace{P(Y|X)}_{posterior} = \frac{\overbrace{P(X|Y)}^{likelihood} \overbrace{P(Y)}^{prior}}{\underbrace{P(X)}_{evidence}}$$

수식	영어 명칭	한글 명칭
$P(Y X)$	Posterior	사후 확률
$P(X Y)$	Likelihood	가능도(우도)
$P(Y)$	Prior	사전 확률
$P(X)$	Evidence	증거

우리가 풀고자하는 대부분의 문제들은  $P(X)$ 는 구하기 힘들기 때문에, 보통은 아래와 같이 접근 하기도 합니다.

$$P(Y|X) \propto P(X|Y)P(Y)$$

위의 성질을 이용하여 주어진 데이터  $X$ 를 만족하며 확률을 최대로 하는 클래스  $Y$ 를 구할 수 있습니다. 이처럼 posterior 확률을 최대화(maximize)하는  $y$ 를 구하는 것을 Maximum A Posterior (MAP)라고 부릅니다. 그 수식은 아래와 같습니다.

$$\hat{y}_{MAP} = \operatorname{argmax}_{y \in \mathcal{Y}} P(Y = y|X)$$

다시한번 수식을 살펴보면,  $X$ (데이터)가 주어졌을 때, 가능한 클래스의 set  $\mathcal{Y}$  중에서 posterior를 최대로 하는 클래스  $y$ 를 선택하는 것입니다.

이와 마찬가지로  $X$ (데이터)가 나타날 likelihood 확률을 최대로 하는 클래스  $y$ 를 선택하는 것을 Maximum Likelihood Estimation (MLE)라고 합니다.

$$\hat{y}_{MLE} = \operatorname{argmax}_{y \in \mathcal{Y}} P(X|Y = y)$$

MLE는 주어진 데이터  $X$ 와 클래스 레이블(label)  $Y$ 가 있을 때, parameter  $\theta$ 를 훈련하는 방법으로도 많이 사용 됩니다.

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(Y|X, \theta)$$

## MLE vs MAP

경우에 따라 MAP는 MLE에 비해서 좀 더 정확할 수 있습니다. prior(사전)확률이 반영되어 있기 때문입니다. 예를 들어보죠.

만약 범죄현장에서 발자국을 발견하고 사이즈를 측정했더니 범인은 신발사이즈(데이터,  $X$ ) 155를 신는 사람인 것으로 의심 됩니다. 이때, 범인의 성별(클래스,  $Y$ )을 예측해 보도록 하죠.

성별 클래스의 set은  $Y = \{male, female\}$  입니다. 신발사이즈  $X$ 는 5단위의 정수로 이루어져 있습니다.  $X = \{\dots, 145, 150, 155, 160, \dots\}$

신발사이즈 155는 남자 신발사이즈 치곤 매우 작은 편입니다. 따라서 우리는 보통 범인을 여자라고 특정할 것 같습니다. 다시 말하면, 남자일 때 신발사이즈 155일 확률  $P(X = 155|Y = male)$ 은 여자일 때 신발사이즈 155일 확률  $P(X = 155|Y = female)$ 일 확률 보다 낮습니다.

보통의 경우 남자와 여자의 비율은 0.5로 같기 때문에, 이는 큰 상관이 없는 예측입니다. 하지만 범죄현장이 만약 군부대였다면 어떻게 될까요? 남녀 성비는  $P(Y = male) >> P(Y = female)$ 로 매우 불균형 할 것입니다.

이때, 이미 갖고 있는 likelihood에 prior를 곱해주면 posterior를 최대화 하는 클래스를 더 정확하게 예측 할 수 있습니다.

$$P(Y = male|X = 155) \propto P(X = 155|Y = male)P(Y = male)$$

## Naive Bayes

Naive Bayes는 MAP를 기반으로 동작합니다. 대부분의 경우 posterior를 바로 구하기 어렵기 때문에, likelihood와 prior의 곱을 통해 클래스  $Y$ 를 예측 합니다. 먼저 우리가 알고자 하는 값인 posterior 확률은 아래와 같을 것 입니다.

$$P(Y = c|X = w_1, w_2, \dots, w_n)$$

이때,  $X$ 가 다양한 feature(특징)들로 이루어진 데이터라면, 훈련 데이터에서 매우 희소(rare)할 것이므로 posterior 뿐만 아니라, likelihood  $P(X = w_1, w_2, \dots, w_n|Y = c)$ 을 구하기 어려울 것 입니다. 왜냐하면 보통 확률을 코퍼스의 출현빈도를 통해 추정할 수 있는데, feature가 복잡할 수록 likelihood 또는 posterior를 만족하는 경우는 코퍼스에 매우 드물 것이기 때문입니다. 그렇다고 코퍼스에 없는 feature의 조합이라고 해서 확률값을 0으로 추정하는 것도 맞지 않습니다.

이때 Naive Bayes가 강력한 힘을 발휘 합니다. 각 feature들이 상호 독립적이라고 가정하는 것 입니다. 그럼 joint probability를 각 확률의 곱으로 근사(approximate)할

수 있습니다. 이 과정을 수식으로 표현하면 아래와 같습니다.

$$\begin{aligned}
 P(Y = c|X = w_1, w_2, \dots, w_n) &\propto P(X = w_1, w_2, \dots, w_n|Y = c)P(Y = c) \\
 &\approx P(w_1|c)P(w_2|c)\cdots P(w_n|c)P(c) \\
 &= \prod_{i=1}^n P(w_i|c)P(c)
 \end{aligned}$$

따라서, 우리가 구하고자 하는 MAP를 활용한 클래스는 아래와 같이 posterior를 최대화하는 클래스가 되고, 이는 Naive Bayes의 가정에 따라 각 feature들의 확률의 곱에 prior 확률을 곱한 값을 최대화 하는 클래스와 같을 것입니다.

$$\begin{aligned}
 \hat{c}_{MAP} &= \operatorname{argmax}_{c \in \mathcal{C}} P(Y = c|X = w_1, w_2, \dots, w_n) \\
 &\approx \operatorname{argmax}_{c \in \mathcal{C}} \prod_{i=1}^n P(w_i|c)P(c)
 \end{aligned}$$

이때 사용되는 prior 확률은 아래와 같이 실제 데이터에서 나타난 횟수를 세어 구할 수 있습니다.

$$P(Y = c) \approx \frac{\text{Count}(c)}{\sum_{i=1}^{|\mathcal{C}|} \text{Count}(c_i)}$$

또한, 각 feature 별 likelihood 확률도 데이터에서 바로 구할 수 있습니다. 만약 모든 feature들의 조합이 데이터에서 나타난 횟수를 통해 확률을 구하려 하였다면 sparseness(희소성) 문제 때문에 구할 수 없었을 것입니다. 하지만 Naive Bayes의 가정(각 feature들은 독립적)을 통해서 쉽게 데이터에서 출현 빈도를 활용할 수 있게 되었습니다.

$$P(w|c) \approx \frac{\text{Count}(w, c)}{\sum_{j=1}^{|V|} \text{Count}(w_j, c)}$$

이처럼 간단한 가정을 통하여 데이터의 sparsity를 해소하여, 간단하지만 강력한 방법으로 우리는 posterior를 최대화하는 클래스를 예측 할 수 있게 되었습니다.

## Example: Sentiment Analysis

그럼 실제 예제로 접근해 보죠. 감성분석은 가장 많이 활용되는 텍스트 분류 기법입니다. 사용자의 댓글이나 리뷰 등을 긍정 또는 부정으로 분류하여 마케팅이나 서비스 향상에 활용하고자 하는 방법입니다. 물론 실제로 딥러닝 이전에는 Naive Bayes를 통해 접근하기보단, 각 클래스 별 어휘 사전(vocabulary)을 만들어 해당 어휘의 등장 여부에 따라 판단하는 방법을 주로 사용하곤 하였습니다.

$$\begin{aligned}\mathcal{C} &= \{\text{pos}, \text{neg}\} \\ \mathcal{D} &= \{d_1, d_2, \dots\}\end{aligned}$$

위와 같이 긍정(*pos*)과 부정(*neg*)으로 클래스가 구성( $\mathcal{C}$ 되어 있고, 문서  $d$ 로 구성된 데이터  $\mathcal{D}$ 가 있습니다.

이때, 우리에게 “I am happy to see this movie!”라는 문장이 주어졌을 때, 이 문장이 긍정인지 부정인지 판단해 보겠습니다.

$$\begin{aligned}P(\text{pos}|I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !) &= \frac{P(I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !|\text{pos})P(\text{pos})}{P(I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !)} \\ &\approx \frac{P(I|\text{pos})P(\text{am}|\text{pos})P(\text{happy}|\text{pos}) \cdots P(!|\text{pos})P(\text{pos})}{P(I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !)}\end{aligned}$$

Naive Bayes의 수식을 활용하여 단어의 조합에 대한 확률을 각각 분해할 수 있습니다. 그리고 그 확률들은 아래와 같이 데이터  $\mathcal{D}$ 에서의 출현 빈도를 통해 구할 수 있습니다.

$$\begin{aligned}P(\text{happy}|\text{pos}) &\approx \frac{\text{Count}(\text{happy}, \text{pos})}{\sum_{j=1}^{|V|} \text{Count}(w_j, \text{pos})} \\ P(\text{pos}) &\approx \frac{\text{Count}(\text{pos})}{|\mathcal{D}|}\end{aligned}$$

마찬가지로 부정 감성에 대해 같은 작업을 반복 할 수 있습니다.

$$\begin{aligned}
P(\text{neg}|I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !) &= \frac{P(I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !|\text{neg})P(\text{neg})}{P(I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !)} \\
&\approx \frac{P(I|\text{neg})P(\text{am}|\text{neg})P(\text{happy}|\text{neg}) \cdots P(!|\text{neg})P(\text{neg})}{P(I, \text{am}, \text{happy}, \text{to}, \text{see}, \text{this}, \text{movie}, !)}
\end{aligned}$$

$$\begin{aligned}
P(\text{happy}|\text{neg}) &\approx \frac{\text{Count}(\text{happy}, \text{neg})}{\sum_{j=1}^{|V|} \text{Count}(w_j, \text{neg})} \\
P(\text{neg}) &\approx \frac{\text{Count}(\text{neg})}{|\mathcal{D}|}
\end{aligned}$$

## Add-one Smoothing

여기에는 문제가 하나 있습니다. 만약 훈련 데이터에서  $\text{Count}(\text{happy}, \text{neg})$ 가 0이었다면  $P(\text{happy}|\text{neg}) = 0$ 이 되겠지만, 그저 훈련 데이터에 존재하지 않는 경우라고 해서 실제 출현 확률을 0으로 여기는 것은 매우 위험한 일입니다.

$$P(\text{happy}|\text{neg}) \approx \frac{\text{Count}(\text{happy}, \text{neg})}{\sum_{j=1}^{|V|} \text{Count}(w_j, \text{neg})} = 0,$$

where  $\text{Count}(\text{happy}, \text{neg}) = 0$ .

따라서 우리는 이런 경우를 위하여 각 출현 횟수에 1을 더해주어 간단하게 문제를 완화할 수 있습니다. 물론 완벽한 해결법은 아니지만, Naive Bayes의 가정과 마찬가지로 간단하고 강력합니다.

$$\tilde{P}(w|c) = \frac{\text{Count}(w, c) + 1}{\left( \sum_{j=1}^{|V|} \text{Count}(w_j, c) \right) + |V|}$$

## Pros and Cons

위와 같이 Naive Bayes를 통해서 단순히 출현빈도를 세는 것처럼 쉽고 간단하지만 강력하게 감성분석을 구현 할 수 있습니다. 하지만 문장 “I am not happy to see this

movie!”라는 문장이 주어지면 어떻게 될까요? “not”이 추가 되었을 뿐이지만 문장의 뜻은 반대가 되었습니다.

$$P(\text{pos}|I, am, not, happy, to, see, this, movie, !)$$
$$P(\text{neg}|I, am, not, happy, to, see, this, movie, !)$$

“not”은 “happy”를 수식하기 때문에 두 단어를 독립적으로 보는 것은 옳지 않을 수 있습니다.

$$P(not, happy) \neq P(not)P(happy)$$

사실 문장은 단어들이 순서대로 나타나서 의미를 이루기 때문에, 각 단어의 출현 여부도 중요하지만, 각 단어 사이의 순서로 인해 생기는 관계도 무시할 수 없습니다. 하지만 Naive Bayes의 가정은 언어의 이런 특징을 단순화하여 접근하기 때문에 한계가 있습니다.

하지만, 레이블(labeled) 데이터가 매우 적은 경우에는 딥러닝보다 이런 간단한 방법을 사용하는 것이 훨씬 더 나은 대안이 될 수도 있습니다. 이처럼 Naive Bayes는 매우 간단하고 강력하지만, Naive Bayes를 강력하게 만들어준 가정이 가져오는 단점 또한 명확합니다. # CNN Based Method

이번 섹션에서는 Convolutional Nueral Network (CNN) Layer를 활용한 텍스트 분류에 대해 다루어 보겠습니다. CNN을 활용한 방법은 [Kim at el.2014]에 의해서 처음 제안되었습니다. 사실 이전까지 딥러닝을 활용한 자연어처리는 Recurrent Nueral Networ (RNN)에 국한되어 있는 느낌이 매우 강했습니다. 텍스트 문장은 여러 단어로 이루어져 있고, 그 문장의 길이가 문장마다 상이하며, 문장 내의 단어들은 같은 문장 내의 단어에 따라서 영향을 받기 때문입니다.

좀 더 비약적으로 표현하면  $t$  time-step에 등장하는 단어  $w_t$ 는 이전 time-step에 등장한 단어들  $w_1, \dots, w_{t_1}$ 에 의존하기 때문입니다. (물론 실제로는  $t$  이후에 등장하는 단어들로부터도 영향을 받습니다.) 따라서 시간 개념이 도입되어야 하기 때문에, RNN의 사용은 불가피하다고 생각되었습니다. 하지만 앞서 소개한 [Kim at el.2014] 논문에 의해서 새로운 시각이 열리게 됩니다.

## Convolution Operation

사실 널리 알려졌다시피, CNN은 영상처리(or Computer Vision) 분야에서 매우 큰 성과를 거두고 있었습니다. CNN의 동기 자체가, 기존의 전통적인 영상처리에서

사용되던 각종 convolution 필터(filter or kernel)를 자동으로 학습하기 위함이기 때문입니다.

### Convolution Filter

전통적인 영상처리 분야에서는 손으로 한땀한땀 만들어낸 필터를 사용하여 윤곽선을 검출하는 등의 전처리 과정을 거쳐, 얻어낸 피쳐(feature)들을 통해 객체 탐지(object detection) 등을 구현하곤 하였습니다. 예를 들어 주어진 이미지에서 윤곽선(edge)을 찾기 위한 convolution 필터는 아래와 같습니다.

-1	0	+1
-2	0	+2
-1	0	+1

$G_x$

+1	+2	+1
0	0	0
-1	-2	-1

$G_y$

Figure 2: Sobel Filters for vertical and horizontal edges

이 필터를 이미지에 적용하면 아래와 같은 결과를 얻을 수 있습니다.

이처럼 전처리 서브모듈에서 여러 필터들을 문제에 따라 적용하여 피쳐들을 얻어낸 이후에, 다음 서브모듈을 적용하여 주어진 문제를 해결하는 방식이었습니다.

### Convolutional Neural Network Layer

만약 문제에 따라서 필요한 convolution 필터를 자동으로 찾아준다면 어떻게 될까요? CNN이 바로 그러한 역할을 해주게 됩니다. Convolution 연산을 통해 feed-forward 된 값에 back-propagation을 하여, 더 나은 convolution 필터 값을

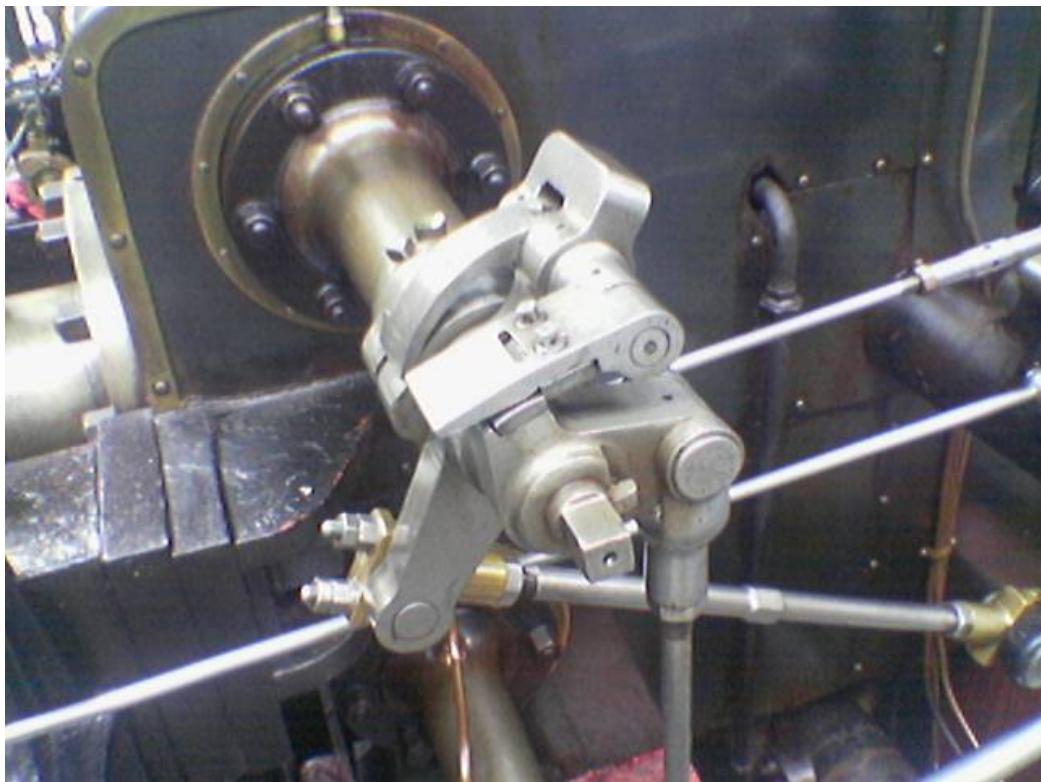


Figure 3: An image before Sobel filter (from Wikipedia)

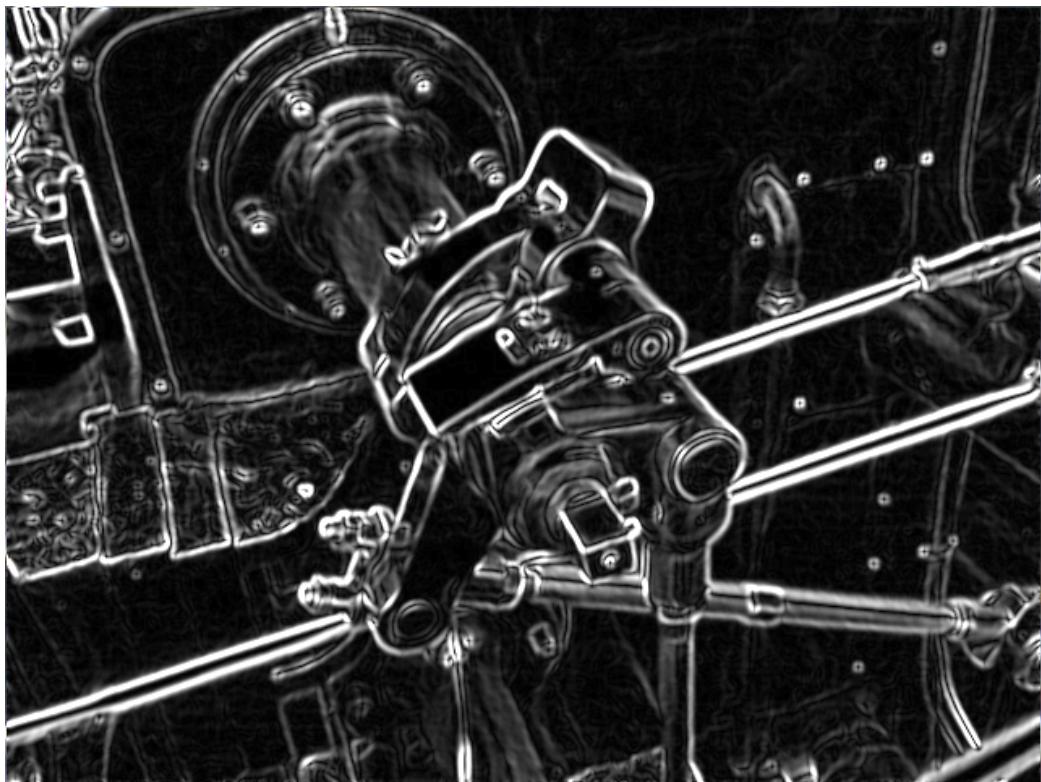


Figure 4: Image after applying Sobel filter (from Wikipedia)

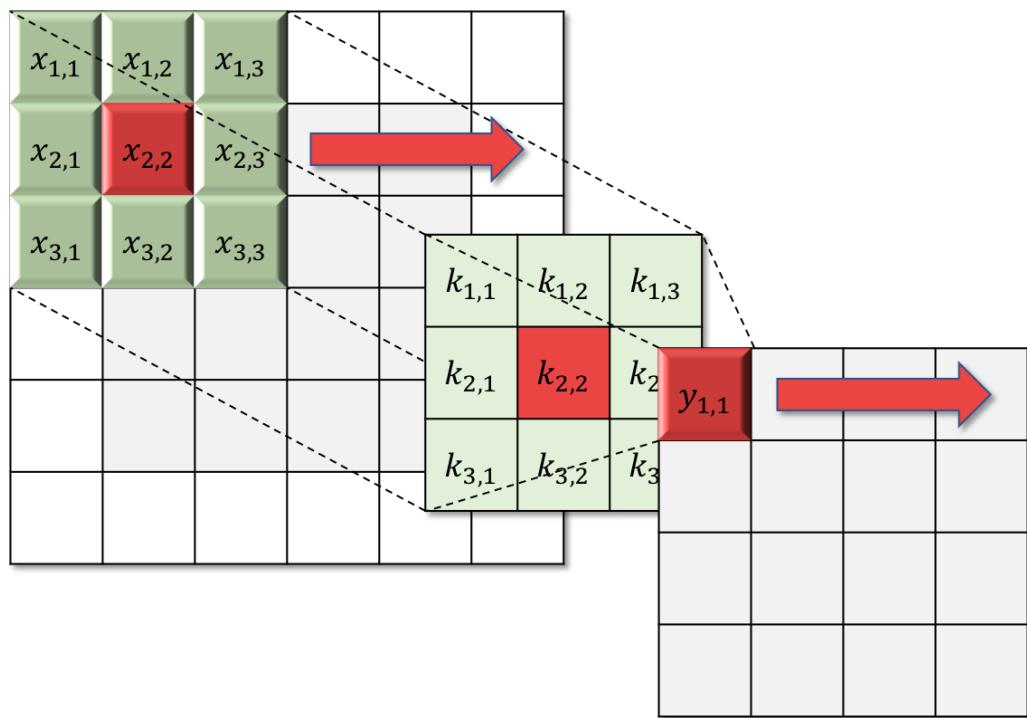


Figure 5: Convolution 연산을 적용하는 과정

찾아나가게 됩니다. 따라서 마지막에 loss 값이 수렴 한 이후에는, 해당 문제에 딱 맞는 여러 종류의 convolution 필터를 찾아낼 수 있게 되는 것입니다.

$$\begin{aligned}y_{1,1} &= x_{1,1} * k_{1,1} + \cdots + x_{3,3} * k_{3,3} \\&= \sum_{i=1}^3 \sum_{j=1}^3 x_{i,j} * k_{i,j}\end{aligned}$$

Convolution 필터 연산의 forward는 위와 같습니다. 필터(또는 커널)가 주어진 이미지 위에서 차례대로 convolution 연산을 수행합니다. 보다시피, 상당히 많은 연산이 병렬(parallel)로 수행될 수 있음을 알 수 있습니다.

기본적으로는 convolution 연산의 결과물은 필터의 크기에 따라 입력에 비해서 크기가 줄어듭니다. 위의 그림에서도 필터의 크기가  $3 \times 3$  이므로,  $6 \times 6$  입력에 적용하면  $4 \times 4$  크기의 결과물을 얻을 수 있습니다. 따라서 입력과 같은 크기를 유지하기 위해서는 결과물의 바깥에 패딩(padding)을 추가하여 크기를 유지할 수도 있습니다.

이처럼 CNN은 문제를 해결하기 위한 패턴을 감지하는 필터를 자동으로 구성하여주는 역할을 통해, 영상처리 등의 Computer Vision 분야에서 빼놓을 수 없는 매우 중요한 역할을 하고 있습니다. 또한, 이미지 뿐만 아니라 아래와 같이 음성 분야에서도 효과를 보고 있습니다. Audio 신호의 경우에도 푸리에 변환을 통해서 2차원의 시계열 데이터를 얻을 수 있습니다. 이렇게 얻어진 데이터 대해서도 마찬가지로 패턴을 찾아내는 convolution 연산이 필요합니다.

## How to Apply CNN on Text Classification

그렇다면 텍스트 분류과정에는 어떻게 CNN을 적용하는 것일까요? 텍스트에 무슨 윤곽선과 같은 패턴이 있는 것일까요? 사실 단어들을 embedding vector로 변환하면, 1차원(vector)이 됩니다. 이때, 1-dimensional CNN을 수행하면, 이제 텍스트에서도 CNN이 효과를 발휘할 수 있게 됩니다.

$$y_{n,m} = \sum_{i=1}^d k_i * x_{n,i}, \text{ where } d = \text{word vec dim.}$$

좀 더 구체적으로 예를 들어, 주어진 문장에 대해서 긍정/부정 분류를 하는 문제를 생각해 볼 수 있습니다. 그럼 문장은 여러 단어로 이루어져 있고, 각각의 단어는

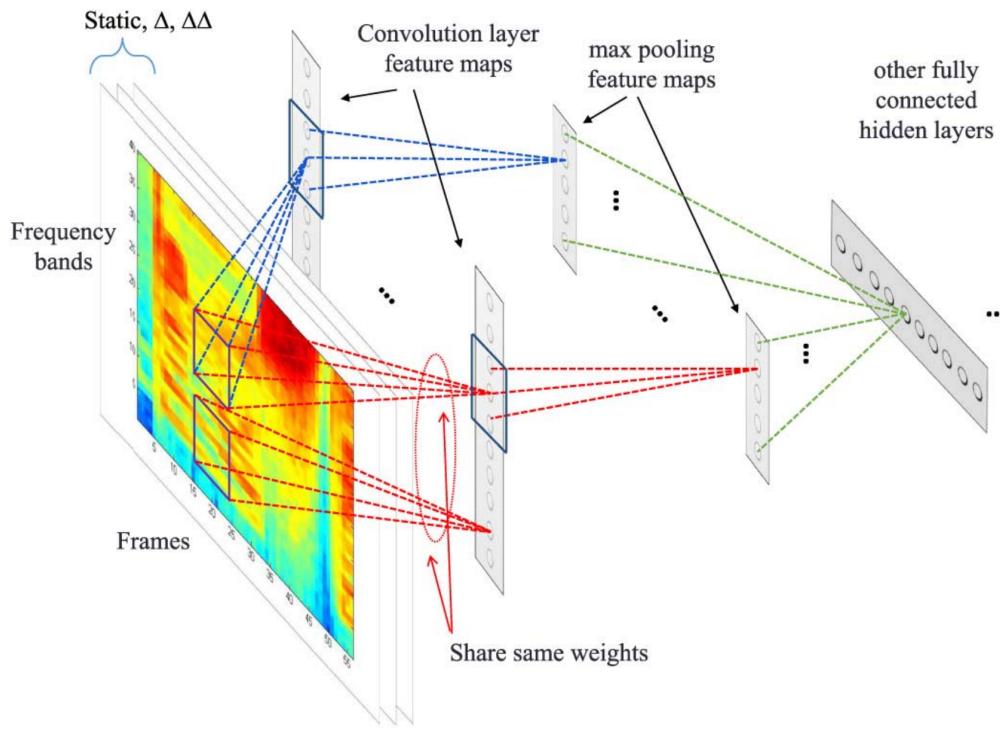


Figure 6: Example of convolutional neural network for speech recognition Abdel-Hamid et al,2014

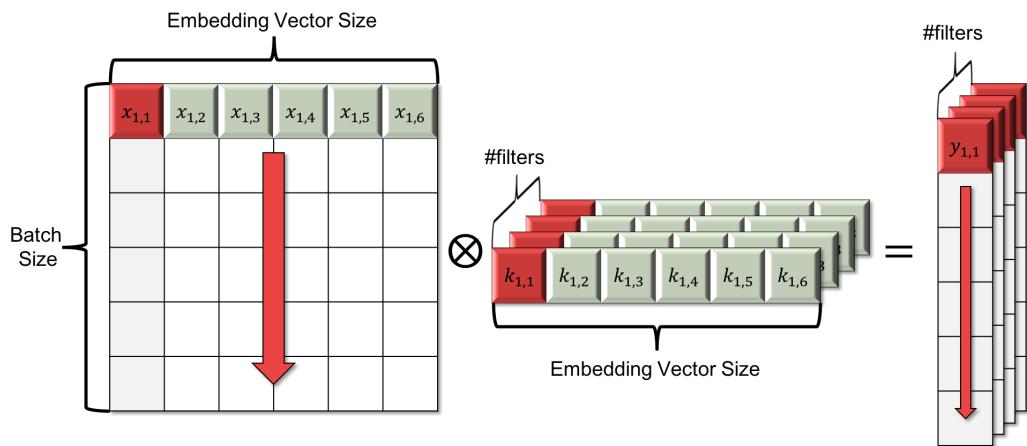


Figure 7: 1D Convolutional neural network

embedding layer를 통해 embedding vector로 변환 된 상태입니다. 각 단어의 embedding vector는 비슷한 의미를 가진 단어일 수록 비슷한 값의 vector 값을 가지도록 될 것 입니다.

예를 들어 ‘good’이라는 단어는 그에 해당하는 embedding vector로 구성되어 있을 것 입니다. 그리고 ‘better’, ‘best’, ‘great’ 등의 단어들도 ‘good’과 비슷한 vector 값을 갖고 있을 것 입니다. 이때, 쉽게 예상할 수 있듯이, ‘good’은 긍정/부정 분류에 있어서 긍정을 나타내는 매우 중요한 신호로 작용 할 수 있을 것 입니다.

그렇다면 ‘good’에 해당하는 embedding vector의 패턴을 감지하는 filter를 가질 수 있다면, ‘good’ 뿐만 아니라, ‘better’, ‘best’, ‘great’ 등의 단어들도 함께 감지할 수 있을 것 입니다. 한발 더 나아가, 단어들의 조합(시퀀스)의 패턴을 감지하는 filter도 학습할 수 있을 것 입니다. 예를 들어 ‘good taste’, ‘worst ever’ 등과 비슷한 embedding vector들로 구성된 매트릭스( $M \in \mathbb{R}^{w \times d}$ )를 감지할 수 있을 것 입니다.

[Kim et al. 2014]에서는 이를 이용하여 CNN 레이어만을 사용한 훌륭한 성능의 텍스트 분류 방법을 제시하였습니다.

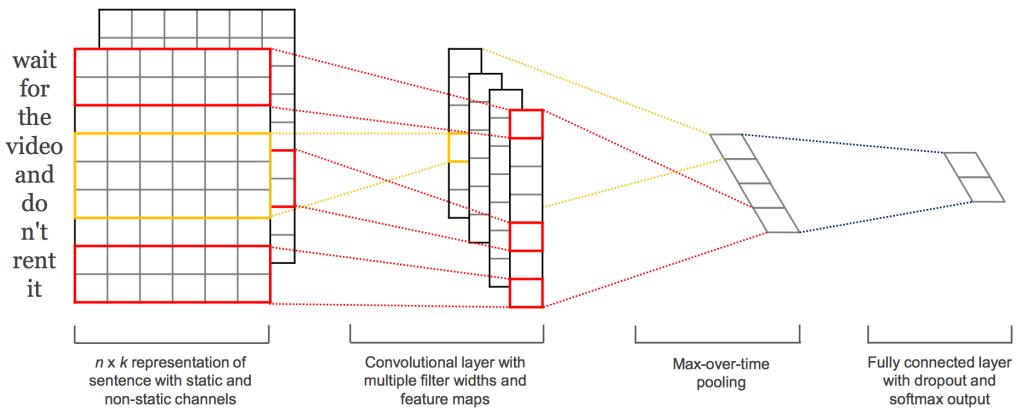


Figure 8: CNN for text classification arthictecture [Kim et al. 2014]

여러 단어로 이루어진 가변 길이의 문장을 입력으로 받아, 각 단어들을 embedding vector로 변환 후, 단어 별로 여러가지 필터를 적용하여 필요한 패턴을 감지합니다. 문제는 문장의 길이가 문장마다 다르기 때문에, 필터를 적용한 결과물의 크기도 다를 것입니다. 이때, max pooling layer를 적용하여 가변 길이의 변수를 제거할 수 있습니다. Max pooling 결과의 크기는 필터의 갯수와 같을 것입니다. 이제 이 위에 linear layer + softmax를 사용하여 각 class 별 확률을 구할 수 있습니다.

## 코드

```
import torch
import torch.nn as nn

class CNNClassifier(nn.Module):

    def __init__(self,
                 input_size,
                 word_vec_dim,
                 n_classes,
                 dropout_p=.5,
                 window_sizes=[3, 4, 5],
                 n_filters=[100, 100, 100]):
        self.input_size = input_size # vocabulary size
        self.word_vec_dim = word_vec_dim
        self.n_classes = n_classes
        self.dropout_p = dropout_p
        # window_size means that how many words a pattern covers.
        self.window_sizes = window_sizes
        # n_filters means that how many patterns to cover.
        self.n_filters = n_filters

        super().__init__()

        self.emb = nn.Embedding(input_size, word_vec_dim)
        # Since number of convolution layers would be vary depend on len(window_
        # we use 'setattr' and 'getattr' methods to add layers to nn.Module obje
        for window_size, n_filter in zip(window_sizes, n_filters):
            cnn = nn.Conv2d(in_channels=1,
                           out_channels=n_filter,
                           kernel_size=(window_size, word_vec_dim)
                           )
            setattr(self, 'cnn-%d-%d' % (window_size, n_filter), cnn)
        # Because below layers are just operations,
```

```

# (it does not have learnable parameters)
# we just declare once.
self.relu = nn.ReLU()
self.dropout = nn.Dropout(dropout_p)
# An input of generator layer is max values from each filter.
self.generator = nn.Linear(sum(n_filters), n_classes)
# We use LogSoftmax + NLLLoss instead of Softmax + CrossEntropy
self.activation = nn.LogSoftmax(dim=-1)

def forward(self, x):
    # |x| = (batch_size, length)
    x = self.emb(x)
    # |x| = (batch_size, length, word_vec_dim)
    min_length = max(self.window_sizes)
    if min_length > x.size(1):
        # Because some input does not long enough for maximum length of window
        # we add zero tensor for padding.
        pad = x.new(x.size(0), min_length - x.size(1), self.word_vec_dim).zero_()
        # |pad| = (batch_size, min_length - length, word_vec_dim)
        x = torch.cat([x, pad], dim=1)
        # |x| = (batch_size, min_length, word_vec_dim)

    # In ordinary case of vision task, you may have 3 channels on tensor,
    # but in this case, you would have just 1 channel,
    # which is added by 'unsqueeze' method in below:
    x = x.unsqueeze(1)
    # |x| = (batch_size, 1, length, word_vec_dim)

    cnn_outs = []
    for window_size, n_filter in zip(self.window_sizes, self.n_filters):
        cnn = getattr(self, 'cnn-%d-%d' % (window_size, n_filter))
        cnn_out = self.dropout(self.relu(cnn(x)))
        # |x| = (batch_size, n_filter, length - window_size + 1, 1)

        # In case of max pooling, we does not know the pooling size,
        # because it depends on the length of the sentence.
        # Therefore, we use instant function using 'nn.functional' package.
        # This is the beauty of PyTorch. :)
```

```

cnn_out = nn.functional.max_pool1d(input=cnn_out.squeeze(-1),
                                    kernel_size=cnn_out.size(-2)
                                    ).squeeze(-1)

# |cnn_out| = (batch_size, n_filter)
cnn_outs += [cnn_out]

# Merge output tensors from each convolution layer.
cnn_outs = torch.cat(cnn_outs, dim=-1)
# |cnn_outs| = (batch_size, sum(n_filters))
y = self.activation(self.generator(cnn_outs))
# |y| = (batch_size, n_classes)

return y

```

<https://arxiv.org/pdf/1510.03820.pdf> # RNN Based Method

그럼 이제 딥러닝을 통한 텍스트 분류 문제를 살펴 보겠습니다. 딥러닝을 통해 텍스트 분류를 하기 위한 가장 간단한 방법은 Recurrent Neural Network(RNN)를 활용하는것 입니다. 문장은 단어들의 시퀀스로 이루어진 시계열(time-series) 데이터입니다. 따라서, 각 위치(또는 time-step)의 단어들은 다른 위치의 단어들과 영향을 주고 받습니다. RNN은 이런 문장의 특성을 가장 잘 활용할 수 있는 neural network 아키텍쳐입니다.

이전 챕터에서 다루었듯이, RNN은 각 time-step의 단어를 입력으로 받아 hidden state를 업데이트 합니다. 우리는 이때, 가장 마지막 hidden state를 활용하여 텍스트의 클래스를 분류할 수 있습니다. 따라서, 마치 RNN은 입력으로 주어진 문장을 분류 문제에 맞게 인코딩 한다고 볼 수 있습니다. 즉, RNN의 출력값은 문장 임베딩 벡터(sentence embedding vector)라고 볼 수 있습니다.

우리가 텍스트 분류를 RNN을 통해 구현한다면 위와 같은 구조가 될 것 입니다. One-hot 벡터로 주어진

## Implementation

```

import torch.nn as nn

class RNNClassifier(nn.Module):

    def __init__(self,

```

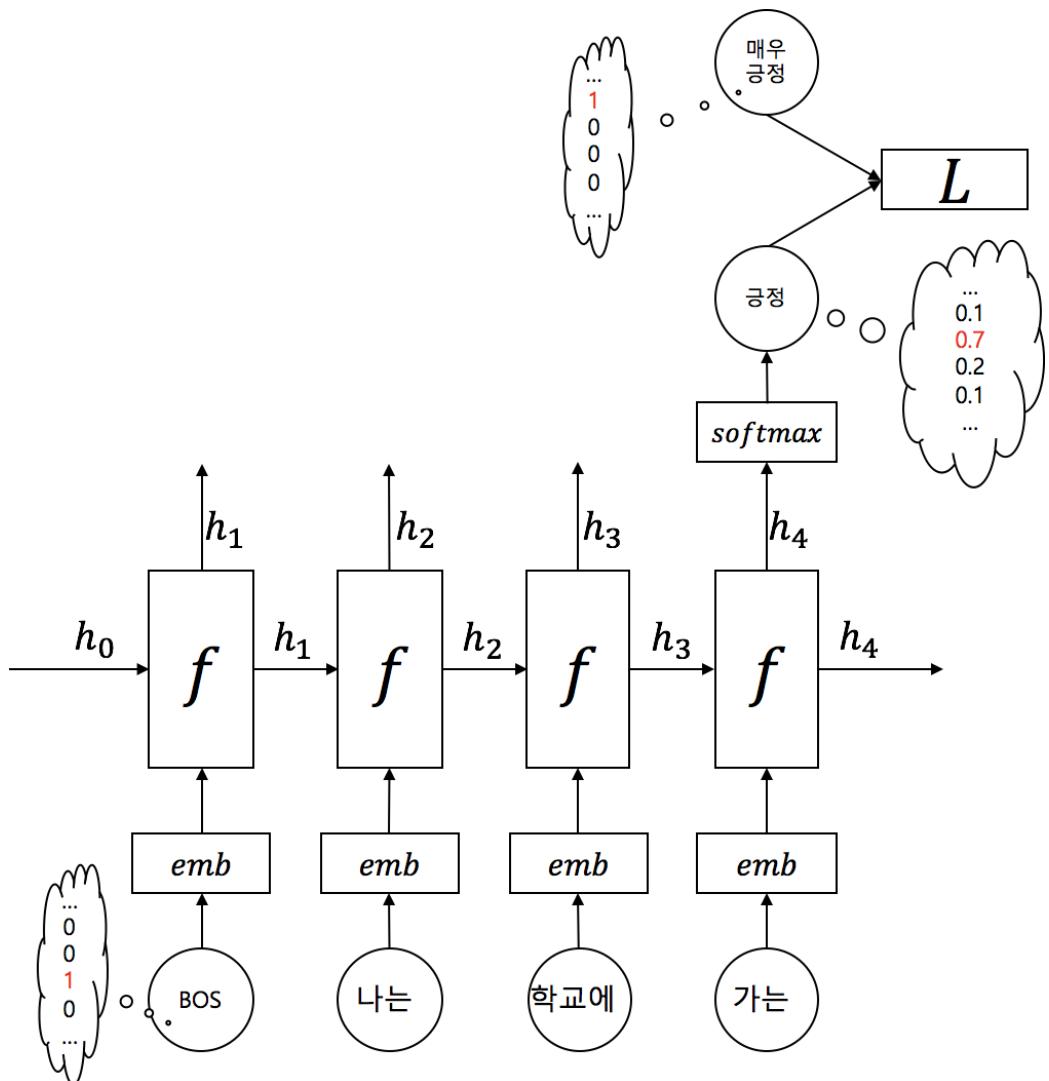


Figure 9: RNN의 마지막 time-step의 출력을 사용 하는 경우

```

        input_size,
        word_vec_dim,
        hidden_size,
        n_classes,
        n_layers=4,
        dropout_p=.3
    ):

    self.input_size = input_size # vocabulary_size
    self.word_vec_dim = word_vec_dim
    self.hidden_size = hidden_size
    self.n_classes = n_classes
    self.n_layers = n_layers
    self.dropout_p = dropout_p

super().__init__()

self.emb = nn.Embedding(input_size, word_vec_dim)
self.rnn = nn.LSTM(input_size=word_vec_dim,
                    hidden_size=hidden_size,
                    num_layers=n_layers,
                    dropout=dropout_p,
                    batch_first=True,
                    bidirectional=True
)
self.generator = nn.Linear(hidden_size * 2, n_classes)
# We use LogSoftmax + NLLLoss instead of Softmax + CrossEntropy
self.activation = nn.LogSoftmax(dim=-1)

def forward(self, x):
    # |x| = (batch_size, length)
    x = self.emb(x)
    # |x| = (batch_size, length, word_vec_dim)
    x, _ = self.rnn(x)
    # |x| = (batch_size, length, hidden_size * 2)
    y = self.activation(self.generator(x[:, -1]))
    # |y| = (batch_size, n_classes)

return y

```

## Implementation

아래는 텍스트 분류를 위한 학습 및 추론 프로그램 전체 소스코드입니다. 최신 소스코드는 저자의 깃허브에서 다운로드 받을 수 있습니다.

- Github Repo: <https://github.com/kh-kim/simple-ntc>

### Code

train.py

```
import argparse

import torch
import torch.nn as nn

from data_loader import DataLoader

from simple_ntc.rnn import RNNClassifier
from simple_ntc.cnn import CNNClassifier
from simple_ntc.trainer import Trainer


def define_argparser():
    """
    Define argument parser to handle parameters.
    """
    p = argparse.ArgumentParser()

    p.add_argument('--model', required=True)
    p.add_argument('--train', required=True)
    p.add_argument('--valid', required=True)
    p.add_argument('--gpu_id', type=int, default=-1)
    p.add_argument('--verbose', type=int, default=2)
    p.add_argument('--min_vocab_freq', type=int, default=2)
    p.add_argument('--max_vocab_size', type=int, default=999999)
```

```

p.add_argument('--batch_size', type=int, default=64)
p.add_argument('--n_epochs', type=int, default=10)
p.add_argument('--early_stop', type=int, default=-1)

p.add_argument('--dropout', type=float, default=.3)
p.add_argument('--word_vec_dim', type=int, default=128)
p.add_argument('--hidden_size', type=int, default=256)

p.add_argument('--rnn', action='store_true')
p.add_argument('--n_layers', type=int, default=4)

p.add_argument('--cnn', action='store_true')
p.add_argument('--window_sizes', type=str, default='3,4,5')
p.add_argument('--n_filters', type=str, default='100,100,100')

config = p.parse_args()

config.window_sizes = list(map(int, config.window_sizes.split(',')))
config.n_filters = list(map(int, config.n_filters.split(',')))

return config

def main(config):
    """
    The main method of the program to train text classification.
    :param config: configuration from argument parser.
    """

    dataset = DataLoader(train_fn=config.train,
                         valid_fn=config.valid,
                         batch_size=config.batch_size,
                         min_freq=config.min_vocab_freq,
                         max_vocab=config.max_vocab_size,
                         device=config.gpu_id
                         )

    vocab_size = len(dataset.text.vocab)
    n_classes = len(dataset.label.vocab)

```

```

print('|vocab| =', vocab_size, '|classes| =', n_classes)

if config.rnn is False and config.cnn is False:
    raise Exception('You need to specify an architecture to train. (--rnn or --cnn)')

if config.rnn:
    # Declare model and loss.
    model = RNNClassifier(input_size=vocab_size,
                           word_vec_dim=config.word_vec_dim,
                           hidden_size=config.hidden_size,
                           n_classes=n_classes,
                           n_layers=config.n_layers,
                           dropout_p=config.dropout
                           )
    crit = nn.NLLLoss()
    print(model)

    if config.gpu_id >= 0:
        model.cuda(config.gpu_id)
        crit.cuda(config.gpu_id)

    # Train until converge
    rnn_trainer = Trainer(model, crit)
    rnn_trainer.train(dataset.train_iter,
                      dataset.valid_iter,
                      batch_size=config.batch_size,
                      n_epochs=config.n_epochs,
                      early_stop=config.early_stop,
                      verbose=config.verbose
                      )

if config.cnn:
    # Declare model and loss.
    model = CNNClassifier(input_size=vocab_size,
                           word_vec_dim=config.word_vec_dim,
                           n_classes=n_classes,
                           dropout_p=config.dropout,
                           window_sizes=config.window_sizes,
                           n_filters=config.n_filters
                           )

```

```

        )
crit = nn.NLLLoss()
print(model)

if config.gpu_id >= 0:
    model.cuda(config.gpu_id)
    crit.cuda(config.gpu_id)

# Train until converge
cnn_trainer = Trainer(model, crit)
cnn_trainer.train(dataset.train_iter,
                   dataset.valid_iter,
                   batch_size=config.batch_size,
                   n_epochs=config.n_epochs,
                   early_stop=config.early_stop,
                   verbose=config.verbose
                  )

torch.save({'rnn': rnn_trainer.best if config.rnn else None,
            'cnn': cnn_trainer.best if config.cnn else None,
            'config': config,
            'vocab': dataset.text.vocab,
            'classes': dataset.label.vocab
           }, config.model)

if __name__ == '__main__':
    config = define_argparser()
    main(config)

```

data\_loader.py

```
from torchtext import data
```

```
class DataLoader(object):
    '''
```

```

Data loader class to load text file using torchtext library.
'''

def __init__(self, train_fn, valid_fn,
             batch_size=64,
             device=-1,
             max_vocab=999999,
             min_freq=1,
             use_eos=False,
             shuffle=True
             ):
    '''
    DataLoader initialization.
    :param train_fn: Train-set filename
    :param valid_fn: Valid-set filename
    :param batch_size: Batchify data for certain batch size.
    :param device: Device-id to load data (-1 for CPU)
    :param max_vocab: Maximum vocabulary size
    :param min_freq: Minimum frequency for loaded word.
    :param use_eos: If it is True, put <EOS> after every end of sentence.
    :param shuffle: If it is True, random shuffle the input data.
    '''
    super(DataLoader, self).__init__()

    # Define field of the input file.
    # The input file consists of two fields.
    self.label = data.Field(sequential=False,
                           use_vocab=True,
                           unk_token=None
                           )
    self.text = data.Field(use_vocab=True,
                          batch_first=True,
                          include_lengths=False,
                          eos_token='<EOS>' if use_eos else None
                          )

    # Those defined two columns will be delimited by TAB.
    # Thus, we use TabularDataset to load two columns in the input file.

```

```

# We would have two separate input file: train_fn, valid_fn
# Files consist of two columns: label field and text field.
train, valid = data.TabularDataset.splits(path='',
                                         train=train_fn,
                                         validation=valid_fn,
                                         format='tsv',
                                         fields=[('label', self.label),
                                                 ('text', self.text)
                                             ]
                                         )

# Those loaded dataset would be feeded into each iterator:
# train iterator and valid iterator.
# We sort input sentences by length, to group similar lengths.
self.train_iter, self.valid_iter = data.BucketIterator.splits((train, valid),
                                                               batch_size=batch_size,
                                                               device=device,
                                                               shuffle=shuffle,
                                                               sort_key=sort_key,
                                                               sort_within_batch=sort_within_batch)

```

*# At last, we make a vocabulary for label and text field.*

*# It is making mapping table between words and indice.*

```

self.label.build_vocab(train)
self.text.build_vocab(train, max_size=max_vocab, min_freq=min_freq)

```

trainer.py

```

from tqdm import tqdm
import torch

import utils

VERBOSE_SILENT = 0
VERBOSE_EPOCH_WISE = 1
VERBOSE_BATCH_WISE = 2

```

```

class Trainer():

    def __init__(self, model, crit):
        self.model = model
        self.crit = crit

        super().__init__()

        self.best = {}

    def get_best_model(self):
        self.model.load_state_dict(self.best['model'])

        return self.model

    def get_loss(self, y_hat, y, crit=None):
        crit = self.crit if crit is None else crit
        loss = crit(y_hat, y)

        return loss

    def train_epoch(self,
                    train,
                    optimizer,
                    batch_size=64,
                    verbose=VERBOSE_SILENT
                    ):
        """
        Train an epoch with given train iterator and optimizer.
        """

        total_loss, total_param_norm, total_grad_norm = 0, 0, 0
        avg_loss, avg_param_norm, avg_grad_norm = 0, 0, 0
        sample_cnt = 0

        progress_bar = tqdm(train,
                            desc='Training: ',

```

```

                unit='batch'
            ) if verbose is VERBOSE_BATCH_WISE else train
# Iterate whole train-set.
for idx, mini_batch in enumerate(progress_bar):
    x, y = mini_batch.text, mini_batch.label
    # Don't forget make grad zero before another back-prop.
    optimizer.zero_grad()

    y_hat = self.model(x)

    loss = self.get_loss(y_hat, y)
    loss.backward()

    total_loss += loss
    total_param_norm += utils.get_parameter_norm(self.model.parameters())
    total_grad_norm += utils.get_grad_norm(self.model.parameters())

    # Calculation to show status
    avg_loss = total_loss / (idx + 1)
    avg_param_norm = total_param_norm / (idx + 1)
    avg_grad_norm = total_grad_norm / (idx + 1)

    if verbose is VERBOSE_BATCH_WISE:
        progress_bar.set_postfix_str('|param|=%.2f |g_param|=%.2f loss=%.2f' % (avg_param_norm, avg_grad_norm, avg_loss))

optimizer.step()

sample_cnt += mini_batch.text.size(0)
if sample_cnt >= len(train.dataset.examples):
    break

if verbose is VERBOSE_BATCH_WISE:
    progress_bar.close()

return avg_loss, avg_param_norm, avg_grad_norm

```

```

def train(self,
          train,
          valid,
          batch_size=64,
          n_epochs=100,
          early_stop=-1,
          verbose=VERBOSE_SILENT
        ):
    """
    Train with given train and valid iterator until n_epochs.
    If early_stop is set,
    early stopping will be executed if the requirement is satisfied.
    """
    optimizer = torch.optim.Adam(self.model.parameters())

    lowest_loss = float('Inf')
    lowest_after = 0

    progress_bar = tqdm(range(n_epochs),
                         desc='Training: ',
                         unit='epoch'
                         ) if verbose is VERBOSE_EPOCH_WISE else range(n_epochs)
    for idx in progress_bar: # Iterate from 1 to n_epochs
        if verbose > VERBOSE_EPOCH_WISE:
            print('epoch: %d/%d\tmin_valid_loss=%.4e' % (idx + 1,
                                                          len(progress_bar),
                                                          lowest_loss
                                                          ))
        avg_train_loss, avg_param_norm, avg_grad_norm = self.train_epoch(train,
                                                               optimizer,
                                                               batch_size,
                                                               verbose
                                                               )
        _, avg_valid_loss = self.validate(valid,
                                           verbose=verbose
                                           )

```

```

# Print train status with different verbosity.
if verbose is VERBOSE_EPOCH_WISE:
    progress_bar.set_postfix_str('|param|=%.2f |g_param|=%.2f train_lo

if avg_valid_loss < lowest_loss:
    # Update if there is an improvement.
    lowest_loss = avg_valid_loss
    lowest_after = 0

    self.best = {'model': self.model.state_dict(),
                 'optim': optimizer,
                 'epoch': idx,
                 'lowest_loss': lowest_loss
                }
else:
    lowest_after += 1

    if lowest_after >= early_stop and early_stop > 0:
        break
if verbose is VERBOSE_EPOCH_WISE:
    progress_bar.close()

def validate(self,
            valid,
            crit=None,
            batch_size=256,
            verbose=VERBOSE_SILENT
            ):
    """
    Validate a model with given valid iterator.
    """
    # We don't need to back-prop for these operations.
    with torch.no_grad():

```

```

total_loss, total_correct, sample_cnt = 0, 0, 0
progress_bar = tqdm(valid,
                     desc='Validation: ',
                     unit='batch'
                     ) if verbose is VERBOSE_BATCH_WISE else valid

y_hats = []
self.model.eval()
# Iterate for whole valid-set.
for idx, mini_batch in enumerate(progress_bar):
    x, y = mini_batch.text, mini_batch.label
    y_hat = self.model(x)
    # |y_hat| = (batch_size, n_classes)

    loss = self.get_loss(y_hat, y, crit)

    total_loss += loss
    sample_cnt += mini_batch.text.size(0)
    total_correct += float(y_hat.topk(1)[1].view(-1).eq(y).sum())

    avg_loss = total_loss / (idx + 1)
    y_hats += [y_hat]

    if verbose is VERBOSE_BATCH_WISE:
        progress_bar.set_postfix_str('valid_loss=%.4e accuarcy=%.4f' %

            if sample_cnt >= len(valid.dataset.examples):
                break
self.model.train()

if verbose is VERBOSE_BATCH_WISE:
    progress_bar.close()

y_hats = torch.cat(y_hats, dim=0)

return y_hats, avg_loss

```

rnn.py

```
import torch.nn as nn

class RNNClassifier(nn.Module):

    def __init__(self,
                 input_size,
                 word_vec_dim,
                 hidden_size,
                 n_classes,
                 n_layers=4,
                 dropout_p=.3
                 ):
        self.input_size = input_size # vocabulary_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.n_classes = n_classes
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        super().__init__()

        self.emb = nn.Embedding(input_size, word_vec_dim)
        self.rnn = nn.LSTM(input_size=word_vec_dim,
                           hidden_size=hidden_size,
                           num_layers=n_layers,
                           dropout=dropout_p,
                           batch_first=True,
                           bidirectional=True
                           )
        self.generator = nn.Linear(hidden_size * 2, n_classes)
        # We use LogSoftmax + NLLLoss instead of Softmax + CrossEntropy
        self.activation = nn.LogSoftmax(dim=-1)

    def forward(self, x):
```

```

# |x| = (batch_size, length)
x = self.emb(x)
# |x| = (batch_size, length, word_vec_dim)
x, _ = self.rnn(x)
# |x| = (batch_size, length, hidden_size * 2)
y = self.activation(self.generator(x[:, -1]))
# |y| = (batch_size, n_classes)

return y

```

cnn.py

```

import torch
import torch.nn as nn

class CNNClassifier(nn.Module):

    def __init__(self,
                 input_size,
                 word_vec_dim,
                 n_classes,
                 dropout_p=.5,
                 window_sizes=[3, 4, 5],
                 n_filters=[100, 100, 100]):
        super().__init__()

        self.input_size = input_size # vocabulary size
        self.word_vec_dim = word_vec_dim
        self.n_classes = n_classes
        self.dropout_p = dropout_p
        # window_size means that how many words a pattern covers.
        self.window_sizes = window_sizes
        # n_filters means that how many patterns to cover.
        self.n_filters = n_filters

        super().__init__()

```

```

self.emb = nn.Embedding(input_size, word_vec_dim)
# Since number of convolution layers would be vary depend on len(window_
# we use 'setattr' and 'getattr' methods to add layers to nn.Module obj
for window_size, n_filter in zip(window_sizes, n_filters):
    cnn = nn.Conv2d(in_channels=1,
                   out_channels=n_filter,
                   kernel_size=(window_size, word_vec_dim)
                   )
    setattr(self, 'cnn-%d-%d' % (window_size, n_filter), cnn)
# Because below layers are just operations,
# (it does not have learnable parameters)
# we just declare once.
self.relu = nn.ReLU()
self.dropout = nn.Dropout(dropout_p)
# An input of generator layer is max values from each filter.
self.generator = nn.Linear(sum(n_filters), n_classes)
# We use LogSoftmax + NLLLoss instead of Softmax + CrossEntropy
self.activation = nn.LogSoftmax(dim=-1)

def forward(self, x):
    # |x| = (batch_size, length)
    x = self.emb(x)
    # |x| = (batch_size, length, word_vec_dim)
    min_length = max(self.window_sizes)
    if min_length > x.size(1):
        # Because some input does not long enough for maximum length of wind
        # we add zero tensor for padding.
        pad = x.new(x.size(0), min_length - x.size(1), self.word_vec_dim).zero_()
        # |pad| = (batch_size, min_length - length, word_vec_dim)
        x = torch.cat([x, pad], dim=1)
        # |x| = (batch_size, min_length, word_vec_dim)

    # In ordinary case of vision task, you may have 3 channels on tensor,
    # but in this case, you would have just 1 channel,
    # which is added by 'unsqueeze' method in below:
    x = x.unsqueeze(1)
    # |x| = (batch_size, 1, length, word_vec_dim)

```

```

cnn_outs = []
for window_size, n_filter in zip(self.window_sizes, self.n_filters):
    cnn = getattr(self, 'cnn-%d-%d' % (window_size, n_filter))
    cnn_out = self.dropout(self.relu(cnn(x)))
    # |x| = (batch_size, n_filter, length - window_size + 1, 1)

    # In case of max pooling, we does not know the pooling size,
    # because it depends on the length of the sentence.
    # Therefore, we use instant function using 'nn.functional' package.
    # This is the beauty of PyTorch. :)
    cnn_out = nn.functional.max_pool1d(input=cnn_out.squeeze(-1),
                                       kernel_size=cnn_out.size(-2)
                                       ).squeeze(-1)
    # |cnn_out| = (batch_size, n_filter)
    cnn_outs += [cnn_out]
# Merge output tensors from each convolution layer.
cnn_outs = torch.cat(cnn_outs, dim=-1)
# |cnn_outs| = (batch_size, sum(n_filters))
y = self.activation(self.generator(cnn_outs))
# |y| = (batch_size, n_classes)

return y

```

utils.py

```

def get_grad_norm(parameters, norm_type=2):
    parameters = list(filter(lambda p: p.grad is not None, parameters))

    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.grad.data.norm(norm_type)
            total_norm += param_norm ** norm_type
    total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

```

```

    return total_norm

def get_parameter_norm(parameters, norm_type=2):
    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.data.norm(norm_type)
            total_norm += param_norm ** norm_type
    total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm

```

classify.py

```

import sys
import argparse

import torch
import torch.nn as nn
from torchtext import data

from simple_ntc.rnn import RNNClassifier
from simple_ntc.cnn import CNNClassifier

def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('--model', required=True)
    p.add_argument('--gpu_id', type=int, default=-1)
    p.add_argument('--batch_size', type=int, default=256)
    p.add_argument('--top_k', type=int, default=1)

```

```

config = p.parse_args()

return config

def read_text():
    # This method gets sentences from standard input and tokenize those.
    lines = []

    for line in sys.stdin:
        if line.strip() != '':
            lines += [line.strip().split(' ')]

    return lines

def define_field():
    return data.Field(use_vocab=True,
                      batch_first=True,
                      include_lengths=False
                      ), data.Field(sequential=False, use_vocab=True, unk_token=None)

def main(config):
    saved_data = torch.load(config.model)

    train_config = saved_data['config']

    rnn_best = saved_data['rnn']
    cnn_best = saved_data['cnn']
    vocab = saved_data['vocab']
    classes = saved_data['classes']

    vocab_size = len(vocab)
    n_classes = len(classes)

    text_field, label_field = define_field()
    text_field.vocab = vocab

```

```

label_field.vocab = classes

lines = read_text()

with torch.no_grad():
    # Converts string to list of index.
    x = text_field.numericalize(text_field.pad(lines),
                                device='cuda:%d' % config.gpu_id if config.gpu_id != -1
                                )

ensemble = []
if rnn_best is not None:
    model = RNNClassifier(input_size=vocab_size,
                          word_vec_dim=train_config.word_vec_dim,
                          hidden_size=train_config.hidden_size,
                          n_classes=n_classes,
                          n_layers=train_config.n_layers,
                          dropout_p=train_config.dropout
                          )
    model.load_state_dict(rnn_best['model'])
    ensemble += [model]
if cnn_best is not None:
    model = CNNClassifier(input_size=vocab_size,
                          word_vec_dim=train_config.word_vec_dim,
                          n_classes=n_classes,
                          dropout_p=train_config.dropout,
                          window_sizes=train_config.window_sizes,
                          n_filters=train_config.n_filters
                          )
    model.load_state_dict(cnn_best['model'])
    ensemble += [model]

y_hats = []
for model in ensemble:
    if config.gpu_id >= 0:
        model.cuda(config.gpu_id)
    model.eval()

```

```

y_hat = []
for idx in range(0, len(lines), config.batch_size):
    y_hat += [model(x[idx:idx + config.batch_size])]
y_hat = torch.cat(y_hat, dim=0)
# |y_hat| = (len(lines), n_classes)

y_hats += [y_hat]
y_hats = torch.stack(y_hats).exp()
# |y_hats| = (len(ensemble), len(lines), n_classes)
y_hats = y_hats.sum(dim=0) / len(ensemble)
# |y_hats| = (len(lines), n_classes)

probs, indice = y_hats.cpu().topk(config.top_k)

for i in range(len(lines)):
    sys.stdout.write('%s\t%s\n' % (' '.join([classes.itos[indice[i][j]] for j in range(config.top_k)])))

if __name__ == '__main__':
    config = define_argparser()
    main(config)

```

## Language Modeling



Figure 1: Daniel Jurafsky – Image from web

## Language Model

### Introduction

이전 챕터까지 우리는 단어 또는 문장을 입력으로 받아서, 어떤 값으로 맵핑(mapping)해주는 방법에 대해서 다루었습니다. 그 값을 통해 해당 단어 또는 문장을 분류하기도 하고 군집을 형성 할 수도 있었습니다. 이러한 방법들도 매우 쓰임새가 많고, 정말 중요합니다. 하지만, 우리는 여기서 더 나아가 컴퓨터로 하여금 필요에 따라 자연스러운 문장을 만들어내도록 하는 방법에 대해 다루어 보도록 하겠습니다.

언어 모델(Language Model)은 문장의 확률을 나타내는 모델입니다. 즉, 우리는 언어 모델을 통해 문장 자체의 출현 확률을 예측하거나, 이전 단어들이 주어졌을 때 다음 단어를 예측할 수 있으며, 결과적으로 우리는 주어진 문장이 얼마나 자연스러운 유창한(fluent) 표현인지 계산 할 수 있게 됩니다. 예를 들어 우리는 아래와 같은 문장이 주어졌을 때, 빈칸을 어렵지 않게 메꿀 수 있습니다.

- 버스 정류장에서 방금 버스를 ○ ○ ○.

1. 사랑해
2. 고양이
3. 놓쳤다
4. 사고남

우리는 정답이 3번 ‘놓쳤다’라고 쉽게 맞출 수 있습니다. 4번 ’사고남’의 경우에는 앞 단어가 ’버스를’ 대신에 ’버스가’ 였다면 정답이 될 수도 있었겠지요. 4번을 정답이라고 할 경우에는 뜻은 이해할 수 있지만 뭔가 어색함이 느껴지는 문장이 될 겁니다.

번호	문장
1	저는 어제 점심을 먹었습니다.
2	저는 2018년 4월 26일 점심을 먹었습니다.

위의 두 문장 중 1번 문장을 접할 기회는, 2번 문장을 접할 기회보다 훨씬 많을 겁니다. 수많은 단어들이 있고, 그 단어들간의 조합은 더욱 많이 존재 합니다. 그러한 조합들은 동등한 확률을 갖기보다는 자주 나타나는 단어나 표현(단어의 조합)이 훨씬 높은 확률로 나타날 겁니다.

이렇게 우리는 살아오면서 수많은 문장을 접하였고, 우리의 머릿속에는 단어와 단어 사이의 확률이 우리도 모르게 학습되어 있습니다. 덕분에 누군가와 대화를 하다가 한 단어 정도는 정확하게 알아듣지 못하여도, 애매한 경우를 제외하고는, 대화에 지장이 없을 겁니다. (물론 여기에는 문맥 정보를 이용 하는 것도 큰 도움이 됩니다.) 이와 같이 꼭 자연어처리 분야가 아니더라도, 음성인식(Speech Recognition)이나 문자인식(Optical Character Recognition, OCR)에 있어서도 언어모델은 큰 역할을 수행합니다. 우리는 인터넷이나 도서에서 많은 문장들을 (대개 수십만에서 수억 까지) 수집하여 단어와 단어 사이의 출현 빈도를 세어 확률을 계산하고, 언어모델을 구성합니다. 궁극적인 목표는 우리가 일상 생활에서 사용하는 문장의 분포를 정확하게 파악하는데에 있습니다. 또한, 만약 특정한 목표를 가지고 있다면 (예를 들어 의료 분야의 음성인식기 제작) 해당 분야(domain)의 문장의 분포를 파악하기 위해서 해당 분야의 말뭉치(corpus)를 수집하기도 합니다.

## Again, Korean is Hell

우리는 언어들의 구조적 특성에 따라서 여러 갈래로 언어를 분류합니다. 이전(Why Korean NLP is Hell)에 다루었듯이 한국어는 대표적인 교착어입니다. 그리고 영어는 고립어(+굴절어)의 특성을 띠고, 중국어는 고립어로 분류합니다. 교착어의 특성상, 단어의 의미 또는 역할은 어순에 의해 결정되기 보단, 단어에 부착되는 어미와 같은 접사에 의해 그 역할이 결정됩니다. 따라서 같은 의미의 단어라 할지라도 붙는 접사에 따라 단어의 형태가 달라지게 되어 단어의 수가 늘어나게 됩니다. (버스가, 버스를, 버스에, 버스로 등 같은 버스라도 뒤에 붙는 어미에 따라 다른 단어가 됩니다.) 다시 말해, 단어의 어순이 중요하지 않기 때문에 (또는 생략 가능하기 때문에), 단어와 단어 사이의 확률을 계산하는데 불리하게 작용할 수 있습니다.

번호	문장
1	나는 학교에 갑니다 버스를 타고 .
2	나는 버스를 타고 학교에 갑니다 .
3	버스를 타고 나는 학교에 갑니다 .
4	(나는) 버스를 타고 학교에 갑니다 .

위의 세문장 모두 같은 의미의 표현이고 사용 된 단어들도 같지만, 어순이 다르기 때문에, 단어와 단어 사이의 확률을 정의하는데 있어서 혼란이 가중됩니다. 같은 의미의 문장을 표현하기 위해서 ‘타고’ 다음에 나타날 수 있는 단어들은 ‘.’, ‘학교에’, ‘나는’ 3개이기 때문에, 확률이 쉽게 말해 퍼지는 현상이 생기게 됩니다. 이에 반해 영어나 기타 라틴어 기반 언어들은 좀 더 어순에 있어서 규칙적이므로 좀 더 유리한 면이 있습니다.

더군다나, 한국어는 교착어의 특성상 어미가 부착되어 단어를 형성하기 때문에 같은 ‘학교’라는 단어가 ‘학교+에’, ‘학교+로’, ‘학교+를’, ‘학교+가’, … 등 수많은 단어로 파생되어 만들어질 수 있습니다. 따라서 사실 어미를 따로 분리해주지 않으면 어휘의 수가 기하급수적으로 늘어나게 되어 sparse함이 더욱 높아질 수 있어 더욱 문제 해결이 어려워질 수 있습니다.

## Probability of Sentence

먼저 문장의 확률을 표현 해 보도록 하겠습니다.  $w_1, w_2$ 라는 2개의 단어가 한 문장 안에 순서대로 나타났을 때, 이 문장의 확률로 표현하면 다음과 같습니다.

$$P(w_1, w_2)$$

우리는 이 수식을 베이즈 정리(Bayes Theorem)에 따라 조건부 확률(Conditional Probability)로 표현할 수 있습니다.

$$\begin{aligned} P(w_1, w_2) &= P(w_1)P(w_2|w_1) \\ &\text{because} \\ P(w_2|w_1) &= \frac{P(w_1, w_2)}{P(w_1)}. \end{aligned}$$

좀 더 나아가서 Chain Rule을 통해,  $W = \{w_1, w_2, w_3, w_4\}$ , 4개의 단어가 한 문장 안에 있을 때를 표현 해 보면 아래와 같습니다.

$$P(W) = P(w_1, w_2, w_3, w_4) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_4|w_1, w_2, w_3)$$

여기서 Chain Rule(연쇄 법칙)이란, 조건부 확률(conditional probability)을 사용하여 결합 확률(joint probability)를 계산 하는 방법으로, 아래와 같이 유도 할 수 있습니다.

$$\begin{aligned} P(A, B, C, D) &= P(D|A, B, C)P(A, B, C) \\ &= P(D|A, B, C)P(C|A, B)P(A, B) \\ &= P(D|A, B, C)P(C|A, B)P(B|A)P(A) \end{aligned}$$

우향을 해석해보면,  $w_1$ 가 나타날 확률과  $w_1$ 가 주어졌을 때  $w_2$ 가 나타날 확률,  $w_1, w_2$ 가 주어졌을 때  $w_3$ 가 주어졌을 확률,  $w_1, w_2, w_3$ 가 주어졌을 때  $w_4$ 가 나타날 확률을 곱하는 것을 알 수 있습니다. 이로써 우리는 language model을 통해서 문장에 대한 확률 뿐만 아니라, 단어와 단어 사이의 확률도 정의 할 수 있습니다. 우리는 이를 일반화하여 아래와 같이 표현할 수 있습니다.

$$P(W) = \prod_{i=1}^n P(w_i|w_{<i})$$

또는 log 확률로 표현하여 곱셈 대신 덧셈으로 표현할 수 있습니다. 참고로, 문장이 길어지게 된다면 당연히 확률에 대한 곱셈이 거듭되면서 확률이 매우 작아지게 되어

정확한 계산 또는 표현이 힘들어지게 됩니다. (또한, 곱셈 연산보다 덧셈 연산이 더 빠릅니다.) 따라서 우리는 log를 취하여 덧셈으로 바꾸어 더 나은 조건을 취할 수 있습니다.

$$\log P(W) = \sum_{i=1}^n \log P(w_i | w_{<i})$$

이제 우리는 실제 예제를 가지고 표현 해 보도록 하겠습니다. Corpus  $\mathcal{C} = \{W_1, W_2, \dots\}$ 에서  $i$ 번째 문장  $W_i = \{BOS, 나는, 학교에, 갑니다, EOS\}$ 에 대한 확률을 Chain rule을 통해 표현하면 아래와 같습니다.

$$P(BOS, 나는, 학교에, 갑니다, EOS) = P(BOS)P(\text{나는}|BOS)P(\text{학교에}|BOS, 나는)P(\text{갑니다}|BOS, 나는)$$

여기서 BOS는 Beginning of Sentence라는 의미의 토큰이고, EOS는 End of sentence라는 의미의 토큰입니다.  $P(BOS)$ 의 경우에는 항상 문장의 시작에 오게 되므로 상수가 될 것 입니다.  $P(\text{나는}|BOS)$  경우에는 문장의 첫 단어로 나는이 올 확률을 나타내게 됩니다.

이제 문장을 어떻게 확률로 나타내는지 알았으니, 확률을 직접 구하는 방법에 대해서 알아보겠습니다. 우리는 앞 챕터에서 문장을 수집하는 방법에 대해서 논의 했습니다. 수집한 말뭉치 내에서 직접 단어들을 카운트 함으로써 우리가 원하는 확률을 구할 수 있습니다. 예를 들어 아래의 확률은 다음과 같이 구할 수 있습니다.

$$P(\text{갑니다}|BOS, 나는, 학교에) = \frac{\text{Count}(BOS, 나는, 학교에, 갑니다)}{\text{Count}(BOS, 나는, 학교에)}$$

## N-gram

### Sparseness

이전 섹션에서 언어모델에 대해 소개를 간략히 했습니다. 하지만 만약 그 수식대로라면 우리는 확률들을 거의 구할 수 없을 것 입니다. 왜냐하면 비록 우리가 수천만~수억 문장을 인터넷에서 긁어 모았다고 하더라도, 애초에 출현할 수 있는 단어의 조합의 경우의 수는 무한대에 가깝기 때문입니다. 문장이 조금만 길어지더라도 Count를 구할 수 없어 문자가 0이 되어 확률이 0이 되거나, 심지어

분모가 0이 되어 정의할 수가 없어져버릴 것이기 때문입니다. 이렇게 너무나도 많은 경우의 수 때문에 생기는 문제를 희소성(sparseness or sparsity) 문제라고 표현합니다.

## Markov Assumption

따라서 희소성(sparseness) 문제를 해결하기 위해서 Markov Assumption을 도입합니다.

$$P(x_i|x_1, x_2, \dots, x_{i-1}) \approx P(x_i|x_{i-k}, \dots, x_{i-1})$$

Markov Assumption을 통해, 다음 단어의 출현 확률을 구하기 위해서, 이전에 출현한 모든 단어를 볼 필요 없이, 앞에  $k$  개의 단어만 상관하여 다음 단어의 출현 확률을 구하도록 하는 것입니다. 이렇게 가정을 간소화 하여, 우리가 구하고자 하는 확률을 근사(approximation) 하겠다는 것입니다. 보통  $k$ 는 0에서 3의 값을 갖게 됩니다. 즉,  $k = 2$  일 경우에는 앞 단어 2개를 참조하여 다음 단어( $x_i$ )의 확률을 근사하여 나타내게 됩니다.

$$P(x_i|x_{i-2}, x_{i-1})$$

이를 이전 chain rule(연쇄법칙) 수식에 적용하여, 문장에 대한 확률도 다음과 같이 표현 할 수 있습니다.

$$P(x_1, x_2, \dots, x_n) \approx \prod_{j=1}^n P(x_j|x_{j-k}, \dots, x_{j-1})$$

이것을 log 확률로 표현하면,

$$\log P(x_1, x_2, \dots, x_n) \approx \sum_{j=1}^n \log P(x_j|x_{j-k}, \dots, x_{j-1})$$

우리는 이렇게 전체 문장 대신에 바로 앞 몇 개의 단어만 상관하여 확률 계산을 간소화하는 방법을  $n = k + 1$ 으로 n-gram 이라고 부릅니다.

k	n-gram	명칭
0	1-gram	uni-gram

k	n-gram	명칭
1	2-gram	bi-gram
2	3-gram	tri-gram

위 테이블과 같이 3-gram 까지는 tri-gram이라고 읽지만 4-gram 부터는 그냥 four-gram 이라고 읽습니다. 앞서 설명 하였듯이,  $n$ 이 커질수록 우리가 가지고 있는 훈련 corpus내에 존재하지 않을 가능성성이 많기 때문에, 오히려 확률을 정확하게 계산하는데 어려움이 있을 수도 있습니다. (예를 들어 훈련 corpus에 존재 하지 않는다고 세상에서 쓰이지 않는 문장 표현은 아니기 때문입니다.) 따라서 당연히 훈련 corpus의 양이 적을수록  $n$ 의 크기도 줄어들어야 합니다. 보통은 대부분 어느정도 훈련 corpus가 적당히 있다는 가정 하에서, 3-gram을 가장 많이 사용하고, 훈련 corpus의 양이 많을 때는 4-gram을 사용하기도 합니다. 하지만 이렇게 4-gram을 사용하면 언어모델의 성능은 크게 오르지 않는데 반해, 단어 조합의 경우의 수는 지수적(exponential)으로 증가하기 때문에, 사실 크게 효율성이 없습니다.

$$P(x_i|x_{i-2}, x_{i-1}) = \frac{\text{Count}(x_{i-2}, x_{i-1}, x_i)}{\text{Count}(x_{i-2}, x_{i-1})}$$

이제 위와 같이 3개 단어의 출현 빈도와, 앞 2개 단어의 출현 빈도만 구하면  $x_i$ 의 확률을 근사할 수 있습니다. 즉, 아래와 같은 문장 전체의 확률에 대해서

$$P(x_1, x_2, \dots, x_n)$$

비록 훈련 corpus 내에 해당 문장이 존재 한 적이 없더라도, Markov assumption을 통해서 우리는 해당 문장의 확률을 근사(approximation)할 수 있게 되었습니다.

## Generalization

머신러닝의 힘은 보지 못한 case에 대한 대처 능력, 즉 일반화(generalization)에 있습니다. 따라서, n-gram도 Markov assumption을 통해서 희소성(sparseness or sparsity)를 해결하는 일반화 능력을 갖게 되었다고 할 수 있습니다. 이제 우리는 이것을 좀 더 향상시킬 수 있는 방법을 살펴 보도록 하겠습니다.

## Smoothing (Discounting)

출현 횟수를 단순히 확률 값으로 이용 할 경우 문제점이 무엇이 있을까요? 바로 훈련 corpus에 출현하지 않는 단어 조합에 대한 대처 방법입니다. 비록, markov assumption을 적용하여 우리는 희소성 문제를 훨씬 줄였고, 문장의 모든 단어 조합에 대해서 출현 횟수를 세지 않아도 되지만, 어찌되었든 그래도 훈련 corpus에 등장하지 않는 경우는 결국 존재 할 것인데, 그 경우인 훈련 셋에서 보지 못한 단어 조합(unseen word sequence)이라고 해서 등장 확률이 0이 되면 맞지 않습니다. 따라서 출현 횟수(counting) 값 또는 확률 값을 좀 더 다듬어 줘야 할 필요성이 있습니다. 아래 파란색 막대기와 같이 둘쭉날쭉한 출현 횟수 값을 주황색 선으로 부드럽게(smooth) 바꾸어 주기 때문에 smoothing 또는 discounting이라고 불립니다. 그럼 그 방법에 대해 살펴보겠습니다.

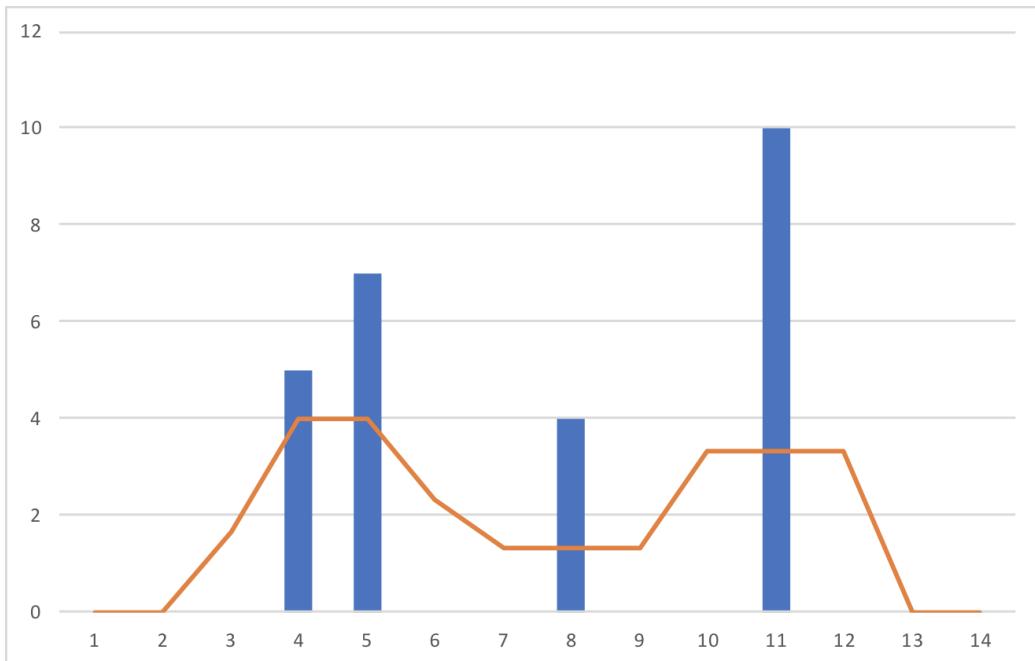


Figure 2:

### Add one Smoothing

먼저 우리가 생각 해 볼 수 있는 가장 간단한 방법은, 모든 n-gram에 1을 더하는 것입니다. 그렇다면 훈련셋에 출현하지 않은 n-gram의 경우에도 작은 확률이나마 가질

수 있을 것 입니다. 이를 수식으로 나타내면 아래와 같습니다.

$$P(w_i|w_{<i}) \approx \frac{\text{Count}(w_{<i}, w_i) + 1}{\text{Count}(w_{<i}) + V}$$

이처럼 1을 더하여 smoothing을 통해  $P(w_i|w_{<i})$ 를 근사할 수 있습니다. 이 수식을 좀 더 일반화하여 표현하면 아래와 같이 쓸 수 있습니다.

$$\begin{aligned} P(w_i|w_{<i}) &\approx \frac{\text{Count}(w_{<i}, w_i) + k}{\text{Count}(w_{<i}) + kV} \\ &\approx \frac{\text{Count}(w_{<i}, w_i) + (m/V)}{\text{Count}(w_{<i}) + m} \end{aligned}$$

이처럼, 1보다 작은 상수값을 더하여 smoothing을 구현 해 볼 수도 있을 것 입니다. 그렇다면 여기서 또 한발 더 나아가 1-gram prior 확률을 이용하여 좀 더 동적으로 대처 해 볼 수도 있을 것 입니다.

$$P(w_i|w_{<i}) \approx \frac{\text{Count}(w_{<i}, w_i) + mP(w_i)}{\text{Count}(w_{<i}) + m}$$

이처럼 add-one smoothing은 매우 간단하지만, 사실 언어모델처럼 희소성(sparseness) 문제가 큰 경우에는 부족합니다. 따라서 언어모델에 쓰이기에는 알맞은 방법은 아닙니다.

### Absolute Smoothing

[Church et al.1991]은 bigram에 대해서 실험을 한 결과를 제시하였습니다. 훈련용 corpus에서  $n$ 번 나타난 2-gram에 대해서, held-out corpus (validation or development set)에서 나타난 횟수를 세어 평균을 낸 것 입니다. 그 결과는 아래와 같습니다.

재미있게도, 0번과 1번 나타난 2-gram을 제외하면, 2번부터 9번 나타난 2-gram의 경우에는 held-out corpus에서의 출현 횟수는 훈련용 corpus 출현 횟수보다 약 0.75번 정도 적게 나타났다는 것 입니다. 즉, 출현 횟수(counting)에서 상수  $d$ 를 빼주는 것과 같은 것입니다.

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

**Figure 4.8** For all bigrams in 22 million words of AP newswire of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

Figure 3:

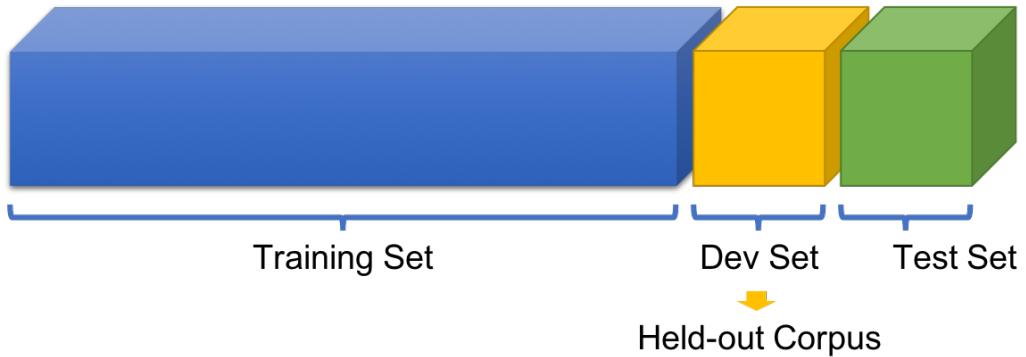


Figure 4:

## Kneser–Ney Smoothing

[Kneser et al.1995]은 여기에서 한발 더 나아가, KN discount를 제시하였습니다.

KN discounting의 주요 아이디어는 단어  $w$ 가 누군가( $v$ )의 뒤에서 출현 할 때, 얼마나 다양한 단어 뒤에서 출현하는지를 알아내는 것입니다. 그래서 다양한 단어 뒤에 나타나는 단어일수록, 훈련셋에서 보지 못한 단어 조합(unseen word sequence)으로써 나타날 확률이 높다는 것입니다.

- learning의 앞에 출현한 단어: machine, deep, supervised, unsupervised, generative, discriminative, ...
- laptop의 앞에 출현한 단어: slim, favorite, fancy, expensive, cheap, ...

예를 들어, 우리 책은 machine learning(기계학습)과 deep learning(딥러닝)을 다루는 책 이므로, 책 내에서 learning이라는 keyword의 빈도는 굉장히 높을 것입니다. 하지만, 해당 단어는 주로 machine과 deep 뒤에서만 나타났다고 해 보죠. learning이라는 단어에 비해서, laptop이라는 표현의 빈도는 낮을 것입니다. 하지만 learning과 같이 특정 단어의 뒤에서 대부분 나타나기 보단, 자유롭게 나타났을 것 같습니다. KN discounting은 이 경우, laptop이 unseen word sequence에서 나타날 확률이 더 높다고 가정하는 것입니다. 한마디로 낮을 덜 가리는 단어를 찾아내는 것입니다.

KN discounting은  $P_{\text{continuation}}$ 을 아래와 같이 모델링 합니다. 즉,  $w$ 와 같이 나타난  $v$ 들의 집합의 크기가 클 수록  $P_{\text{continuation}}$ 은 클 것이라고 가정 합니다.

$$P_{\text{continuation}}(w) \propto |\{v : \text{Count}(v, w) > 0\}|$$

위의 수식은 이렇게 나타내 볼 수 있습니다.  $w$ 와 같이 나타난  $v$ 들의 집합의 크기를,  $v, w'$ 가 함께 나타난 집합의 크기의 합으로 나누어 줍니다.

$$P_{\text{continuation}}(w) = \frac{|\{v : \text{Count}(v, w) > 0\}|}{\sum_{w'} |\{v : \text{Count}(v, w') > 0\}|}$$

이렇게 우리는  $P_{KN}$ 를 정의 할 수 있습니다.

$$P_{\text{KN}}(w_i | w_{i-1}) = \frac{\max(C(w_{i-1}, w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1}) P_{\text{continuation}}(w_i),$$

where  $\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}, v)} \times |\{w : c(w_{i-1}, w) > 0\}|$ .

## Interpolation

Interpolation에 의한 generalization(일반화)을 살펴 보도록 하겠습니다. Interpolation은 두 다른 언어모델을 선형적으로 일정 비율( $\lambda$ )로 섞어 주는 것입니다. Interpolation을 통해 얻을 수 있는 효과는, 일반 영역의 corpus를 통해 구축한 언어모델을, 필요에 따라 다른 특정 영역(domain)의 양이 적은 corpus를 통해 구축한 영역 특화(domain specific or adapted) 언어모델과 섞어 주는 것입니다. 이를 통해 일반적인 언어 모델을 해당 영역에 특화 된 언어 모델로 만들 수 있습니다.

$$\tilde{P}(w_n|w_{n-k}, \dots, w_{n-1}) = \lambda P_1(w_n|w_{n-k}, \dots, w_{n-1}) + (1-\lambda) P_2(w_n|w_{n-k}, \dots, w_{n-1})$$

예를 들어 의료 쪽 음성인식(ASR) 또는 기계번역(MT) 시스템을 구축한다고 가정 해보겠습니다. 그렇다면 기존의 일반 영역 corpus를 통해 생성한 language model의 경우, 의료 용어 표현이 낯설 수도 있습니다. 하지만 만약 특화 영역의 corpus만 사용하여 언어모델을 생성할 경우, generalization 능력이 너무 떨어질 수도 있습니다.

- 일반 영역(general domain)
  - $P(\text{진정제}|\text{준비}, \text{된}) = 0.00001$
  - $P(\text{사나이}|\text{준비}, \text{된}) = 0.01$
- 특화 영역(specialized domain)
  - $P(\text{진정제}|\text{준비}, \text{된}) = 0.09$
  - $P(\text{약}|\text{준비}, \text{된}) = 0.04$
- Interpolated
  - $P(\text{진정제}|\text{준비}, \text{된}) = 0.5 * 0.09 + (10.5) * 0.00001 = 0.045005$

따라서 일반적인 대화에서와 다른 의미를 지닌 단어가 나올 수도 있고, 일반적인 대화에서는 희소(rare)한 표현(word sequence)가 훨씬 더 자주 등장 할 수 있습니다. 이런 상황들에 잘 대처하기 위해서 해당 영역 corpus로 생성한 언어모델을 섞어주어 해당 영역(domain)에 특화 시킬 수 있습니다.

## Back-off

너무 긴 단어 조합(word sequence)은 실제 훈련 corpus에서 굉장히 희소(rare, sparse)하기 때문에, 우리는 Markov assumption을 통해서 일반화(generalization) 할 수 있었습니다. 그럼 그것을 응용해서 좀 더 나아가 보도록 하겠습니다.

아래 수식을 보면 특정 n-gram의 확률을  $n$ 보다 더 작은 시퀀스에 대해서 확률을 구하여 선형결합(linear combination)을 계산 하는 것을 볼 수 있습니다. 아래와 같이

$n$ 보다 더 작은 시퀀스에 대해서도 확률을 가져옴으로써 smoothing을 통해 일반화 효과를 좀 더 얻을 수 있습니다. 사실 따라서 이 방법도 interpolation의 일종이라고 볼 수 있습니다.

$$\begin{aligned}\tilde{P}(w_n | w_{n-k}, \dots, w_{n-1}) = & \lambda_1 P(w_n | w_{n-k}, \dots, w_{n-1}) \\ & + \lambda_2 P(w_n | w_{n-k+1}, \dots, w_{n-1}) \\ & + \dots \\ & + \lambda_k P(w_n),\end{aligned}$$

where  $\sum_i \lambda_i = 1$ .

또한, 다음 단어를 예측 해 내는 추론(inference)에서도, 훈련 corpus에 존재하지 않는 (unseen) n-gram에 대해서도 훈련 corpus에 나타났던 단어 시퀀스(word sequence)가 있을 때까지 back-off하여, 다음 단어의 확률을 예측 해 볼 수 있습니다.

## Conclusion

n-gram 방식은 출현 횟수(count)를 통해 확률을 근사하기 때문에 굉장히 쉽고 간편합니다. 대신에 단점도 명확합니다. 훈련 corpus에 등장하지 않은 단어 조합은 확률을 정확하게 알 수 없습니다. 따라서 Markov assumption을 통해서 단어 조합에 필요한 조건을 간소화 시키고, 더 나아가 Smoothing과 Back-off 방식을 통해서 남은 단점을 보완하려 했습니다만, 이 또한 근본적인 해결책은 아니므로 실제로 음성인식이나 통계기반 기계번역에서 쓰이는 언어모델 어플리케이션 적용에 있어서 큰 난관으로 작용하였습니다. 하지만, 워낙 간단하고 명확하기 때문에 성공적으로 음성인식, 기계번역 등에 정착하였고 십 수년 동안 널리 사용되어 왔습니다.

## How to evaluate Language Model?

좋은 언어모델이란 실제 우리가 쓰는 언어에 대해서 최대한 비슷하게 확률 분포를 근사하는 모델(또는 파라미터,  $\theta$ )이 될 것입니다. 많이 쓰이는 문장 또는 표현일수록 높은 확률을 가져야 하며, 적게 쓰이는 문장 또는 이상한 문장 또는 표현일수록 확률은 낮아야 합니다. 즉, 테스트 문장을 잘 예측 해 낼 수록 좋은 언어모델이라고 할 수 있습니다. 문장을 잘 예측 한다는 것은, 다르게 말하면 주어진 테스트 문장이

언어모델에서 높은 확률을 가진다고 할 수 있습니다. 또는 문장의 앞부분이 주어지고, 다음에 나타나는 단어의 확률 분포가 실제 테스트 문장의 다음 단어에 대해 높은 확률을 갖는다면 더 좋은 언어모델이라고 할 수 있을 것 입니다. 그럼 이번 섹션은 언어모델의 성능을 평가하는 방법에 대해서 다루도록 하겠습니다.

## Perplexity

Perplexity 측정 방법은 explicit evaluation 방법의 하나입니다. 줄여서 PPL이라고 부르기도 합니다. PPL을 이용하여 테스트 문장에 대해서 언어모델을 이용하여 점수를 구하고 언어모델의 성능을 측정합니다. 문장에 대한 확률을 구하고 문장의 길이에 대해서 normalization 합니다. 수식은 아래와 같습니다.

$$\begin{aligned} \text{PPL}(w_1, w_2, \dots, w_n) &= P(w_1, w_2, \dots, w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1, w_2, \dots, w_N)}} \end{aligned}$$

문장이 길어지게 되면 문장의 확률은 당연히 굉장히 작아지게 됩니다. 따라서 우리는 문장의 길이( $N$ )로 제곱근을 취해주어 기하평균을 구하고, 문장 길이에 대해서 normalization을 해 주는 것을 볼 수 있습니다. 수식을 보면 문장의 확률이 분모에 들어가 있기 때문에, 확률이 높을수록 PPL은 작아지는 것을 쉽게 예상할 수 있습니다.

다시말해서 테스트 문장에 대해서 확률을 높게 예측할 수록 좋은 언어모델인 만큼, 해당 테스트 문장에 대한 PPL이 작을 수록 좋은 언어모델이라고 할 수 있습니다. 즉, PPL은 수치가 낮을수록 좋습니다. 또한, n-gram의 n이 클 수록 보통 더 낮은 PPL을 보여주기도 합니다.

위 PPL의 수식은 다시한번 Chain rule에 의해서

$$= \sqrt[N]{\frac{1}{\prod_{i=1}^N P(w_i|w_1, \dots, w_{i-1})}}$$

라고 표현 될 수 있고, 여기에 n-gram<sup>o1</sup> 적용 될 경우,

$$= \sqrt[N]{\frac{1}{\prod_{i=1}^N P(w_i|w_{i-n+1}, \dots, w_{i-1})}}$$

로 표현 될 수 있습니다.

## Perplexity의 해석

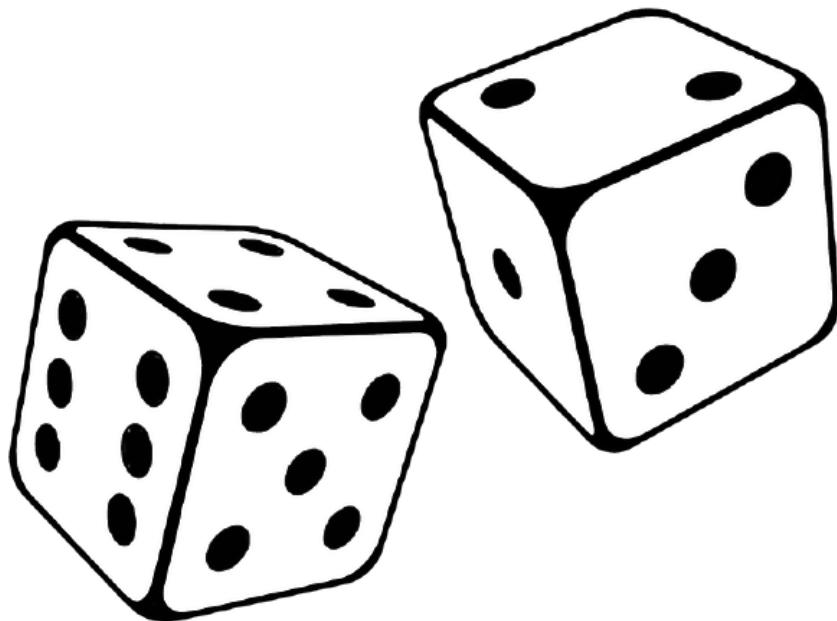


Figure 5:

Perplexity의 개념을 좀 더 짚고 넘어가도록 해 보겠습니다. 이것은 Perplexity의 수치를 해석하는 방법에 대해서라고 할 수 있습니다. 예를 들어 우리가 6면 주사위를 던져서 나오는 값을 통해 수열을 만들어낸다고 해 보겠습니다. 따라서 1부터 6까지 숫자의 출현 확률은 모두 같다(uniform distribution)고 가정하겠습니다. 그럼  $N$ 번 주사위를 던져 얻어내는 수열에 대한 perplexity는 아래와 같습니다.

$$PPL(x) = \left(\frac{1}{6}\right)^{\frac{1}{N}} = 6$$

매 time-step 가능한 경우의 수인 6이 PPL로 나왔습니다. 즉, PPL은 우리가 뺀어나갈 수 있는 branch(가지)의 숫자를 의미하기도 합니다. 다른 예를 들어 만약 20,000개의 어휘로 이루어진 뉴스 기사에 대해서 PPL을 측정한다고 하였을 때, 단어의 출현 확률이 모두 동일하다면 PPL은 20,000이 될 것입니다. 하지만 3-gram을 사용한 언어모델을 만들어 측정한 PPL이 30이 나왔다면, 우리는 이 언어모델을 통해 해당 신문기사에서 매번 기사의 앞 부분을 통해 다음 단어를 예측 할 때, 평균적으로 30개의 후보 단어 중에서 선택할 수 있다는 얘기가 됩니다. 따라서 우리는 perplexity를 통해서 언어모델의 성능을 단순히 측정할 뿐만 아니라 실제 어느정도인지 가늠해 볼 수도 있습니다.

### Entropy와 Perplexity의 관계

엔트로피(Entropy)는 물리학에서 중요하게 사용되는 개념이지만, 정보이론(Information Theory)에서도 굉장히 중요하게 사용 됩니다. 정보이론에서 엔트로피는 어떤 정보의 불확실성을 나타내는 수치로 사용 될 수 있습니다. 정보량과 불확실성은 어떤 관계를 가질까요? 불확실성은 일어날 것 같은 사건(likely event)의 확률로 정의할 수 있습니다. 따라서 불확실성과 정보량은 아래와 같은 관계를 가집니다.

- 자주 발생하는(일어날 확률이 높은) 사건은 낮은 정보량을 가진다.
- 반대로 드물게 발생하는(일어날 확률이 낮은) 사건은 높은 정보량을 가진다.

위의 예를 실제 사례를 들어 적용 해 보겠습니다.

---

번호	문장
1	내일 아침에는 해가 동쪽에서 뜬다.
2	내일 아침에는 해가 서쪽에서 뜬다.
3	대한민국 올 여름의 평균 기온은 섭씨 28도로 예상 된다.
4	대한민국 올 여름의 평균 기온은 섭씨 5도로 예상 된다.

---

누군가에게 위와 같은 말을 들었을 때, 어떤 말이 우리에게 큰 도움이 될까요? 몇십억년 동안 반복되온 아침 해가 뜨는 위치가 내일은 바뀐다는 정보는 정말 천지 개벽의 정보가 될 겁니다. 따라서 우리는 이처럼 확률이 낮을 수록 그 안에 포함된 정보량은 높다고 생각 할 수 있습니다.

섀넌(Claude Elwood Shannon)은 위와 같이 정보 엔트로피라는 개념과 함께 아래의 수식을 제시하였습니다. 확률 변수(Random Variable)  $X$ 의 값이  $x$ 인 경우의 정보량은 아래와 같이 표현 할 수 있습니다.

$$I(x) = -\log P(x)$$

$-\log$  를 취하였기 때문에, 0과 1 사이로 표현되는 확률은 0에 가까워질수록 지수적으로 높은 정보량을 가짐을 알 수 있습니다. 언어모델 관점에서 적용시켜 보면, 흔히 나올 수 없는 문장(확률이 낮은 문장)일수록 더 높은 정보량을 가질 것이라고 생각 할 수 있습니다.

우리는 이러한 정보량의 기대값을 엔트로피(entropy)라고 부릅니다.

$$H(P) = -\mathbb{E}_{X \sim P}[\log P(x)] = -\sum_{\forall x} P(x) \log P(x)$$

위 식을 해석 해 보면, 우리는 확률 분포  $P$ 로부터 발생한 사건  $X$ 의 정보량에 대한 기대값을 구하는 것이라고 할 수 있습니다.

### Cross Entropy Loss

크로스 엔트로피(Cross entropy)는 entropy로부터 한 걸음 더 나아가, 우리가 구하고자 하는 ground-truth 확률분포  $P$ 를 통해 우리가 학습중인 확률분포  $Q$ 의 정보량의 기대값을 이릅니다. 크로스 엔트로피의 수식은 아래와 같습니다.

$$H(P, Q) = -\mathbb{E}_{X \sim P}[\log Q(x)] = -\sum_{\forall x} P(x) \log Q(x)$$

이것을  $Q$  대신, 우리의 모델( $P_\theta$ )을 최적화 하기 위해 최소화(minimize)해야 하는 loss(손실)함수로 적용하여 보면 아래와 같습니다.

$$\mathcal{L} = H(P, P_\theta) = -\mathbb{E}_{Y|X \sim P}[\log P_\theta(y|x)]$$

여기서 우리는 N번의 Sampling 하는 과정을 통해 expectation을 제거할 수 있습니다.  
 - 강화학습(Reinforcement Learning) 챕터 참고

$$\begin{aligned} H(P, P_\theta) &= -\mathbb{E}_{Y|X \sim P}[\log P_\theta(w_i|w_{<i})] \\ &\approx -\frac{1}{N} \sum_{i=1}^N \log P_\theta(w_i|w_{<i}) \end{aligned}$$

여기서  $Y = \{y_1, y_2, \dots, y_N\}$ 는  $N$ 개의 단어로 이루어진 문장(word sequence)로 생각하고, 한 문장에 대한 cross entropy는 아래와 같이 표현할 수 있습니다.

$$\begin{aligned} \mathcal{L} &= -\frac{1}{N} \sum_{i=1}^N \log P_\theta(w_i|w_{<i}) \\ &= \log \left( \prod_{i=1}^N P_\theta(w_i|w_{<i}) \right)^{-\frac{1}{N}} \\ &= \log \sqrt[N]{\frac{1}{\prod_{i=1}^N P_\theta(w_i|w_{<i})}} \end{aligned}$$

여기에서 PPL 수식을 다시 떠올려 보겠습니다.

$$\begin{aligned} \text{PPL}(W) &= P(w_1, w_2, \dots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \dots, w_N)}} \\ &\quad \text{by chain rule,} \\ \text{PPL}(W) &= \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i|w_1, \dots, w_{i-1})}} \end{aligned}$$

앞서 정리했던 Cross Entropy와 수식이 비슷한 형태임을 알 수 있습니다. 따라서 PPL과 Cross Entropy의 관계는 아래와 같습니다.

$$\text{PPL} = \exp(\text{Cross Entropy})$$

따라서, 우리는 Maximum Likelihood Estimation(MLE)을 통해 parameter( $\theta$ )를 배울 때, cross entropy를 통해 얻은 ( $P_\theta$ 의 로그 확률 값) loss 값에 exp를 취함으로써, perplexity를 얻어 언어모델의 성능을 나타낼 수 있습니다. # n-gram Exercise with SRILM

SRILM은 음성인식, segmentation, 기계번역 등에 사용되는 통계 언어 모델(n-gram language model)을 구축하고 적용 할 수 있는 toolkit입니다. 이 책에서 다루는 다른

알고리즘이나 기법들에 비하면, SRI speech research lab에서 1995년부터 연구/개발해 온 유서깊은(?) toolkit입니다.

## Install SRILM

- 다운로드: <http://www.speech.sri.com/projects/srilm/download.html>

위의 주소에서 SRILM은 간단한 정보를 기입 한 후, 다운로드 가능합니다. 이후에 아래와 같이 디렉터리를 만들고 그 안에 압축을 풀어 놓습니다.

```
$ mkdir srilm  
$ cd ./srilm  
$ tar -xzvf ./srilm-1.7.2.tar.gz
```

디렉터리 내부에 Makefile을 열어 7번째 라인의 SRILM의 경로 지정 후에 주석을 해제하여 줍니다. 그리고 make 명령을 통해 SRILM을 빌드 합니다.

```
$ vi ./Makefile  
$ make
```

빌드가 정상적으로 완료 된 후에, PATH에 SRILM/bin 내부에 새롭게 생성된 디렉터리를 등록 한 후, export 해 줍니다.

```
PATH={SRILM_PATH}/bin/{MACHINE}:$PATH  
#PATH=/home/khkim/Workspace/nlp/srilm/bin/i686-m64:$PATH  
export PATH
```

그리고 아래와 같이 ngram-count와 ngram이 정상적으로 동작하는 것을 확인합니다.

```
$ source ~/.profile  
$ ngram-count -help  
$ ngram -help
```

## Prepare Dataset

이전 preprocessing 챕터에서 다루었던 대로 tokenize가 완료된 파일을 데이터로 사용합니다. 그렇게 준비된 파일을 training set과 test set으로 나누어 준비 합니다.

## Basic Usage

아래는 주로 SRILM에서 사용되는 프로그램들의 주요 arguments에 대한 설명입니다.

- ngram-count: LM을 훈련
  - vocab: lexicon file name
  - text: training corpus file name
  - order: n-gram count
  - write: output countfile file name
  - unk: mark OOV as
  - kndiscountn: Use Kneser-Ney discounting for N-grams of order n
- ngram: LM을 활용
  - ppl: calculate perplexity for test file name
  - order: n-gram count
  - lm: language model file name

## Language Modeling

위에서 나온 대로, language model을 훈련하기 위해서는 ngram-count 모듈을 이용합니다. 아래의 명령어는 예를 들어 kndiscount를 사용한 상태에서 tri-gram을 훈련하고 LM과 거기에 사용된 vocabulary를 출력하도록 하는 명령입니다.

```
$ time ngram-count -order 3 -kndiscount -text <text_fn> -lm <output_lm_fn> -write...
```

## Sentence Generation

아래의 명령은 위와 같이 ngram-count 모듈을 사용해서 만들어진 lm을 활용하여 문장을 생성하는 명령입니다. 문장을 생성 한 이후에는 preprocessing 챕터에서 다루었듯이 detokenization을 수행해 주어야 하며, pipeline을 통해 sed와 regular expression을 사용하여 detokenization을 해 주도록 해 주었습니다.

```
$ ngram -lm <input_lm_fn> -gen <n_sentence_to_generate> | sed "s/ //g" | sed "s/ / /g"
```

위와 같이 매번 sed와 regular expression을 통하는 것이 번거롭다면, preprocessing 챕터에서 구현한 detokenization.py python script를 통하여 detokenization을 수행 할 수도 있습니다.

## Evaluation

이렇게 language model을 훈련하고 나면 test set에 대해서 evaluation을 통해 얼마나 훌륭한 language model이 만들어졌는지 체크 할 필요가 있습니다. Language model에 대한 성능평가는 아래와 같은 명령을 통해 수행 될 수 있습니다.

```
$ ngram -ppl <test_fn> -lm <input_lm_fn> -order 3 -debug 2
```

아래는 위의 명령에 대한 예시입니다. 실행을 하면 OOVs(Out of Vocabularies)와 해당 test 문장들에 대한 perplexity가 나오게 됩니다. 주로 문장 수에 대해서 normalize를 수행한 (ppl1이 아닌) ppl을 참고 하면 됩니다.

```
$ ngram -ppl ./data/test.refined.tok.bpe.txt -lm ./data/ted_aligned.en.refined.tok.bpe.txt -order 3 -debug 2
...
...
file ./data/test.refined.tok.bpe.txt: 1000 sentences, 13302 words, 32 OOVs
0 zeroprobs, logprob= -36717.49 ppl= 374.1577 ppl1= 584.7292
```

위의 evaluation에서는 1,000개의 테스트 문장에 대해서 13,302개의 단어가 포함되어 있었고, 개중에 32개의 OOV가 발생하였습니다. 결국 이 테스트에 대해서는 약 374의 ppl이 측정되었습니다. 이 ppl을 여러가지 hyper-parameter 튜닝 또는 적절한 훈련데이터 추가를 통해서 낮추는 것이 관건이 될 것 입니다.

그리고 -debug 파라미터를 2를 주게 되면 아래와 같이 좀 더 자세한 각 문장과 단어 별 log를 볼 수 있습니다. 실제 language model 상에서 어떤 n-gram이 hit되었는지와 이에 대한 확률을 볼 수 있습니다. 3-gram이 없는 경우에는 2-gram이나 1-gram으로 back-off 되는 것을 확인 할 수 있고, back-off 시에는 확률이 굉장히 떨어지는 것을 볼 수 있습니다. 따라서 back-off를 통해서 unseen word sequence에 대해서 generalization을 할 수 있었지만, 여전히 성능에는 아쉬움이 남는 것을 알 수 있습니다.

```
I m pleased with the way we handled it .
p( I | <s> ) = [2gram] 0.06806267 [ -1.167091 ]
p( m | I ... ) = [1gram] 6.597231e-06 [ -5.180638 ]
p( pleased | m ... ) = [1gram] 6.094323e-06 [ -5.215075 ]
p( with | pleased ... ) = [2gram] 0.1292281 [ -0.8886431 ]
p( the | with ... ) = [2gram] 0.05536767 [ -1.256744 ]
p( way | the ... ) = [3gram] 0.003487763 [ -2.457453 ]
p( we | way ... ) = [3gram] 0.1344272 [ -0.8715127 ]
p( handled | we ... ) = [1gram] 1.902798e-06 [ -5.720607 ]
```

```

p( it | handled ...) = [1gram] 0.002173233 [-2.662894]
p( . | it ...) = [2gram] 0.05907027 [-1.228631]
p( </s> | . ...) = [3gram] 0.8548805 [-0.06809461]
1 sentences, 10 words, 0 OOVs
0 zeroprobs, logprob= -26.71738 ppl= 268.4436 ppl1= 469.6111

```

위의 결과에서는 10개의 단어가 주어졌고, 5번의 back-off이 되어 3-gram은 3개만 hit되었고, 4개의 2-gram과 4개의 1-gram이 hit되었습니다. 그리하여 -26.71의 log-probability가 계산되어, 268의 PPL로 환산되었음을 볼 수 있습니다.

### Interpolation

SRILM을 통해서 단순한 smoothing(or discounting) 뿐만이 아니라 interpolation을 수행 할 수도 있습니다. 이 경우에는 완성된 두 개의 별도의 language model이 필요하고, 이를 섞어주기 위한 hyper parameter lambda가 필요합니다. 아래와 같이 명령어를 입력하여 interpolation을 수행할 수 있습니다.

```
$ ngram -lm <input_lm_fn> -mix-lm <mix_lm_fn> -lambda <mix_ratio_between_0_and_1>
```

## Neural Network Language Model

### Against to Sparseness

앞서 설명한 것과 같이 기존의 n-gram 기반의 언어모델은 간편하지만 훈련 데이터에서 보지 못한 단어의 조합에 대해서 상당히 취약한 부분이 있었습니다. 그것의 근본적인 원인은 n-gram 기반의 언어모델은 단어간의 유사도를 알지 못하기 때문입니다. 예를 들어 우리에게 훈련 corpus로 아래와 같은 문장이 주어졌다고 했을 때,

- 고양이는 좋은 반려동물입니다.

사람은 단어간의 유사도를 알기 때문에 다음 중 어떤 확률이 더 큰지 알 수 있습니다. 하지만, 컴퓨터는 훈련 corpus에 해당 n-gram이 존재하지 않으면, count를 할 수 없기 때문에 확률을 구할 수 없고, 따라서 확률 간 비교를 할 수도 없습니다.

- $P(\text{반려동물}|\text{강아지는}, \text{좋은})$
- $P(\text{반려동물}|\text{자동차는}, \text{좋은})$

비록 강아지가 개의 새끼이고 포유류에 속하는 가축에 해당한다는 깊고 해박한 지식이 없을지라도, 강아지와 고양이 사이의 유사도가 자동차와 고양이 사이의 유사도보다 높은 것을 알기 때문에 자동차 보다는 강아지에 대한 반려동물의 확률이 더 높음을 유추할 수 있습니다. 하지만 n-gram 방식의 언어모델은 단어간의 유사도를 구할 수 없기 때문에, 이와 같이 훈련 corpus에서 보지 못한 단어(unseen word sequence)의 조합(n-gram)에 대해서 효과적으로 대처할 수 없습니다.

하지만 Neural Network LM은 word embedding을 사용하여 단어를 벡터화(vectorize)함으로써, 강아지와 고양이를 비슷한 vector로 학습하고, 자동차와 고양이 보다 훨씬 높은 유사도를 가지게 합니다. 따라서 NNLM이 훈련 corpus에서 보지 못한 단어의 조합을 보더라도, 비슷한 훈련 데이터로부터 배운 것과 유사하게 대처할 수 있습니다.

Neural Network LM은 많은 형태를 가질 수 있지만 우리는 가장 효율적이고 흔한 형태인 Recurrent Neural Network(RNN)의 일종인 Long Short Term Memory(LSTM)을 활용한 방식에 대해서 짚고 넘어가도록 하겠습니다.

## Recurrent Neural Network LM

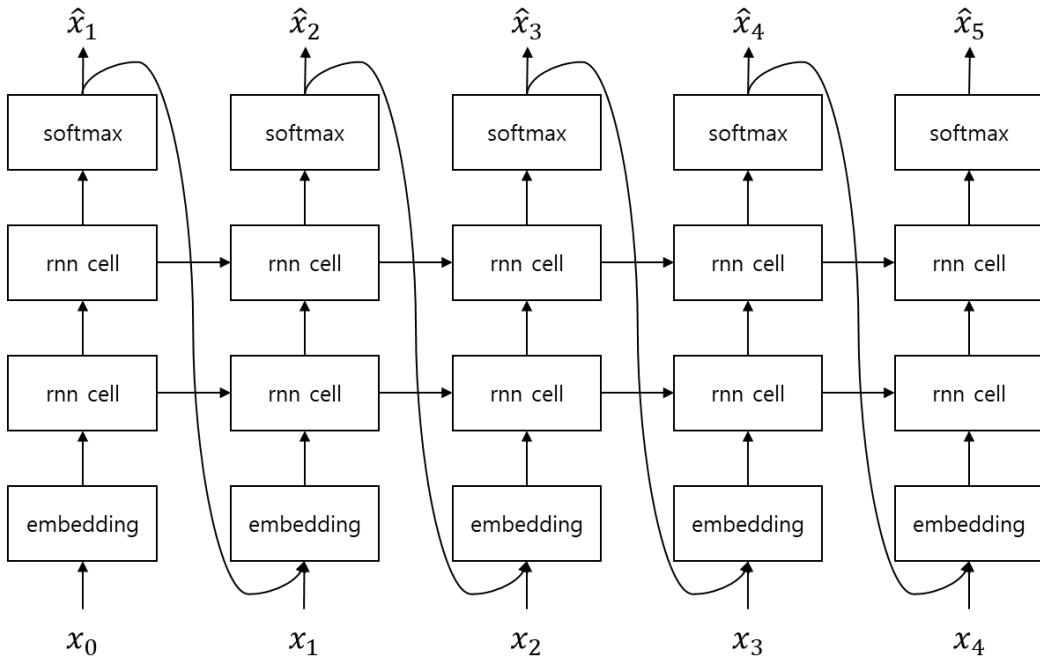


Figure 6: Recurrent Neural Language Model 아키텍처

Recurrent Neural Network Language Model (RNNLM)은 위와 같은 구조를 지니고 있습니다. 기존의 언어모델은 각각의 단어를 discrete한 존재로써 처리하였기 때문에, 문장(word sequence)의 길이가 길어지면 희소성(sparseness)문제가 발생하여 어려운 부분이 있었습니다. 따라서,  $n - 1$  이전까지의 단어만 (주로  $n = 3$ ) 조건부로 잡아 확률을 근사(approximation) 하였습니다. 하지만, RNN LM은 단어를 embedding하여 벡터화(vectorize)함으로써, 희소성 문제를 해소하였기 때문에, 문장의 첫 단어부터 모두 조건부에 넣어 확률을 근사 할 수 있습니다.

$$P(w_1, w_2, \dots, w_k) = \prod_{i=1}^k P(w_i | w_{<i})$$

로그를 취하여 표현 해보면 아래와 같습니다.

$$\log P(w_1, w_2, \dots, w_k) = \sum_{i=1}^k \log P(w_i | w_{<i})$$

## Implementation

이제 RNN을 활용한 언어모델을 구현해 보도록 하겠습니다. PyTorch로 구현하기에 앞서, 이를 수식화 해보면 아래와 같습니다. – language\_model.py가 이를 구현한 코드입니다.

$$X = \{x_0, x_1, \dots, x_n, x_{n+1}\}$$

where  $x_0 = BOS$  and  $x_{n+1} = EOS$ .

$$\hat{x}_{i+1} = \text{softmax}(\text{linear}_{\text{hidden} \rightarrow |V|}(\text{RNN}(\text{emb}(x_i))))$$

$$\hat{X}[1:] = \text{softmax}(\text{linear}_{\text{hidden} \rightarrow |V|}(\text{RNN}(\text{emb}(X[: -1])))),$$

where  $|V|$  is size of vocabulary.

위의 수식을 따라가 보면, 문장  $X$ 를 입력으로 받아 각 time-step 별( $x_i$ )로 Emb(embedding layer)에 넣어 정해진 차원(dimension)의 embedding vector를 얻습니다. RNN은 해당 embedding vector를 입력으로 받아, hidden size의 vector 형태로 반환합니다. 이 RNN 출력 vector를 linear layer를 통해 어휘(vocabulary)수 dimension의 vector로 변환 한 후, softmax를 취하여  $\hat{x}_{i+1}$ 을 구합니다.

여기서 우리는 LSTM을 사용하여 RNN을 대체 할 것이고, LSTM은 여러 층(layer)로 구성되어 있으며, 각 층 사이에는 dropout이 들어가 있습니다. 이 결과( $\hat{X}$ )를 이전 셉션에서 perplexity와 엔트로피(entropy)와의 관계를 설명하였듯이, cross entropy loss를 사용하여 모델( $\theta$ ) 최적화를 수행 합니다.

## Code

아래의 PyTorch 코드는 저자의 github에서 다운로드 할 수 있습니다. (업데이트 여부에 따라 코드가 약간 달라질 수 있습니다.)

- github repo url: <https://github.com/kh-kim/OpenNLMTK>

language\_model.py

```
import torch
import torch.nn as nn

import data_loader

class LanguageModel(nn.Module):

    def __init__(self,
                 vocab_size,
                 word_vec_dim=512,
                 hidden_size=512,
                 n_layers=4,
                 dropout_p=.2,
                 max_length=255
                 ):
        self.vocab_size = vocab_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p
        self.max_length = max_length
```

```

super(LanguageModel, self).__init__()

self.emb = nn.Embedding(vocab_size,
                      word_vec_dim,
                      padding_idx=data_loader.PAD
                     )
self.rnn = nn.LSTM(word_vec_dim,
                   hidden_size,
                   n_layers,
                   batch_first=True,
                   dropout=dropout_p
                  )
self.out = nn.Linear(hidden_size, vocab_size, bias=True)
self.log_softmax = nn.LogSoftmax(dim=2)

def forward(self, x):
    # |x| = (batch_size, length)
    x = self.emb(x)
    # |x| = (batch_size, length, word_vec_dim)
    x, (h, c) = self.rnn(x)
    # |x| = (batch_size, length, hidden_size)
    x = self.out(x)
    # |x| = (batch_size, length, vocab_size)
    y_hat = self.log_softmax(x)

    return y_hat

def search(self, batch_size=64, max_length=255):
    x = torch.LongTensor(batch_size, 1).to(next(self.parameters()).device).zero_()
    # |x| = (batch_size, 1)
    is undone = x.new_ones(batch_size, 1).float()

    y_hats, indice = [], []
    h, c = None, None
    while is undone.sum() > 0 and len(indice) < max_length:
        x = self.emb(x)
        # |emb_t| = (batch_size, 1, word_vec_dim)

```

```

x, (h, c) = self.rnn(x, (h, c)) if h is not None and c is not None else (x, (None, None))
# |x| = (batch_size, 1, hidden_size)
y_hat = self.log_softmax(x)
# |y_hat| = (batch_size, 1, output_size)
y_hats += [y_hat]

# y = torch.topk(y_hat, 1, dim = -1)[1].squeeze(-1)
y = torch.multinomial(y_hat.exp().view(batch_size, -1), 1)
y = y.masked_fill_((1. - is undone).byte(), data_loader.PAD)
is undone = is undone * torch.ne(y, data_loader.EOS).float()
# |y| = (batch_size, 1)
# |is undone| = (batch_size, 1)
indice += [y]

x = y

y_hats = torch.cat(y_hats, dim=1)
indice = torch.cat(indice, dim=-1)
# |y_hat| = (batch_size, length, output_size)
# |indice| = (batch_size, length)

return y_hats, indice

```

data\_loader.py

```
from torchtext import data, datasets
```

```
PAD, BOS, EOS = 1, 2, 3
```

```
class DataLoader():
```

```
def __init__(self,
            train_fn,
            valid_fn,
            batch_size=64,
```

```

        device='cpu',
        max_vocab=99999999,
        max_length=255,
        fix_length=None,
        use_bos=True,
        use_eos=True,
        shuffle=True
    ):

    super(DataLoader, self).__init__()

    self.text = data.Field(sequential=True,
                          use_vocab=True,
                          batch_first=True,
                          include_lengths=True,
                          fix_length=fix_length,
                          init_token='<BOS>' if use_bos else None,
                          eos_token='<EOS>' if use_eos else None
                        )

train = LanguageModelDataset(path=train_fn,
                            fields=[('text', self.text)],
                            max_length=max_length
                          )
valid = LanguageModelDataset(path=valid_fn,
                            fields=[('text', self.text)],
                            max_length=max_length
                          )

self.train_iter = data.BucketIterator(train,
                                      batch_size=batch_size,
                                      device='cuda:%d' % device if device
                                      shuffle=shuffle,
                                      sort_key=lambda x: -len(x.text),
                                      sort_within_batch=True
                                    )
self.valid_iter = data.BucketIterator(valid,
                                      batch_size=batch_size,

```

```

        device='cuda:%d' % device if device
        shuffle=False,
        sort_key=lambda x: -len(x.text),
        sort_within_batch=True
    )

self.text.build_vocab(train, max_size=max_vocab)

class LanguageModelDataset(data.Dataset):

    def __init__(self, path, fields, max_length=None, **kwargs):
        if not isinstance(fields[0], (tuple, list)):
            fields = [('text', fields[0])]

        examples = []
        with open(path) as f:
            for line in f:
                line = line.strip()
                if max_length and max_length < len(line.split()):
                    continue
                if line != '':
                    examples.append(data.Example.fromlist(
                        [line], fields))

    super(LanguageModelDataset, self).__init__(examples, fields, **kwargs)

```

trainer.py

```

import time
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.utils as torch_utils

```

```

import utils

def get_loss(y, y_hat, criterion, do_backward=True):
    batch_size = y.size(0)

    loss = criterion(y_hat.contiguous().view(-1, y_hat.size(-1)),
                     y.contiguous().view(-1)
                     )
    if do_backward:
        # since size_average parameter is off, we need to devide it by batch size
        loss.div(batch_size).backward()

    return loss

def train_epoch(model, criterion, train_iter, valid_iter, config):
    current_lr = config.lr

    lowest_valid_loss = np.inf
    no_improve_cnt = 0

    for epoch in range(1, config.n_epochs + 1):
        # optimizer = optim.Adam(model.parameters(), lr = current_lr)
        optimizer = optim.SGD(model.parameters(),
                              lr=current_lr
                              )
        print("current learning rate: %f" % current_lr)
        print(optimizer)

        sample_cnt = 0
        total_loss, total_word_count, total_parameter_norm, total_grad_norm = 0, 0
        start_time = time.time()
        train_loss = np.inf

        for batch_index, batch in enumerate(train_iter):
            optimizer.zero_grad()

```

```

current_batch_word_cnt = torch.sum(batch.text[1])
# Most important lines in this method.
# Since model takes BOS + sentence as an input and sentence + EOS as
# x(input) excludes last index, and y(index) excludes first index.
x = batch.text[0][:, :-1]
y = batch.text[0][:, 1:]
# feed-forward
y_hat = model(x)

# calculate loss and gradients with back-propagation
loss = get_loss(y, y_hat, criterion)

# simple math to show stats
total_loss += float(loss)
total_word_count += int(current_batch_word_cnt)
total_parameter_norm += float(utils.get_parameter_norm(model.parameters()))
total_grad_norm += float(utils.get_grad_norm(model.parameters()))

if (batch_index + 1) % config.print_every == 0:
    avg_loss = total_loss / total_word_count
    avg_parameter_norm = total_parameter_norm / config.print_every
    avg_grad_norm = total_grad_norm / config.print_every
    elapsed_time = time.time() - start_time

    print("epoch: %d batch: %d/%d\t|param|: %.2f\t|g_param|: %.2f\t|loss|: %.2f"
          % (epoch, batch_index + 1, config.max_batches, avg_parameter_norm,
             avg_grad_norm, loss))

    total_loss, total_word_count, total_parameter_norm, total_grad_norm = 0, 0, 0, 0
    start_time = time.time()

```

```

        train_loss = avg_loss

# Another important line in this method.
# In orther to avoid gradient exploding, we apply gradient clipping.
    torch_utils.clip_grad_norm_(model.parameters(),
                                config.max_grad_norm
                                )
# Take a step of gradient descent.
    optimizer.step()

    sample_cnt += batch.text[0].size(0)
    if sample_cnt >= len(train_iter.dataset.examples) * config.iter_ratio:
        break

    sample_cnt = 0
    total_loss, total_word_count = 0, 0

    model.eval()
    for batch_index, batch in enumerate(valid_iter):
        current_batch_word_cnt = torch.sum(batch.text[1])
        x = batch.text[0][:, :-1]
        y = batch.text[0][:, 1:]
        y_hat = model(x)

        loss = get_loss(y, y_hat, criterion, do_backward=False)

        total_loss += float(loss)
        total_word_count += int(current_batch_word_cnt)

        sample_cnt += batch.text[0].size(0)
        if sample_cnt >= len(valid_iter.dataset.examples):
            break

    avg_loss = total_loss / total_word_count
    print("valid loss: %.4f\tPPL: %.2f" % (avg_loss, np.exp(avg_loss)))

    if lowest_valid_loss > avg_loss:
        lowest_valid_loss = avg_loss

```

```

        no_improve_cnt = 0
    else:
        # decrease learning rate if there is no improvement.
        current_lr = max(config.min_lr, current_lr * config.lr_decay_rate)
        no_improve_cnt += 1

    model.train()

    model_fn = config.model.split(".")
    model_fn = model_fn[:-1] + [">%02d" % epoch,
                                "%.2f-%.2f" % (train_loss, np.exp(train_loss)),
                                "%.2f-%.2f" % (avg_loss, np.exp(avg_loss))]
                                ] + [model_fn[-1]]

    # PyTorch provides efficient method for save and load model, which uses
    torch.save({"model": model.state_dict(),
                "config": config,
                "epoch": epoch + 1,
                "current_lr": current_lr
                }, ".".join(model_fn))

    if config.early_stop > 0 and no_improve_cnt >= config.early_stop:
        break

```

utils.py

```

def get_grad_norm(parameters, norm_type = 2):
    parameters = list(filter(lambda p: p.grad is not None, parameters))

    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.grad.data.norm(norm_type)
            total_norm += param_norm ** norm_type
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:

```

```

        print(e)

    return total_norm

def get_parameter_norm(parameters, norm_type = 2):
    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.data.norm(norm_type)
            total_norm += param_norm ** norm_type
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm

```

train.py

```

import argparse

import torch
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from language_model import LanguageModel as LM
import trainer


def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required=True)
    p.add_argument('-train', required=True)
    p.add_argument('-valid', required=True)
    p.add_argument('-gpu_id', type=int, default=-1)

```

```

p.add_argument('-batch_size', type=int, default=64)
p.add_argument('-n_epochs', type=int, default=20)
p.add_argument('-print_every', type=int, default=50)
p.add_argument('-early_stop', type=int, default=3)
p.add_argument('-iter_ratio_in_epoch', type=float, default=1.)
p.add_argument('-lr_decay_rate', type=float, default=.5)

p.add_argument('-dropout', type=float, default=.3)
p.add_argument('-word_vec_dim', type=int, default=256)
p.add_argument('-hidden_size', type=int, default=256)
p.add_argument('-max_length', type=int, default=80)

p.add_argument('-n_layers', type=int, default=4)
p.add_argument('-max_grad_norm', type=float, default=5.)
p.add_argument('-lr', type=float, default=1.)
p.add_argument('-min_lr', type=float, default=.000001)

p.add_argument('-gen', type=int, default=32)

config = p.parse_args()

return config

def to_text(indice, vocab):
    lines = []

    for i in range(len(indice)):
        line = []
        for j in range(len(indice[i])):
            index = indice[i][j]

            if index == data_loader.EOS:
                # line += ['<EOS>']
                break
            else:
                line += [vocab.itos[index]]
```

```

        line = ' '.join(line)
        lines += [line]

    return lines

if __name__ == '__main__':
    config = define_argparser()

    loader = DataLoader(config.train,
                        config.valid,
                        batch_size=config.batch_size,
                        device=config.gpu_id,
                        max_length=config.max_length
                        )
    model = LM(len(loader.text.vocab),
               word_vec_dim=config.word_vec_dim,
               hidden_size=config.hidden_size,
               n_layers=config.n_layers,
               dropout_p=config.dropout,
               max_length=config.max_length
               )

# Let criterion cannot count PAD as right prediction, because PAD is easy to
loss_weight = torch.ones(len(loader.text.vocab))
loss_weight[data_loader.PAD] = 0
criterion = nn.NLLLoss(weight=loss_weight, size_average=False)

print(model)
print(criterion)

if config.gpu_id >= 0:
    model.cuda(config.gpu_id)
    criterion.cuda(config.gpu_id)

if config.n_epochs > 0:
    trainer.train_epoch(model,

```

```

        criterion,
        loader.train_iter,
        loader.valid_iter,
        config
    )

if config.gen > 0:
    total_gen = 0
    while total_gen < config.gen:
        current_gen = min(config.batch_size, config.gen - total_gen)
        _, indice = model.search(batch_size=current_gen)
        total_gen += current_gen

    lines = to_text(indice, loader.text.vocab)
    print('\n'.join(lines))

```

## Conclusion

NNLM은 word embedding vector를 사용하여 희소성(sparseness)을 해결하여 큰 효과를 보았습니다. 따라서 훈련 데이터셋에서 보지 못한 단어(unseen word sequence)의 조합에 대해서도 훌륭한 대처가 가능합니다. 하지만 그만큼 연산량에 있어서 n-gram에 비해서 매우 많은 대가를 치루어야 합니다. 단순히 table look-up 수준의 연산량을 필요로 하는 n-gram방식에 비해서 NNLM은 다수의 matrix 연산등이 포함된 feed forward 연산을 수행해야 하기 때문입니다. 그럼에도 불구하고 GPU의 사용과 점점 빨라지는 하드웨어 사이에서 NNLM의 중요성은 커지고 있고, 실제로도 많은 분야에 적용되어 훌륭한 성과를 거두고 있습니다. # Applications

사실 언어모델 자체를 단독으로 사용하는 경우는 굉장히 드뭅니다. 그렇다면 왜 언어모델이 중요할까요? 하지만, 자연어생성(Natural Language Generation, NLG)의 가장 기본이라고 할 수 있습니다. NLG는 현재 딥러닝을 활용한 NLP 분야에서 가장 큰 연구주제입니다. 기계번역에서부터 챗봇까지 모두 NLG의 영역에 포함된다고 할 수 있습니다. 이러한 NLG의 기본 초석이 되는 것이 바로 언어모델(LM)입니다. 따라서, LM 자체의 활용도는 그 중요성에 비해 떨어질지언정, LM의 중요성과 그 역할은 부인할 수 없습니다. 대표적인 활용 분야는 아래와 같습니다.

## Automatic Speech Recognition (ASR)

음성인식(Speech recognition) 시스템을 구성할 때, 언어모델은 중요하게 쓰입니다. 사실 실제 사람의 경우에도 말을 들을 때 언어모델이 굉장히 중요하게 작용합니다. 어떤 단어를 발음하고 있는지 명확하게 알아듣지 못하더라도, 머릿속에 저장되어 있는 언어모델을 이용하여 알아듣기 때문입니다. 예를 들어, 우리는 갑자기 땅뚱맞은 주제로 대화를 전환하게 되면 보통 한번에 잘 못알아듣는 경우가 많습니다. 컴퓨터의 경우에도 음소별 분류(classification)의 성능은 이미 사람보다 뛰어납니다. 다만, 사람에 비해 주변 문맥(context) 정보를 활용할 수 있는 능력, 눈치가 없기 때문에 음성인식률이 떨어지는 경우가 많습니다. 따라서, 그나마 좋은 언어모델을 학습하여 사용함으로써, 음성인식의 정확도를 높일 수 있습니다.

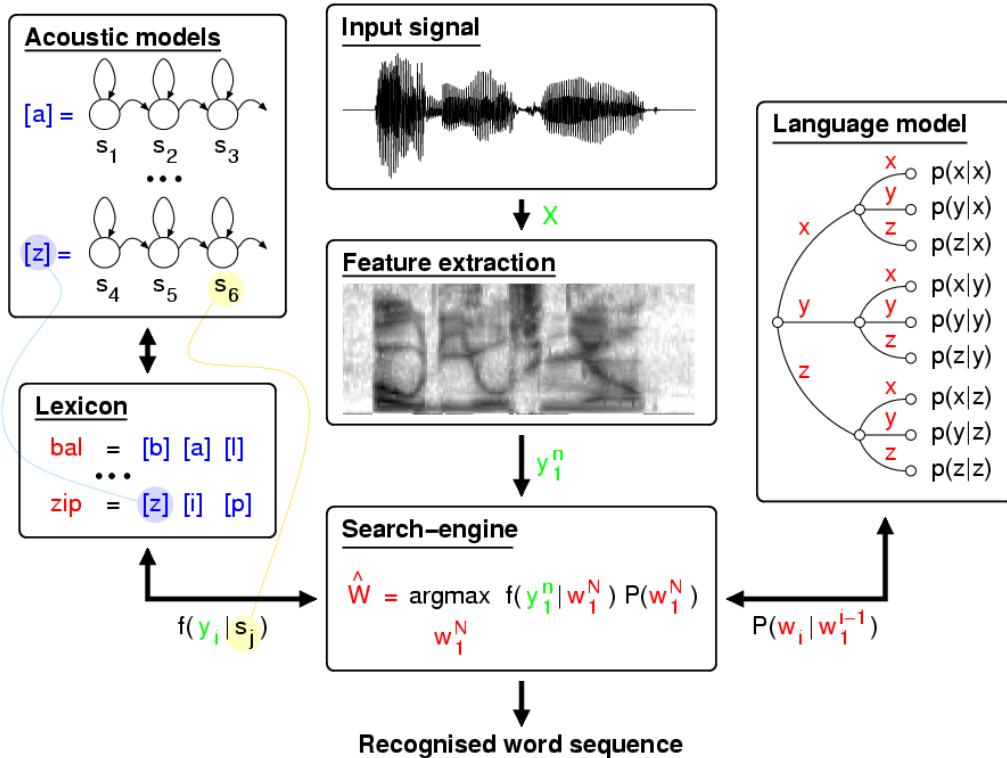


Figure 7: Traditional ASR System based on WFST, Image from web

아래는 음성인식 시스템의 수식을 개략적으로 나타낸 것 입니다. 음성 신호  $X$ 가 주어졌을 때 확률을 최대로 하는 문장  $\hat{Y}$ 를 구하는 것이 목표 입니다.

$$\hat{Y} = \operatorname{argmax}_{Y \in \mathcal{Y}} P(Y|X) = \operatorname{argmax}_{Y \in \mathcal{Y}} \frac{P(X|Y)P(Y)}{P(X)},$$

where X is an audio signal and Y is a word sequence,  $Y = \{y_1, y_2, \dots, y_n\}$ .

그럼 베이즈 정리(Bayes Theorem)에 의해서 수식은 AM과 LM 부분으로 나뉠 수 있습니다. 그리고 밀변(evidence)  $P(X)$ 는 날려버릴 수 있습니다.

$$\operatorname{argmax}_{Y \in \mathcal{Y}} \frac{P(X|Y)P(Y)}{P(X)} = \operatorname{argmax}_{Y \in \mathcal{Y}} P(X|Y)P(Y)$$

여기서  $P(X|Y)$ 가 음향모델(Acoustic Model, AM)을 의미하고,  $P(Y)$ 는 언어모델(LM)을 의미 합니다. 즉,  $P(Y)$ 는 문장의 확률을 의미하고,  $P(X|Y)$ 는 문장(단어 시퀀스 또는 음소 시퀀스)이 주어졌을 때, 해당 음향 시그널이 나타날 확률을 나타냅니다.

이처럼 LM은 AM과 결합하여, 문맥 정보에 기반하여 다음 단어를 예측하도록 함으로써, 음성인식의 성능을 더울 향상 시킬 수 있고, 매우 중요한 역할을 차지하고 있습니다.

## Machine Translation (MT)

번역 시스템을 구성 할 때에도 언어모델은 중요한 역할을 합니다. 기존 통계기반번역(Statistical Machine Translation, SMT) 시스템에서는 위의 음성인식 시스템과 매우 유사하게 LM이 번역모델(translation model)과 결합하여 자연스러운 문장을 만들어내도록 동작 하였으며, 신경망기계번역(Neural Machine Translation, NMT)의 경우에도 LM이 마찬가지로 매우 중요한 역할을 차지 합니다. 더 자세한 내용은 다음 챕터에서 다루도록 하겠습니다.

## Optical Character Recognition (OCR)

광학문자인식 시스템을 만들 때에도 언어모델이 사용 됩니다. 사진에서 추출하여 글자를 인식 할 때에 각 글자(character) 간의 확률을 정의하면 훨씬 더 높은 성능을 얻어낼 수 있습니다. 따라서, OCR에서도 언어모델의 도움을 받아 글자나 글씨를 인식합니다.

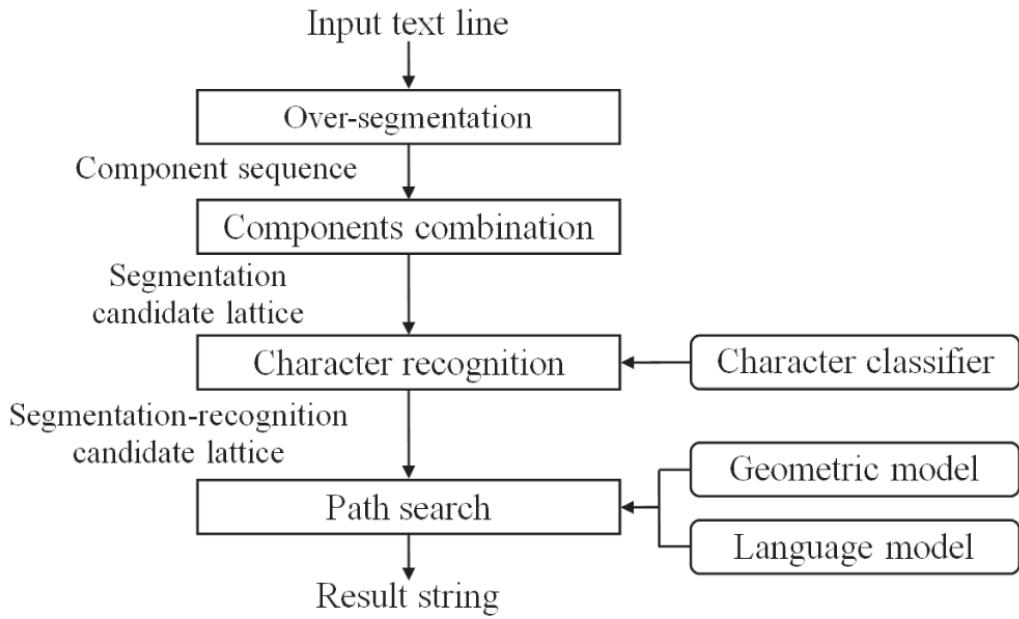


Figure 8: OCR system, Image from web

## Natural Language Generation (NLG)

사실 위에 나열한 ASR, MT, OCR도 주어진 정보를 바탕으로 문장을 생성해내는 NLG에 속한다고 볼 수 있습니다. 이외에도 NLG가 적용 될 수 있는 영역은 굉장히 많습니다. 기계학습의 결과물로써 문장을 만들어내는 작업은 모두 NLG의 범주에 속한다고 볼 수 있습니다. 예를 들어, 주어진 정보를 바탕으로 뉴스 기사를 쓸 수도 있고, 주어진 뉴스 기사를 요약하여 제목을 생성 해 낼 수도 있습니다. 또한, 사용자의 응답에 따라 대답을 생성 해 내는 chatbot도 생각 해 볼 수 있습니다.

## Others

이외에도 여러가지 영역에 정말 다양하게 사용됩니다. 검색엔진에서 사용자가 검색어를 입력하는 도중에 밑에 drop-down으로 제시되는 검색어 완성 등에도 언어모델이 사용 될 수 있습니다.

# Neural Machine Translation



Figure 1: Kyunghyun Cho

# Machine Translation (MT)

번역은 인류가 언어를 사용하기 시작한 이래로 큰 관심사 중에 하나였습니다. 그러한 의미에서 기계 번역(Machine Translation)은 단순히 언어를 번역하는 것이 아닌, 자연언어처리 영역에서의 종합예술이라 할 수 있습니다. 사실 이번 챕터를 위해서 이전 챕터들을 다루었다고 해도 과언이 아닙니다. Neural Machine Translation (NMT)는 end-to-end learning 으로써, Rule Based Machine Translation (RBMT), Statistical Machine Translation (SMT)로 이어져온 기계 번역의 30년 역사 중에서 가장 큰 성취를 이루어냈습니다. 이번 챕터는 NMT에 대한 인사이트를 얻을 수 있도록, seq2seq with attention의 동작 방식/원리를 이해하고, 더 나아가 응용 방법에 대해서도 소개 합니다. 또한, 기계번역 시스템을 만들기 위한 프로세스와 최신 연구 동향을 아울러 소개 합니다.

## Objective

$$\hat{e} = \operatorname{argmax} P_{f \rightarrow e}(e|f)$$

번역의 궁극적인 목표는 어떤 언어( $f$ , e.g. french)의 문장이 주어졌을 때, 우리가 원하는 언어( $e$ , e.g. english)로 확률을 최대로 하는 문장을 찾아내는 것입니다.

## Why it is so hard?

인간의 언어는 컴퓨터의 언어(e.g. programming language)와 같이 명확하지 않습니다. 우리는 언어의 모호성(ambiguity)을 적극적으로 활용함으로써, 의사소통의 효율을 극대화 하였습니다. 예를 들어 우리는 정보나 단어를 생략하고 문장을 짧게 만든다거나, 같은 단어와 어절이라고 하더라도 때에 따라 다른 의미로 해석될 수 있습니다. 더욱이 한국어의 경우에는 첫 챕터에서 다루었듯이 어순이 불규칙하고 주어가 생략 되는 등, 다른 언어에 비해 더욱 그 효율이 극대화 되었습니다. 또한, 언어라는 것은 그 민족의 문화를 담고 있기 때문에, 수 천년의 세월동안 쌓여온 사람들의 의식, 철학이 담겨져있고, 그러한 문화의 차이들로 하여금 번역을 더욱 어렵게 만듭니다. 결국, 이러한 특징들은 기계가 우리의 말을 번역하고자 할 때 큰 장벽으로 다가옵니다.

원문	오번역
In brightest day, in blackest night,	일기가 좋은 날, 진흙같은 어두운 밤,

---

원문	오번역
No evil shall escape my sight.	아니다 이 악마야, 내 앞에서 사라지지.
Let those who worship evil's might,	누가 사악한 수도악마를 숭배하는지 볼까,
Beware my power, Green Lantern's light!!!	나의 능력을 조심해라, 그린 랜턴 빛!

---

Why it is so important?

하지만 이러한 어려움에도 불구하고 기계 번역은 우리에게 꼭 필요한 과제입니다. 이 순간에도 전세계에서는 기계번역을 통해서 많은 일들이 일어납니다. Facebook과 같은 세계인이 소통하는 SNS, Amazon과 같은 전 세계를 대상으로 하는 인터넷 쇼핑몰에서도 번역 서비스를 제공하며 사용자들은 이를 통해 편리함을 얻을 수 있습니다.

## History

### Rule based Machine Translation

규칙 기반의 기계번역은 우리가 흔히 어릴때 부터 배워 온 방식으로 가장 전통적인 방식의 번역입니다. 주어진 문장의 구조를 분석하고, 그 분석에 따라 규칙을 세우고 분류를 나누어서 정해진 규칙에 따라서 번역을 합니다. 사람의 경우에는 일반화(generalization) 능력이 뛰어나기 때문에 몇 가지 규칙으로도 훌륭하게 적용하여 번역을 수행할 수 있지만, 컴퓨터의 경우에는 사람에 비해 일반화 능력이 현저히 떨어지기 때문에 규칙 기반 기계번역은 매우 어렵습니다. 잘 만들어진 규칙 아래에서는 밑에서 다를 SMT에 비해서 자연스러운 표현이 가능하지만, 그 규칙을 일일히 사람이 만들어내야 하므로 번역기를 만드는데 많은 자원과 시간이 소모됩니다. 따라서 번역 언어쌍을 확장할 때에도, 매번 새로운 규칙을 찾아내고 적용해야 하기 때문에 굉장히 불리합니다.

### Statistical Machine Translation

NMT이전에 세상을 지배하던 번역 방식입니다. 대량의 양방향 코퍼스로부터 통계를 얻어내어 번역 시스템을 구성합니다. Google이 자신의 번역 시스템에 도입하면서 더욱 유명해졌습니다. 이 시스템 또한 여러가지 모듈로 이루어져 굉장히 복잡합니다.



Figure 2: 대표적인 번역 실패 사례

통계기반 방식을 사용하므로 언어쌍을 확장할 때, 대부분의 알고리즘이나 시스템은 유지되므로 기존의 규칙기반 기계번역(RBMT)에 비하여 매우 유리하였습니다.

### Neural Machine Translation

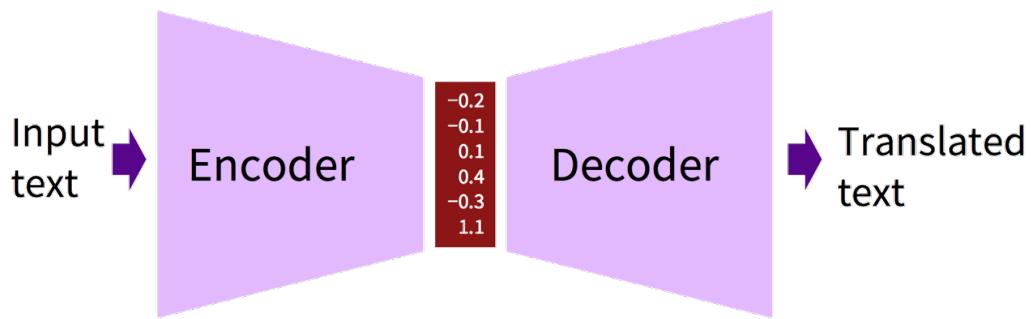


Figure 3: Image from CS224n

사실, 딥러닝 이전의 인공지능의 전성기(1980년대)에도 신경망을 사용하여 기계번역 문제를 해결하려는 시도는 여럿 있었습니다. 당시에도 Encoder → Decoder 형태의 구조를 가지고 있었지만, 당연히 지금과 같은 성능을 내기는 어려웠습니다.

## Progress in Machine Translation

[Edinburgh En-De WMT newstest2013 Cased BLEU; NMT 2015 from U. Montréal]

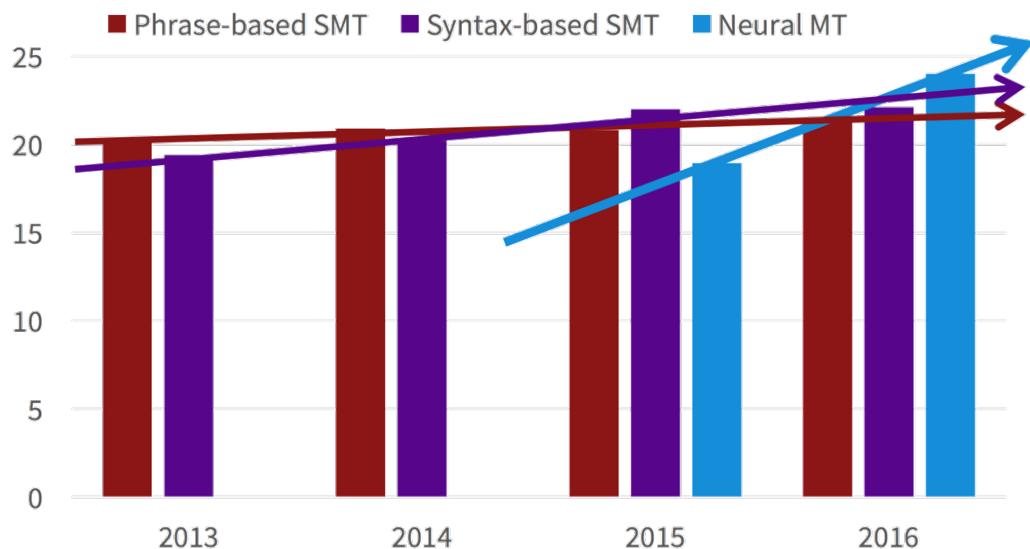


Figure 4: Image from CS224n

## Invasion of NMT

현재의 NMT 방식이 제안되고, 곧 기존의 SMT 방식을 크게 앞질러 버렸습니다.  
이제는 구글의 번역기에도 NMT 방식이 사용됩니다.

## Why it works well?

왜 Neural Machine Translation이 잘 동작하는 것일까요?

번호 항목 내용

1 End-NMT  
to- 이전의  
end SMT의  
Mode에는  
번역시스템이  
여러가지  
모듈로  
구성이  
되어  
있었고,  
이로  
하여금  
시스템의  
복잡도를  
증가시켜  
훈련에  
있어서  
훨씬  
지금보다  
어려운  
경향이  
있었습니다.  
하지만  
NMT는  
단  
하나의  
모델로  
번역을  
해결함으로써,  
성능을  
극대화  
하였습니다.

---

---

### 번호 항목 내용

---

2 Better 신경망  
lan- 언어모델(Neural  
guageNet-  
modelwork  
Lan-  
guage  
Model)을  
기반으로  
하는  
구조이므로  
기준의  
SMT방식의  
언어모델보다  
더  
강합니다.  
따라서  
희소성(sparseness)문제가  
해결  
되었으며,  
자연스러운  
번역  
결과  
문장을  
생성함에  
있어서  
더  
강점을  
나타냅니다.

---

---

### 번호 항목 내용

---

3 Great Neural  
con-Net-  
text work의  
embedding  
집분  
발휘하여  
문장의  
의미를  
벡터화(vectorize)하는  
능력이  
뛰어납니다.  
따라서,  
노이즈나  
희소성(sparseness)에도  
훨씬  
더  
잘  
대처할  
수  
있게  
되었습니다.

---

## Sequence to Sequence

### Architecture Overview

먼저 번역 또는 seq2seq 모델을 이용한 작업을 간단하게 수식화 해보겠습니다.

$$\theta^* \approx \underset{\theta}{\operatorname{argmax}} P(Y|X; \theta) \text{ where } X = \{x_1, x_2, \dots, x_n\}, Y = \{y_1, y_2, \dots, y_m\}$$

$P(Y|X; \theta)$ 를 최대로 하는 모델 파라미터( $\theta$ )를 Maximum Likelihood Estimation(MLE)를 통해 찾아야 합니다. 즉, 모델 파라미터가 주어졌을 때, source

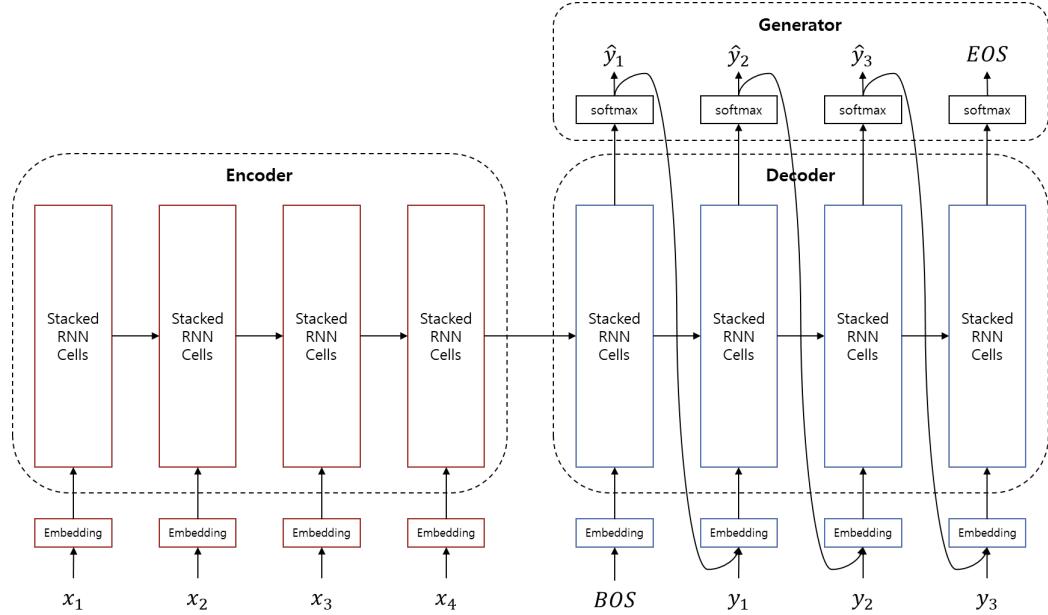


Figure 5: 기본적인 Sequence-to-Sequence 아키텍처

문장  $X$ 를 받아서 target 문장  $Y$ 를 반환할 확률을 최대로 하는 모델 파라미터를 학습하는 것입니다. 이를 위해서 seq2seq는 크게 3개의 서브 모듈(encoder, decoder, generator)로 구성되어 있습니다.

## Encoder

인코더는 source 문장을 입력으로 받아 문장을 함축하는 sentence embedding vector로 만들어 냅니다.  $P(X)$ 를 모델링 하는 것이라고 볼 수 있습니다. 사실 새로운 형태라기 보단, 이전 챕터에서 다루었던 텍스트 분류(Text Classification)에서 사용되었던 RNN 모델과 거의 같다고 볼 수 있습니다.  $P(X)$ 를 모델링하여, 주어진 문장을 벡터화(vectorize)하여 해당 도메인의 매니폴드(manifold or hyper-plane)의 어떤 한 점에 투영 시키는 작업이라고 할 수 있습니다.

다만, 기존의 텍스트 분류 문제에서는 모든 정보가 필요하지 않기 때문에 (예를 들어 감성분석(Sentiment Analysis)에서는 “나는”과 같이 중립적인 단어는 감성을 분류하는데 필요하지 않기 때문에 해당 정보를 굳이 간직해야 하지 않을 수도 있습니다.) vector로 만들어내는 과정에서 많은 정보를 간직하지 않아도 되지만, 기계번역을 위한 sentence embedding vector를 생성하기 위해서는 최대한 많은

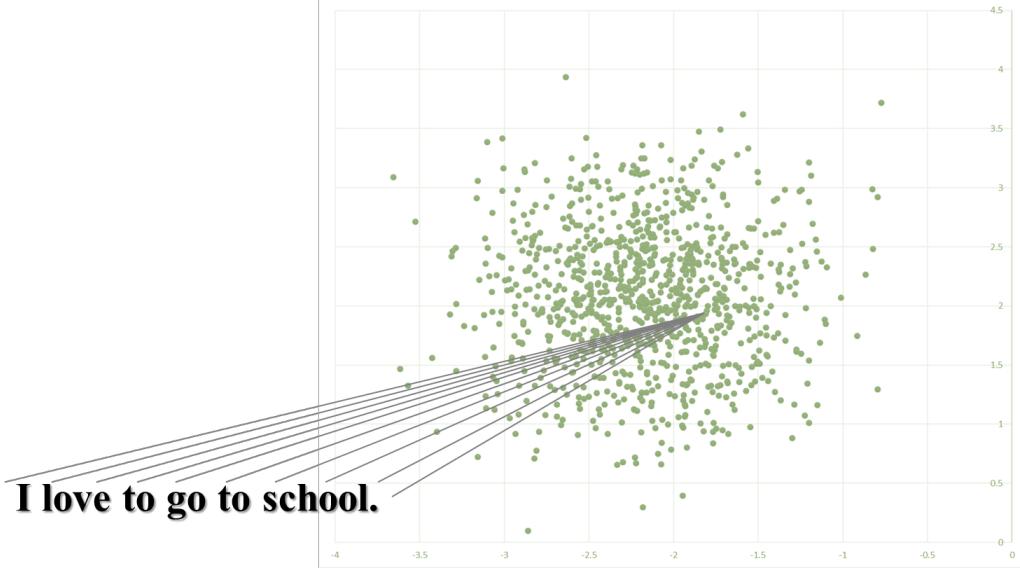


Figure 6: 3개의 구성요소로 이루어진 Sequence-to-Sequence 아키텍처

정보를 간직해야 할 것 입니다.

$$h_t^{src} = \text{RNN}_{enc}(\text{emb}_{src}(x_t), h_{t-1}^{src})$$

$$H^{src} = [h_1^{src}; h_2^{src}; \dots; h_n^{src}]$$

인코더(encoder)를 수식으로 나타내면 위와 같습니다.  $[;]$ 는 concatenate를 의미합니다. 위의 수식은 time-step 별로 GRU를 통과시킨 것을 나타낸 것이고, 사실상 실제 구현을 하면 아래와 같이 됩니다.

$$H^{src} = \text{RNN}_{enc}(\text{emb}_{src}(X), h_0^{src})$$

### Decoder

마찬가지로 디코더(decoder)도 사실 새로운 개념이 아닙니다. 이전 챕터에서 다루었던 신경망언어모델(Nerual Network Langauge Model, NNLM)의 연장선으로써, 조건부 신경망언어모델(Conditional Neural Network Language Model)이라고 할 수 있습니다. 위에서 다루었던 seq2seq모델의 수식을 좀 더 time-step에 대해서 풀어서 써보면 아래와 같습니다.

$$P_{\theta}(Y|X) = \prod_{t=1}^m P_{\theta}(y_t|X, y_{<t})$$

$$\log P_{\theta}(Y|X) = \sum_{t=1}^m \log P_{\theta}(y_t|X, y_{<t})$$

보면 RNNLM의 수식에서 조건부 랜덤 변수(random variable) 부분에  $X$ 가 추가된 것을 확인 할 수 있습니다. 즉, 이제까지 번역하여 생성한 (이전 time-step의) 단어들과 인코더의 결과에 기반해서 현재 time-step의 단어를 생성하기 위해 유추하는 작업을 수행합니다.

$$h_t^{tgt} = \text{RNN}_{dec}(\text{emb}_{tgt}(y_{t-1}), h_{t-1}^{tgt}) \text{ where } h_0^{tgt} = h_n^{src} \text{ and } y_0 = BOS$$

위 수식은 디코더를 나타낸 것입니다. 특기할 점은 디코더 입력의 초기값으로써,  $y_0$ 에  $BOS$ 를 넣어준다는 것 입니다.

### Generator

이 모듈은 아래와 같이 디코더에서 벡터를 받아 softmax를 계산하여 최고 확률을 가진 단어를 선택하는 단순한 작업을 하는 모듈입니다.  $|Y| = m$ 일 때,  $y_m$ 은  $EOS$  토큰이 됩니다. 주의할 점은 이 마지막  $y_m$ 은 디코더 계산의 종료를 나타내기 때문에, 이론상으로는 디코더의 입력으로 들어가는 일이 없습니다.

$$\hat{y}_t = \text{softmax}(\text{linear}_{hs \rightarrow |V_{tgt}|}(h_t^{tgt})) \text{ and } \hat{y}_m = EOS$$

where  $hs$  is hidden size of RNN, and  $|V_{tgt}|$  is size of output vocabulary.

### Applications of seq2seq

이와 같이 구성된 Seq2seq 모델은 꼭 기계번역 문제에서만 사용해야 하는 것이 아니라 정말 많은 분야에 적용할 수 있습니다. 특정 도메인의 time-series 데이터 또는 sequential한 입력을 다른 도메인의 sequential한 데이터로 출력하는데 탁월한 능력을 발휘합니다.

Seq2seq Applications	Task (From-To)
Neural Machine Translation (NMT)	특정 언어 문장을 입력으로 받아 다른 언어의 문장으로 출력
Chatbot	사용자의 문장 입력을 받아 대답을 출력
Summarization	긴 문장을 입력으로 받아 같은 언어의 요약된 문장으로 출력
Other NLP Task	사용자의 문장 입력을 받아 프로그래밍 코드로 출력 등
Automatic Speech Recognition (ASR)	사용자의 음성을 입력으로 받아 해당 언어의 문자열(문장)으로 출력
Lip Reading	입술 움직임의 동영상을 입력으로 받아 해당 언어의 문장으로 출력
Image Captioning	변형된 seq2seq를 사용하여 이미지를 입력으로 받아 그림을 설명하는 문장을 출력

## Limitation

사실 seq2seq는 AutoEncoder와 굉장히 역할이 비슷하다고 볼 수 있습니다. 그 중에서도 특히 time-series 데이터에 대한 강점이 있는 모델이라고 볼 수 있습니다. 하지만 아래와 같은 한계점들이 있습니다.

### Memorization

Neural Network 모델은 데이터를 압축하는데에 탁월한 성능(Manifold Assumption 참고)을 지녔습니다. 하지만, seq2seq를 통하여도 도라에몽의 주머니처럼 무한하게 정보를 압축 할 수 없습니다. 따라서 압축 할 수 있는 정보는 한계가 있기 때문에, 문장(또는 sequence)이 길어질수록 압축 성능이 떨어지게 됩니다. 비록 LSTM이나 GRU를 사용함으로써 RNN에 비하여 성능을 끌어올릴 수 있었지만, 한계가 있기 마련입니다.

### Lack of Structural Information

현재 주류의 딥러닝 자연어처리에서는 문장을 이해함에 있어서 구조 정보를 사용하기보단, 단순히 시계열(time-series) 데이터로 다루는 경향이 있습니다. 비록 이러한 접근방법은 현재까지 대성공을 거두고 있지만, 다음 단계로 나아가기 위해서는 구조 정보도 필요할 것이라 생각하는 사람들이 많습니다.

## Chatbot?

사실 이 항목은 단점이라기보다는 그냥 당연한 이야기일 수 있습니다. seq2seq는 시계열 데이터를 입력으로 받아서 다른 도메인의 시계열 데이터로 출력하는 능력이 뛰어납니다. 따라서, 처음에는 많은 사람들이 seq2seq를 잘 훈련시키면 챗봇의 기능도 어느정도 할 수 있지 않을까 하는 기대를 했습니다. 하지만 자세히 생각해보면, 대화의 흐름에서 대답은 질문에 비해서 새로운 정보(지식-knowledge, 문맥-context)가 추가 된 경우가 많습니다. 따라서 기존의 전형적인 seq2seq의 문제(번역, 요약)등은 새로운 정보의 추가가 없기 때문에 잘 해결할 수 있었지만, 대화의 경우에는 좀 더 발전된 구조가 필요할 것 입니다.

## Code

NMT를 목표로하는 seq2seq를 PyTorch로 구현하는 방법을 소개합니다. 이번 챕터에서 사용될 전체 코드는 저자의 깃허브에서 다운로드 할 수 있습니다. (업데이트 여부에 따라 코드가 약간 다를 수 있습니다.)

- github repo url: <https://github.com/kh-kim/simple-nmt>

### Encoder Class

```
class Encoder(nn.Module):  
  
    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):  
        super(Encoder, self).__init__()  
  
        # Be aware of value of 'batch_first' parameter.  
        # Also, its hidden_size is half of original hidden_size, because it is b  
        self.rnn = nn.LSTM(word_vec_dim, int(hidden_size / 2), num_layers = n_layer  
  
    def forward(self, emb):  
        # |emb| = (batch_size, length, word_vec_dim)  
  
        if isinstance(emb, tuple):  
            x, lengths = emb  
            x = pack(x, lengths.tolist(), batch_first = True)
```

```

else:
    x = emb

    y, h = self.rnn(x)
    # |y| = (batch_size, length, hidden_size)
    # |h[0]| = (num_layers * 2, batch_size, hidden_size / 2)

if isinstance(emb, tuple):
    y, _ = unpack(y, batch_first = True)

return y, h

```

### Pack Padded Sequence

아래는 pack\_padded\_sequence 함수가 동작하는 모습입니다. 이 함수는 기존의 샘플 별 mini-batch를 time-step 별로 표현해 줍니다. PackedSequence로 표현된 time-step별 mini-batch는 각 time-step별 샘플의 숫자를 추가적인 정보로 갖고 있습니다. 따라서, 이를 위해서는 mini-batch 내에는 가장 긴 길이의 문장부터 차례대로 정렬되어 있어야 합니다.

```

a = [torch.tensor([1, 2, 3]), torch.tensor([3, 4])]
b = torch.nn.utils.rnn.pad_sequence(a, batch_first=True)
>>>
tensor([[ 1,  2,  3],
       [ 3,  4,  0]])
torch.nn.utils.rnn.pack_padded_sequence(b, batch_first=True, lengths=...
>>>PackedSequence(data=tensor([ 1,  3,  2,  4,  3]), batch_sizes=tens

```

### Decoder Class

디코더 클래스의 코드는 이후 섹션에서 다루기로 합니다.

### Generator Class

```

class Generator(nn.Module):

    def __init__(self, hidden_size, output_size):

```

```

super(Generator, self).__init__()

self.output = nn.Linear(hidden_size, output_size)
self.softmax = nn.LogSoftmax(dim = -1)

def forward(self, x):
    # |x| = (batch_size, length, hidden_size)

    y = self.softmax(self.output(x))
    # |y| = (batch_size, length, output_size)

    # Return log-probability instead of just probability.
    return y

```

### Loss function

seq2seq는 기본적으로 각 time-step 별로 가장 확률이 높은 단어를 선택하는 분류 작업(classification task)이므로, cross entropy를 손실함수(loss function)으로 사용합니다. 또한 기본적으로 조건부 언어모델(conditional language model)이라고 볼 수 있기 때문에, 이전 언어모델 챕터에서 다루었듯이 perplexity를 통해 번역 모델의 성능을 나타낼 수 있고, 이 또한 (cross) entropy와 매우 깊은 연관을 가집니다.

아래는 손실(loss)값을 계산하기 위해 PyTorch로부터 손실함수를 준비하는 모습입니다. 사실, 실제 구현할 때에는 softmax layer + cross entropy를 사용하기보단, log softmax layer + negative log likelihood를 사용합니다.

```

loss_weight = torch.ones(output_size)
loss_weight[data_loader.PAD] = 0
criterion = nn.NLLLoss(weight = loss_weight, size_average = False)

```

아래와 같이 cross entropy 수식은 실제 정답의 확률과 feed-forward를 통해 얻은 신경망( $\theta$ )의 해당 로그(log)확률 값을 곱하여 평균을 구합니다.

$$\begin{aligned}
\mathcal{L}(P, P_{\theta}) &= -\frac{1}{N} \sum_{i=1}^N P(y_i) \log P_{\theta}(y_i) \\
&\approx -\frac{1}{N} \sum_{i=1}^N \log P_{\theta}(y_i)
\end{aligned}$$

따라서 softmax layer 대신, log softmax layer를 사용하여 로그 확률을 구하고, 수식의 나머지 작업을 수행하면 됩니다.

```
def get_loss(y, y_hat, criterion, do_backward = True):
    # |y| = (batch_size, length)
    # |y_hat| = (batch_size, length, output_size)
    batch_size = y.size(0)

    loss = criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguous().view(-1))
    if do_backward:
        loss.div(batch_size).backward()

    return loss
```

## Attention

### Motivation

Attention은 쿼리(Query)와 비슷한 값을 가진 키(Key)를 찾아서 그 값(Value)을 얻는 과정입니다. 따라서, 우리가 흔히 json이나 프로그래밍에서 널리 사용하는 Key-Value 방식과 비교하며 attention에 대해서 설명 하겠습니다.

### Key-Value function

Attention을 본격 소개하기 전에 먼저 우리가 알고 있는 자료형을 짚고 넘어갈까 합니다. Key-Value 또는 Python에서 Dictionary라고 부르는 자료형입니다.

```
>>> dic = {'computer': 9, 'dog': 2, 'cat': 3}
```

위와 같이 Key와 Value에 해당하는 값을 넣고 Key를 통해 Value 값에 접근 할 수 있습니다. 좀 더 바꿔 말하면, Query가 주어졌을 때, Key 값에 따라 Value 값에 접근 할 수 있습니다. 위의 작업을 함수로 나타낸다면, 아래와 같이 표현할 수 있을겁니다. (물론 실제 Python Dictionary 동작은 매우 다릅니다.)

```
def key_value_func(query):
    weights = []
```

```

for key in dic.keys():
    weights += [is_same(key, query)]

weight_sum = sum(weights)
for i, w in enumerate(weights):
    weights[i] = weights[i] / weight_sum

answer = 0

for weight, value in zip(weights, dic.values()):
    answer += weight * value

return answer

def is_same(key, query):
    if key == query:
        return 1.
    else:
        return .0

```

코드를 살펴보면, 순차적으로 dic 내부의 key값들과 query 값을 비교하여, key가 같을 경우 weights에 1.0을 추가하고, 다른 경우에는 0.0을 추가합니다. 그리고 weights를 weights의 총 합으로 나누어 weights의 합이 1이 되도록 (마치 확률과 같아) normalize하여 줍니다. 다시 dic 내부의 value값들과 weights의 값을 inner product (스칼라곱, dot product) 합니다. 즉,  $weight = 1.0$ 인 경우에만 value 값을 answer에 더합니다.

### Differentiable Key–Value function

좀 더 발전시켜서, 만약 is\_same 함수 대신에 다른 함수를 써 보면 어떻게 될까요? how\_similar라는 key와 query 사이의 유사도를 리턴해 주는 가상의 함수가 있다고 가정해 봅시다.

```

>>> query = 'puppy'
>>> how_similar('computer', query)
0.1
>>> how_similar('dog', query)
0.9

```

```
>>> how_similar('cat', query)
0.7
```

그리고 해당 함수에 'puppy'라는 단어를 테스트 해 보았더니 위와 같은 값을 리턴해 주었다고 해 보겠습니다. 그럼 아래와 같이 실행될 겁니다.

```
>>> query = 'puppy'
>>> key_value_func(query)
2.823 # = .1 / (.9 + .7 + .1) * 9 + .9 / (.9 + .7 + .1) * 2 + .7 / (.9 + .7 + .1)
```

2.823라는 값이 나왔습니다. 강아지와 고양이, 그리고 컴퓨터의 유사도의 비율에 따른 dic의 값의 비율을 지녔다라고 볼 수 있습니다. `is_same` 함수를 쓸 때에는 두 값이 같은지 if문을 통해 검사하고 값을 할당했기 때문에, 미분을 할 수 없거나 할 수 있더라도 gradient가 0이 됩니다. 하지만, 이제 우리는 `key_value_func`을 딥러닝에 사용할 수 있습니다.

## Differentiable Key–Value Vector function

- 만약, dic의 value에는 100차원의 vector로 들어있었다면 어떻게 될까요?
- 거기에, query와 key 값 모두 vector라면 어떻게 될까요? 즉, Word Embedding Vector라면?
- how\_similar 함수는 이 vector들 간의 cosine similarity를 반환 해 주는 함수라면?
- 그리고, dic의 key 값과 value 값이 서로 같다면 어떻게 될까요?

그럼 다시 가상의 함수를 만들어보겠습니다. word2vec 함수는 단어를 입력으로 받아서 그 단어에 해당하는 미리 정해진 word embedding vector를 반환 해 준다고 가정하겠습니다. 그럼 좀 전의 how\_similar 함수는 두 벡터 간의 cosine similarity 값을 반환 할 겁니다.

```
def key_value_func(query):
    weights = []

    for key in dic.keys():
        weights += [how_similar(key, query)]      # cosine similarity

    weights = softmax(weights)      # weight      softmax      .
    answer = 0
```

```

for weight, value in zip(weights, dic.values()):
    answer += weight * value

return answer

```

이번에 key\_value\_func는 그럼 그 값을 받아서 weights에 저장 한 후, 모든 weights의 값이 채워지면 softmax를 취할 겁니다. 여기서 softmax는 weights의 합의 크기를 1로 고정시키는 정규화 역할을 합니다. 따라서 유사도의 총 합에서 차지하는 비율 만큼 weight의 값이 채워질 겁니다.

```

>>> len(word2vec('computer'))
100
>>> word2vec('dog')
[0.1, 0.3, -0.7, 0.0, ...
>>> word2vec('cat')
[0.15, 0.2, -0.3, 0.8, ...
>>> dic = {word2vec('computer'): word2vec('computer'), word2vec('dog'): word2vec(
>>>
>>> query = 'puppy'
>>> answer = key_value_func(word2vec(query))

```

그럼 이제 answer의 값에는 어떤 벡터 값이 들어 있을 겁니다. 그 벡터는 ‘puppy’ 벡터와 ‘dog’, ‘computer’, ‘cat’ 벡터들의 코사인 유사도에 따라서 값이 정해집니다.

즉, 이 함수는 query와 비슷한 key 값을 찾아서 비슷한 정도에 따라서 weight를 나누고, 각 key의 value값을 weight 값 만큼 가져와서 모두 더하는 것입니다. 이것이 Attention<sup>o</sup> 하는 역할입니다.

## Attention for Machine Translation task

### Overview

그럼 번역에서 attention은 어떻게 작용할까요? 번역 과정에서는 인코더의 각 time-step 별 출력을 Key와 Value로 삼고, 현재 time-step의 디코더 출력을 Query로 삼아 attention을 취합니다.

항목	구성
Query	현재 time-step의 decoder output

---

항목	구성
Keys	각 time-step 별 encoder output
Values	각 time-step 별 encoder output

---

>>> context\_vector = attention(query = decoder\_output, keys = encoder\_outputs, values = encoder\_outputs)

Attention을 추가한 seq2seq의 수식은 아래와 같은 부분이 추가/수정 됩니다.

$$w = \text{softmax}(h_t^{tgt} W \cdot H^{src})$$

$$c = H^{src} \cdot w \text{ and } c \text{ is a context vector.}$$

$$\tilde{h}_t^{tgt} = \tanh(\text{linear}_{2hs \rightarrow hs}([h_t^{tgt}; c]))$$

$$\hat{y}_t = \text{softmax}(\text{linear}_{hs \rightarrow |V_{tgt}|}(\tilde{h}_t^{tgt}))$$

where  $hs$  is hidden size of RNN, and  $|V_{tgt}|$  is size of output vocabulary.

원하는 정보를 attention을 통해 인코더에서 획득한 후, 해당 정보를 디코더의 출력과 concatenate하여  $\tanh$ 를 취한 후, softmax 계산을 통해 다음 time-step의 입력이 되는  $\hat{y}_t$ 을 구합니다.

### Linear Transformation

이때, 각 입력 파라미터들은 다음을 의미한다고 볼 수 있습니다.

---

항목	의미
decoder_output	현재 time-step 까지 번역 된 target language 단어들 또는 문장, 의미
encoder_outputs	각 time-step 에서의 source language 단어 또는 문장, 의미

---

사실 신경망 내부의 각 차원들은 숨겨진 특징 값(latent feature)이므로 딱 잘라 정의할 수 없습니다. 하지만 분명한건, source 언어와 target 언어가 다르다는 것입니다. 따라서 단순히 벡터 내적을 해 주기보단 source 언어와 target 언어 간에 연결고리를 하나 놓아주어야 합니다. 그래서 우리는 두 언어의 embedding hyper plane이 선형(linear) 관계에 있다고 가정하고, 내적 연산을 수행하기 전에 선형 변환(linear transformation)을 해 줍니다.

위와 같이 꼭 번역이 아니더라도, 두 다른 도메인(domain) 사이의 변환을 위해서 사용합니다.

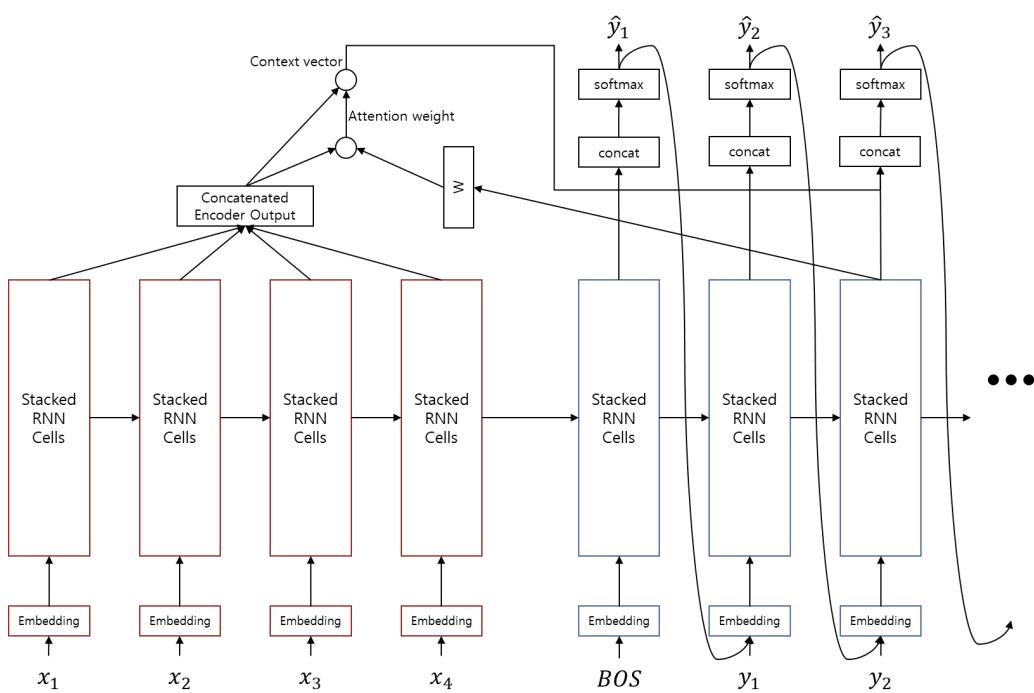


Figure 7: Attention<sup>o</sup> 추가된 Sequence-to-Sequence의 아키텍처

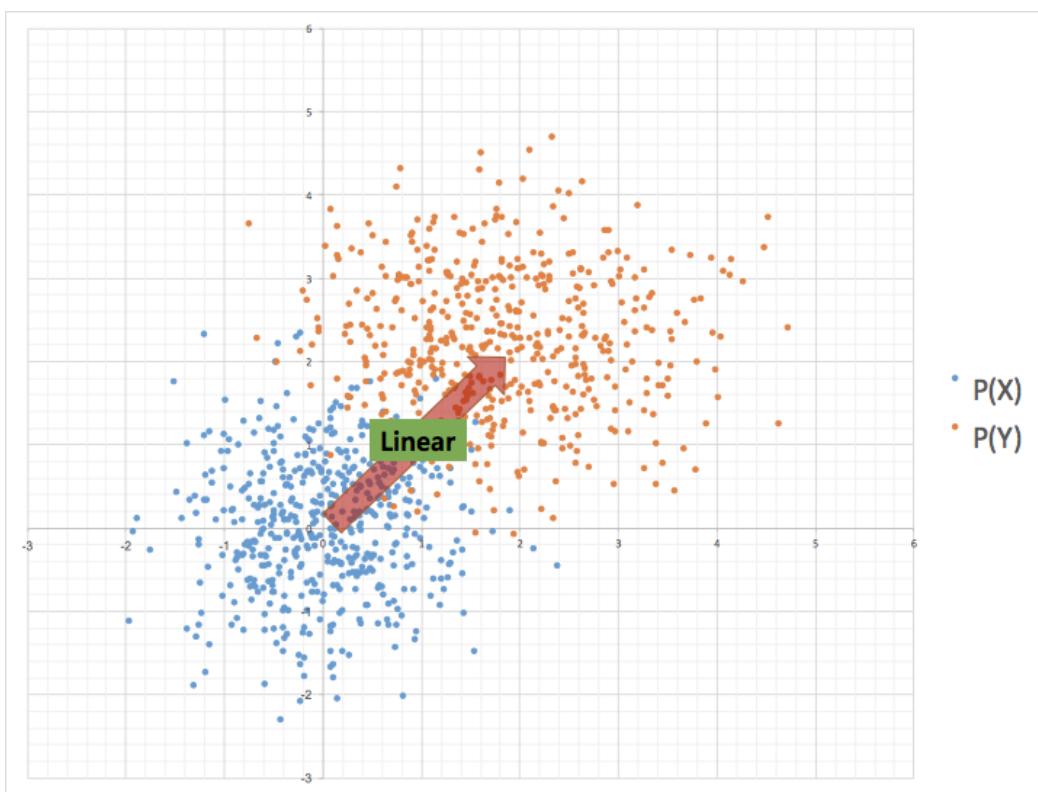


Figure 8: 인코더의 출력값과 디코더의 출력값 사이의 선형 맵핑

## Why

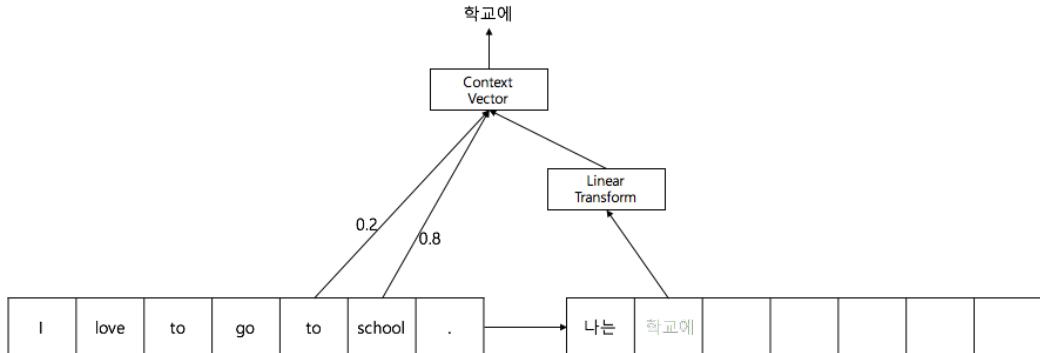


Figure 9: Attention이 번역에서 동작하는 직관적인 예

왜 Attention이 필요한 것일까요? 기존의 seq2seq는 두 개의 RNN(인코더와 디코더)로 이루어져 있습니다. 여기서 문장의 embedding 벡터에 해당하는 인코더의 정보를 디코더의 hidden state (LSTM의 경우에는 + cell state)로 전달해야 합니다. 그리고 디코더는 인코더로부터 넘겨받은 hidden state로부터 다시 새로운 문장을 만들어냅니다. 이때, hidden state만으로는 문장의 모든 정보를 완벽하게 전달하기 힘들기 때문입니다. 따라서 디코더의 각 time-step마다, 시간을 뛰어넘어, hidden state의 정보에 따라 필요한 인코더의 정보에 접근하여 끌어다 쓰겠다는 것입니다.

## Evaluation

Attention을 사용하지 않은 seq2seq는 전반적으로 성능이 떨어짐을 알 수 있을 뿐만 아니라, 특히 문장이 길어질수록 성능이 더욱 하락함을 알 수 있습니다. 하지만 이에 비해서 attention을 사용하면 문장이 길어지더라도 성능이 크게 하락하지 않음을 알 수 있습니다.

## Code

```
class Attention(nn.Module):  
  
    def __init__(self, hidden_size):  
        super(Attention, self).__init__()
```

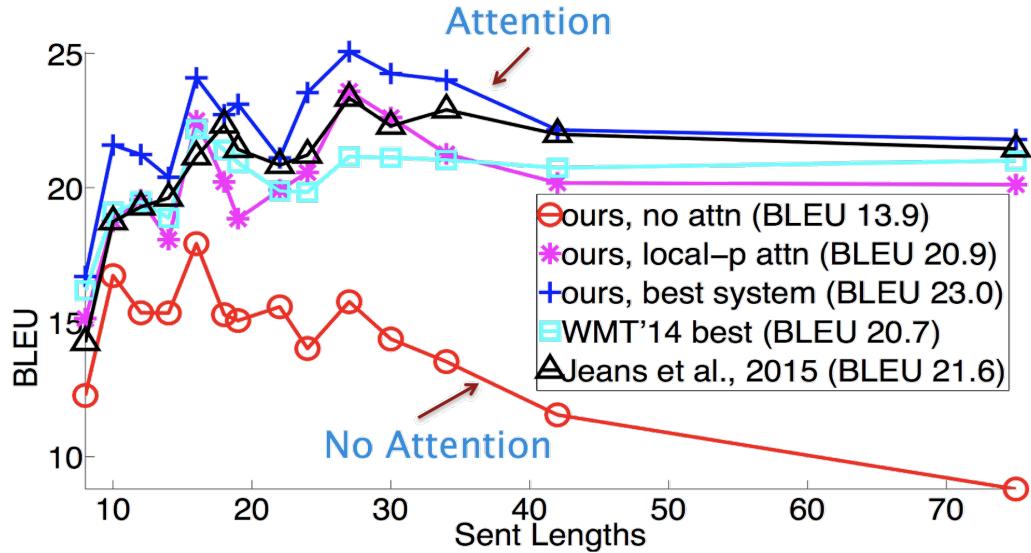


Figure 10: Image from CS224n

```

self.linear = nn.Linear(hidden_size, hidden_size, bias = False)
self.softmax = nn.Softmax(dim = -1)

def forward(self, h_src, h_t_tgt, mask = None):
    # |h_src| = (batch_size, length, hidden_size)
    # |h_t_tgt| = (batch_size, 1, hidden_size)
    # |mask| = (batch_size, length)

    query = self.linear(h_t_tgt.squeeze(1)).unsqueeze(-1)
    # |query| = (batch_size, hidden_size, 1)

    weight = torch.bmm(h_src, query).squeeze(-1)
    # |weight| = (batch_size, length)
    if mask is not None:
        # Set each weight as -inf, if the mask value equals to 1.
        # Since the softmax operation makes -inf to 0, masked weights would
        # Thus, if the sample is shorter than other samples in mini-batch, t
        weight.masked_fill_(mask, -float('inf'))
    weight = self.softmax(weight)

```

```

context_vector = torch.bmm(weight.unsqueeze(1), h_src)
# |context_vector| = (batch_size, 1, hidden_size)

return context_vector

```

## Input Feeding

### Overview

각 time-step의 디코더 출력값과 Attention 결과값을 concatenate한 이후에 제너레이터 모듈에서 softmax를 취하여  $\hat{y}_t$ 을 구합니다. 하지만 이러한 softmax 과정에서 많은 정보(예를 들어 attention 정보 등)가 손실됩니다. 따라서 단순히 다음 time-step에  $\hat{y}_t$ 을 입력으로 넣어 주는 것보다, concatenation 레이어의 출력도 같이 넣어주어 계산한다면, 좀 더 정보의 손실 없이 더 좋은 효과를 얻을 수 있습니다.

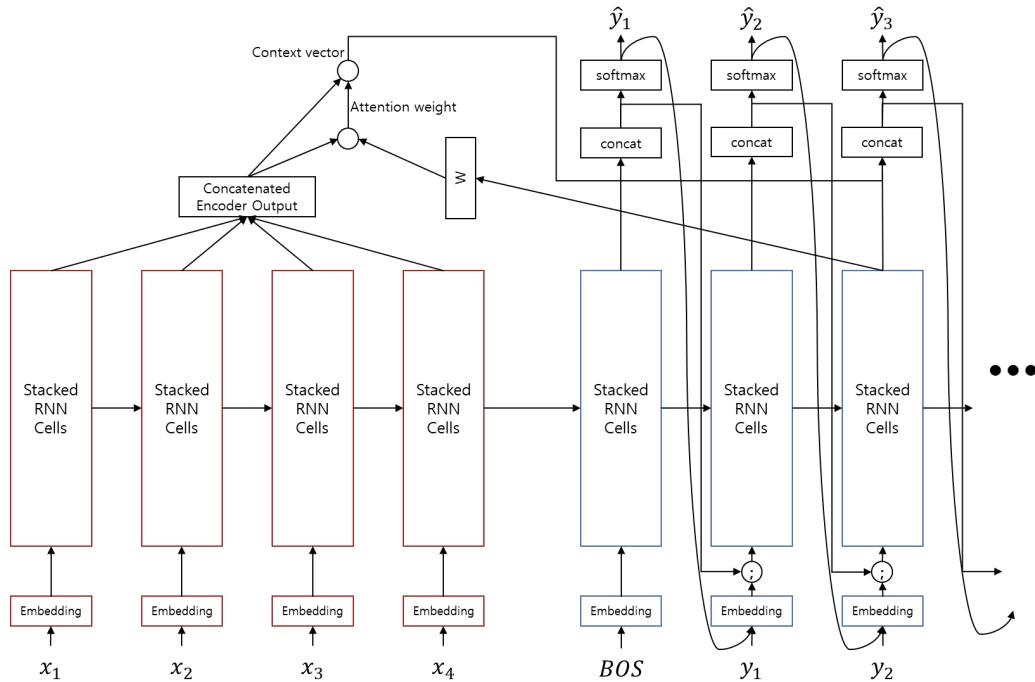


Figure 11: Input Feeding이 추가 된 Sequence-to-Sequence 아키텍처

$y$ 와 달리 concatenation 레이어의 출력은  $y$ 가 임베딩 레이어에서 dense 벡터로 변환되고 난 이후에 임베딩 벡터와 concatenate되어 디코더 RNN에 입력으로 주어지게 됩니다. 이러한 과정을 input feeding이라고 합니다.

$$\begin{aligned} h_t^{src} &= \text{RNN}_{enc}(\text{emb}_{src}(x_t), h_{t-1}^{src}) \\ H^{src} &= [h_1^{src}; h_2^{src}; \dots; h_n^{src}] \end{aligned}$$

$$h_t^{tgt} = \text{RNN}_{dec}([\text{emb}_{tgt}(y_{t-1}); \tilde{h}_{t-1}^{tgt}], h_{t-1}^{tgt}) \text{ where } h_0^{tgt} = h_n^{src} \text{ and } y_0 = BOS.$$

$$\begin{aligned} w &= \text{softmax}(h_t^{tgt} W \cdot H^{src}) \\ c &= H^{src} \cdot w \text{ and } c \text{ is a context vector.} \end{aligned}$$

$$\tilde{h}_t^{tgt} = \tanh(\text{linear}_{2hs \rightarrow hs}([h_t^{tgt}; c]))$$

$$\hat{y}_t = \text{softmax}(\text{linear}_{hs \rightarrow |V_{tgt}|}(\tilde{h}_t^{tgt}))$$

where  $hs$  is hidden size of RNN, and  $|V_{tgt}|$  is size of output vocabulary.

위는 attention과 input feeding이 추가된 seq2seq의 처음부터 끝까지를 수식으로 나타낸 것입니다.  $\text{RNN}_{dec}$ 는 이제  $\tilde{h}_{t-1}^{tgt}$ 를 입력으로 받기 때문에, 모든 time-step을 한번에 처리하도록 구현할 수 없다는 점이 구현상의 차이점입니다. 그리고 실제 이전 time-step의 예측값을 사용할 수 없는 teacher forcing의 단점은 어느정도 보완해 주는 역할을 합니다.

## Disadvantage

이 방식은 훈련 속도 저하라는 단점을 가집니다. input feeding 이전의 방식에서는 훈련 할 때, teacher forcing 방식을 사용하여, 인코더와 마찬가지로 디코더도 모든 time-step에 대해서 한번에 계산하는 작업이 가능했습니다. 하지만 input feeding으로 인해, 디코더 RNN의 입력으로 이전 time-step의 결과( $\tilde{h}_t^{tgt}$ )가 필요하게 되어, 마치 추론(inference)할때처럼 auto-regressive 속성으로 인해 각 time-step 별로 순차적으로 계산을 해야 합니다.

하지만 이 단점이 크게 부각되지 않는 이유는 어차피 추론(inference)단계에서는 디코더는 input feeding이 아니더라도 time-step 별로 순차적으로 계산되어야 하기 때문입니다. 추론 단계에서는 이전 time-step의 출력값인  $\hat{y}_t$ 를 디코더(정확하게는 디코더 RNN이전의 임베딩 레이어)의 입력으로 사용해야 하기 때문에, 어쩔 수 없이 병렬(parallel)처리가 아닌 순차적으로 계산해야 합니다. 따라서 추론 할 때, input feeding으로 인한 속도 저하는 거의 없습니다.

## Evaluation

NMT system	Perplexity	En/De BLEU
Base	10.6	11.3
Base + reverse	9.9	12.6(+1.3)
Base + reverse + dropout	8.1	14.0(+1.4)
Base + reverse + dropout + attention	7.3	16.8(+2.8)
Base + reverse + dropout + attention + feed input	6.4	18.1(+1.3)

현재 방식을 처음 제안한 [Loung et al,2015] Effective Approaches to Attention-based Neural Machine Translation에서는 실험 결과를 위와 같이 주장하였습니다. 실험 대상은 아래와 같습니다.

- Baseline: 기본적인 seq2seq 모델
- Reverse: Bi-directional LSTM을 encoder에 적용
- Dropout: probability 0.2
- Global Attention
- Input Feeding

우리는 이 실험에서 attention과 input feeding을 사용함으로써, 훨씬 더 나은 성능을 얻을 수 있음을 알 수 있습니다.

## Code

NMT를 목표로하는 seq2seq를 PyTorch로 구현하는 방법을 소개합니다. 이번 챕터에서 사용될 전체 코드는 저자의 깃허브에서 다운로드 할 수 있습니다. (업데이트 여부에 따라 코드가 약간 다를 수 있습니다.)

- github repo url: <https://github.com/kh-kim/simple-nmt>

## Decoder Class

```
class Decoder(nn.Module):

    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
        super(Decoder, self).__init__()

        # Be aware of value of 'batch_first' parameter and 'bidirectional' parameter
        self.rnn = nn.LSTM(word_vec_dim + hidden_size, hidden_size, num_layers = n_layers)

    def forward(self, emb_t, h_t_1_tilde, h_t_1):
        # |emb_t| = (batch_size, 1, word_vec_dim)
        # |h_t_1_tilde| = (batch_size, 1, hidden_size)
        # |h_t_1[0]| = (n_layers, batch_size, hidden_size)
        batch_size = emb_t.size(0)
        hidden_size = h_t_1[0].size(-1)

        if h_t_1_tilde is None:
            # If this is the first time-step,
            h_t_1_tilde = emb_t.new(batch_size, 1, hidden_size).zero_()

        # Input feeding trick.
        x = torch.cat([emb_t, h_t_1_tilde], dim = -1)

        # Unlike encoder, decoder must take an input for sequentially.
        y, h = self.rnn(x, h_t_1)

    return y, h
```

## Sequence-to-Sequence

### Initialization

```
def __init__(self, input_size, word_vec_dim, hidden_size, output_size, n_layers):
    self.input_size = input_size
    self.word_vec_dim = word_vec_dim
    self.hidden_size = hidden_size
```

```

    self.output_size = output_size
    self.n_layers = n_layers
    self.dropout_p = dropout_p

    super(Seq2Seq, self).__init__()

    self.emb_src = nn.Embedding(input_size, word_vec_dim)
    self.emb_dec = nn.Embedding(output_size, word_vec_dim)

    self.encoder = Encoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
    self.decoder = Decoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
    self.attn = Attention(hidden_size)

    self.concat = nn.Linear(hidden_size * 2, hidden_size)
    self.tanh = nn.Tanh()
    self.generator = Generator(hidden_size, output_size)

```

### Mask Generation

```

def generate_mask(self, x, length):
    mask = []

    max_length = max(length)
    for l in length:
        if max_length - l > 0:
            # If the length is shorter than maximum length among samples,
            # set last few values to be 1s to remove attention weight.
            mask += [torch.cat([x.new_ones(1, 1).zero_(), x.new_ones(1, (max_
        else:
            # If the length of the sample equals to maximum length among sam
            # set every value in mask to be 0.
            mask += [x.new_ones(1, 1).zero_()]

    mask = torch.cat(mask, dim = 0).byte()

    return mask

```

## Convert Hidden State from Encoder to Decoder

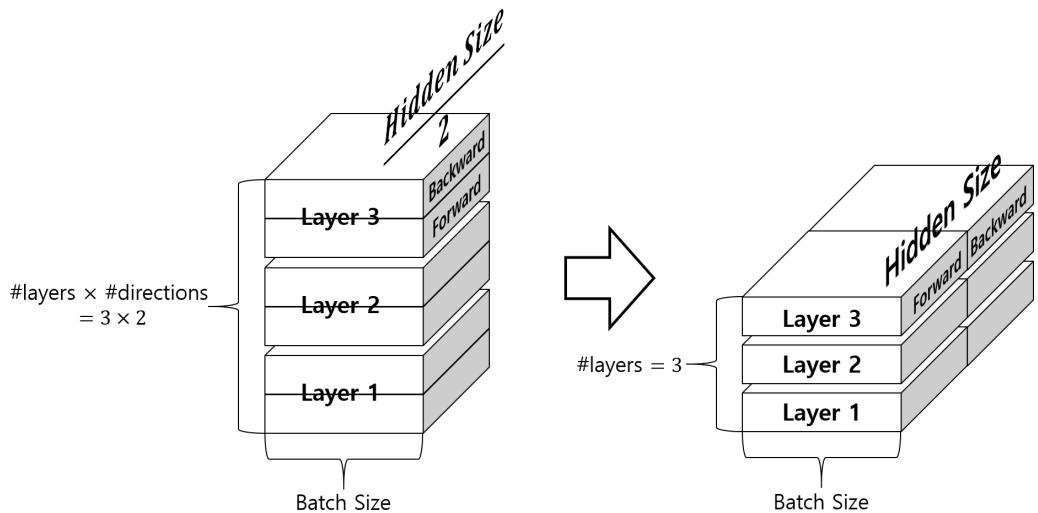


Figure 12:

```

def merge_encoder_hiddens(self, encoder_hiddens):
    new_hiddens = []
    new_cells = []

    hiddens, cells = encoder_hiddens

    # i-th and (i+1)-th layer is opposite direction.
    # Also, each direction of layer is half hidden size.
    # Therefore, we concatenate both directions to 1 hidden size layer.
    for i in range(0, hiddens.size(0), 2):
        new_hiddens += [torch.cat([hiddens[i], hiddens[i + 1]], dim = -1)]
        new_cells += [torch.cat([cells[i], cells[i + 1]], dim = -1)]

    new_hiddens, new_cells = torch.stack(new_hiddens), torch.stack(new_cells)

    return (new_hiddens, new_cells)

```

Forward

```

def forward(self, src, tgt):
    batch_size = tgt.size(0)

    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        # Based on the length information, gererate mask to prevent that short
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src

    if isinstance(tgt, tuple):
        tgt = tgt[0]

    # Get word embedding vectors for every time-step of input sentence.
    emb_src = self.emb_src(x)
    # |emb_src| = (batch_size, length, word_vec_dim)

    # The last hidden state of the encoder would be a initial hidden state of
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    # |h_src| = (batch_size, length, hidden_size)
    # |h_0_tgt| = (n_layers * 2, batch_size, hidden_size / 2)

    # Merge bidirectional to uni-directional
    # We need to convert size from (n_layers * 2, batch_size, hidden_size /
    # Thus, the converting operation will not working with just 'view' method
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hid
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hid
    # You can use 'merge_encoder_hiddens' method, instead of using above 3 lines
    # 'merge_encoder_hiddens' method works with non-parallel way.
    # h_0_tgt = self.merge_encoder_hiddens(h_0_tgt)

    # |h_src| = (batch_size, length, hidden_size)
    # |h_0_tgt| = (n_layers, batch_size, hidden_size)
    h_0_tgt = (h_0_tgt, c_0_tgt)

```

```

emb_tgt = self.emb_dec(tgt)
# |emb_tgt| = (batch_size, length, word_vec_dim)
h_tilde = []

h_t_tilde = None
decoder_hidden = h_0_tgt
# Run decoder until the end of the time-step.
for t in range(tgt.size(1)):
    # Teacher Forcing: take each input from training set, not from the
    # Because of Teacher Forcing, training procedure and inference proce
    # Of course, because of sequential running in decoder, this causes s
    emb_t = emb_tgt[:, t, :].unsqueeze(1)
    # |emb_t| = (batch_size, 1, word_vec_dim)
    # |h_t_tilde| = (batch_size, 1, hidden_size)

    decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decoder)
    # |decoder_output| = (batch_size, 1, hidden_size)
    # |decoder_hidden| = (n_layers, batch_size, hidden_size)

    context_vector = self.attn(h_src, decoder_output, mask)
    # |context_vector| = (batch_size, 1, hidden_size)

    h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_v
    # |h_t_tilde| = (batch_size, 1, hidden_size)

    h_tilde += [h_t_tilde]

h_tilde = torch.cat(h_tilde, dim = 1)
# |h_tilde| = (batch_size, length, hidden_size)

y_hat = self.generator(h_tilde)
# |y_hat| = (batch_size, length, output_size)

return y_hat

```

Seq2Seq Class

```

class Seq2Seq(nn.Module):

    def __init__(self, input_size, word_vec_dim, hidden_size, output_size, n_layers,
                 dropout_p):
        self.input_size = input_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        super(Seq2Seq, self).__init__()

        self.emb_src = nn.Embedding(input_size, word_vec_dim)
        self.emb_dec = nn.Embedding(output_size, word_vec_dim)

        self.encoder = Encoder(word_vec_dim, hidden_size, n_layers=n_layers, drop=
        self.decoder = Decoder(word_vec_dim, hidden_size, n_layers=n_layers, drop=
        self.attn = Attention(hidden_size)

        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.tanh = nn.Tanh()
        self.generator = Generator(hidden_size, output_size)

    def generate_mask(self, x, length):
        mask = []

        max_length = max(length)
        for l in length:
            if max_length - l > 0:
                # If the length is shorter than maximum length among samples,
                # set last few values to be 1s to remove attention weight.
                mask += [torch.cat([x.new_ones(1, 1).zero_(), x.new_ones(1, (max_
            else:
                # If the length of the sample equals to maximum length among sam
                # set every value in mask to be 0.
                mask += [x.new_ones(1, 1).zero_()]

        mask = torch.cat(mask, dim=0).byte()

```

```

    return mask

def merge_encoder_hiddens(self, encoder_hiddens):
    new_hiddens = []
    new_cells = []

    hiddens, cells = encoder_hiddens

    # i-th and (i+1)-th layer is opposite direction.
    # Also, each direction of layer is half hidden size.
    # Therefore, we concatenate both directions to 1 hidden size layer.
    for i in range(0, hiddens.size(0), 2):
        new_hiddens += [torch.cat([hiddens[i], hiddens[i + 1]], dim = -1)]
        new_cells += [torch.cat([cells[i], cells[i + 1]], dim = -1)]

    new_hiddens, new_cells = torch.stack(new_hiddens), torch.stack(new_cells)

    return (new_hiddens, new_cells)

def forward(self, src, tgt):
    batch_size = tgt.size(0)

    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        # Based on the length information, gererate mask to prevent that short
        # sentences are padded with zeros.
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src

    if isinstance(tgt, tuple):
        tgt = tgt[0]

    # Get word embedding vectors for every time-step of input sentence.
    emb_src = self.emb_src(x)

```

```

# |emb_src| = (batch_size, length, word_vec_dim)

# The last hidden state of the encoder would be a initial hidden state of
h_src, h_0_tgt = self.encoder((emb_src, x_length))
# |h_src| = (batch_size, length, hidden_size)
# |h_0_tgt| = (n_layers * 2, batch_size, hidden_size / 2)

# Merge bidirectional to uni-directional
# We need to convert size from (n_layers * 2, batch_size, hidden_size / 2)
# Thus, the converting operation will not work with just 'view' method
h_0_tgt, c_0_tgt = h_0_tgt
h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hidden_size)
c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hidden_size)
# You can use 'merge_encoder_hiddens' method, instead of using above 3 lines
# 'merge_encoder_hiddens' method works with non-parallel way.
# h_0_tgt = self.merge_encoder_hiddens(h_0_tgt)

# |h_src| = (batch_size, length, hidden_size)
# |h_0_tgt| = (n_layers, batch_size, hidden_size)
h_0_tgt = (h_0_tgt, c_0_tgt)

emb_tgt = self.emb_dec(tgt)
# |emb_tgt| = (batch_size, length, word_vec_dim)
h_tilde = []

h_t_tilde = None
decoder_hidden = h_0_tgt
# Run decoder until the end of the time-step.
for t in range(tgt.size(1)):
    # Teacher Forcing: take each input from training set, not from the previous output
    # Because of Teacher Forcing, training procedure and inference procedure are different
    # Of course, because of sequential running in decoder, this causes some problems
    emb_t = emb_tgt[:, t, :].unsqueeze(1)
    # |emb_t| = (batch_size, 1, word_vec_dim)
    # |h_t_tilde| = (batch_size, 1, hidden_size)

    decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decoder_hidden)
    # |decoder_output| = (batch_size, 1, hidden_size)

```

```

# |decoder_hidden| = (n_layers, batch_size, hidden_size)

context_vector = self.attn(h_src, decoder_output, mask)
# |context_vector| = (batch_size, 1, hidden_size)

h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_...
# |h_t_tilde| = (batch_size, 1, hidden_size)

h_tilde += [h_t_tilde]

h_tilde = torch.cat(h_tilde, dim = 1)
# |h_tilde| = (batch_size, length, hidden_size)

y_hat = self.generator(h_tilde)
# |y_hat| = (batch_size, length, output_size)

return y_hat

```

## Auto-regressive and Teacher Focusing

많은 분들이 여기까지 잘 따라왔다면 궁금증을 하나 가질 수 있습니다. 디코더의 입력으로 이전 time-step의 출력이 들어가는것이 훈련 때도 같은 것인가? 사실, 안타깝게도 seq2seq의 기본적인 훈련(training) 방식은 추론(inference)할 때의 방식과 상이합니다.

### Auto-regressive

Sequence-to-sequence의 훈련 방식과 추론 방식의 차이는 근본적으로 auto-regressive라는 속성 때문에 생겨납니다. Auto-regressive는 과거의 자신의 값을 참조하여 현재의 값을 추론(또는 예측)하는 특성을 가리키는 이름입니다. 이는 수식에서도 확인 할 수 있습니다. 예를 들어 아래는 전체적인 신경망 기계번역의 수식입니다.

$$\hat{Y} = \operatorname{argmax}_{Y \in \mathcal{Y}} P(Y|X) = \operatorname{argmax}_{Y \in \mathcal{Y}} \prod_{i=1}^n P(y_i|X, y_{<i})$$

or

$$y_i = \operatorname{argmax}_{y \in \mathcal{Y}} P(y|X, y_{<i})$$

where  $y_0 = BOS$ .

위와 같이 현재 time-step의 출력값  $y_t$ 는 인코더의 입력 문장(또는 시퀀스)  $X$ 와 이전 time-step까지의  $y_{<t}$ 를 조건부로 받아 결정 되기 때문에, 과거 자신의 값을 참조하게 되는 것입니다. 이러한 점은 과거에 잘못된 예측을 하게 되면 점점 시간이 지날수록 더 큰 잘못된 예측을 할 가능성을 야기하기도 합니다. 또한, 과거의 결과값에 따라 문장(또는 시퀀스)의 구성이 바뀔 뿐만 아니라, 예측 문장(시퀀스)의 길이 마저도 바뀌게 됩니다. 학습 과정에서는 이미 정답을 알고 있고, 현재 모델의 예측값과 정답과의 차이를 통해 학습하기 때문에, 우리는 auto-regressive 속성을 유지한 채 훈련을 할 수 없습니다.

$$\hat{y}_t = \operatorname{argmax}_{y \in \mathcal{Y}} P(y|X, \hat{y}_{<t}; \theta) \text{ where } X = \{x_1, x_2, \dots, x_n\} \text{ and } Y = \{y_0, y_1, \dots, y_{m+1}\}$$

## Teacher Forcing

따라서 우리는 Teacher Forcing이라고 불리는 방법을 사용하여 훈련 합니다. 훈련 할 때에 각 time-step 별 수식은 아래와 같습니다. 위와 같이 조건부에  $\hat{y}_{<t}$ 가 들어가는 것이 아닌,  $y_{<t}$ 가 들어가는 것이기 때문에, 훈련시에는 이전 time-step의 출력  $\hat{y}_{<t}$ 을 현재 time-step의 입력으로 넣어줄 수 없습니다. 만약 넣어주게 된다면, 현재 time-step의 디코더에게 잘못된 것을 가르쳐 주는 꼴이 될 것입니다.

$$\begin{aligned} \mathcal{L}(Y) &= - \sum_{i=1}^{m+1} \log P(y_i|X, y_{<i}; \theta) \\ \theta &\leftarrow \theta - \lambda \frac{1}{N} \sum_{i=1}^N \mathcal{L}(Y_i) \end{aligned}$$

또한, 실제 손실함수(loss function)을 계산하여 gradient descent를 수행할 때도, 해당 time-step의 argmax값인  $\hat{y}_i$ 의 확률을 사용하지 않고, softmax 레이어에서 정답에 해당하는  $y_i$ 의 인덱스(index)에 있는 로그(log)확률값을 사용 합니다.

중요한 점은 훈련(training)시에는 디코더의 입력으로 이전 time-step의 decoder의 출력값이 아닌, 실제  $Y$ 가 들어간다는 것입니다. 하지만, 추론(inference) 할 때에는 실제  $Y$ 를 모르기 때문에, 이전 time-step에서 계산되어 나온  $\hat{y}_{t-1}$ 를 decoder의 입력으로 사용합니다. 이 훈련 방법을 Teacher Forcing이라고 합니다.

이전에 언급하였듯이, 추론(inference) 할 때에는 auto-regressive 속성 때문에 과거 자기자신을 참조해야 합니다. 따라서 이전 time-step의 자기자신의 상태를 알기 위해서, 각 time-step 별로 순차적(sequential)으로 진행해야 합니다. 하지만 훈련(training) 할 때에는 입력값이 정해져 있으므로, 모든 time-step을 한번에 계산할 수 있습니다. 그러므로 decoder도 모든 time-step을 합쳐 수식을 정리할 수 있습니다.

$$H^{tgt} = \text{RNN}_{dec}(\text{emb}_{tgt}([BOS; Y[: -1]]), h_n^{src})$$

이런 auto-regressive 속성 및 teacher forcing 방법은 신경망 언어모델(NNLM)에도 똑같이 적용되는 문제입니다. 하지만 언어모델의 경우에는 perplexity는 문장의 확률과 직접적으로 연관이 있기 때문에, 큰 문제가 되지 않는 반면에 기계번역에서는 좀 더 큰 문제로 다가옵니다. 이에 대해서는 추후 다루도록 하겠습니다.

## Inference

### Overview

이제까지  $X$ 와  $Y$ 가 모두 주어진 훈련상황을 가정하였습니다만, 이제부터는  $X$ 만 주어진 상태에서  $\hat{Y}$ 을 예측하는 방법에 대해서 서술하겠습니다. 이러한 과정을 우리는 추론(inference) 또는 탐색(search)이라고 부릅니다. 우리가 기본적으로 이 방식을 탐색이라고 부르는 이유는 탐색 알고리즘(search algorithm)에 기반하기 때문입니다. 결국 우리가 원하는 것은 state로 이루어진 단어(word)들 사이에서 최고의 확률을 갖는 경로(path)를 찾는 것이기 때문입니다.

### Sampling

사실 먼저 우리가 생각할 수 있는 가장 정확한 방법은 각 time-step별  $\hat{y}_t$ 를 고를 때, 마지막 softmax 레이어에서의 확률 분포(probability distribution)대로 랜덤 샘플링을 하는 것입니다. 그리고 다음 time-step에서 그 선택( $\hat{y}_t$ )을 기반으로 다음

$\hat{y}_{t+1}$ 을 또 다시 샘플링하여 최종적으로  $EOS$ 가 나올 때 까지 샘플링을 반복하는 것입니다. 이렇게 하면 우리가 원하는  $P(Y|X)$ 에 가장 가까운 형태의 번역이 완성될 겁니다. 하지만, 이러한 방식은 같은 입력에 대해서 매번 다른 출력 결과물을 만들어낼 수 있습니다. 따라서 우리가 원하는 형태의 결과물이 아닙니다.

## Gready Search

우리는 자료구조, 알고리즘 수업에서 DFS, BFS, Dynamic Programming 등 수많은 탐색 기법에 대해 배웠습니다. 우리는 이중에서 Greedy 알고리즘을 기반으로 탐색을 구현합니다. 즉, softmax 레이어에서 가장 값이 큰 인덱스(index)를 뽑아 해당 time-step의  $\hat{y}_t$ 로 사용하게 되는 것 입니다.

## Code

아래의 코드는 샘플링 또는 greedy 탐색을 위한 코드입니다. 사실 인코더가 동작하는 부분까지는 완전히 똑같습니다. 다만, 이후 추론을 위한 부분은 기존 훈련 코드와 상이합니다. Teacher Forcing을 사용하였던 훈련 방식(실제 정답  $y_{t-1}$ 을  $t$  time-step의 입력으로 사용함)과 달리, 실제 이전 time-step의 출력을 현재 time-step의 입력으로 사용 합니다.

```

def search(self, src, is_greedy = True, max_length = 255):
    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        mask = self.generate_mask(x, x_length)
    else:
        x = src
        batch_size = x.size(0)

    emb_src = self.emb_src(x)
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.b)
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.b)
    h_0_tgt = (h_0_tgt, c_0_tgt)

```

```

# Fill a vector, which has 'batch_size' dimension, with BOS value.
y = x.new(batch_size, 1).zero_() + data_loader.BOS
is undone = x.new_ones(batch_size, 1).float()
decoder_hidden = h_0_tgt
h_t_tilde, y_hats, indice = None, [], []

# Repeat a loop while sum of 'is undone' flag is bigger than 0, or current
while is undone.sum() > 0 and len(indice) < max_length:
    # Unlike training procedure, take the last time-step's output during
    emb_t = self.emb_dec(y)
    # |emb_t| = (batch_size, 1, word_vec_dim)

    decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decoder_hidden)
    context_vector = self.attn(h_src, decoder_output, mask)
    h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_vector])))
    y_hat = self.generator(h_t_tilde)
    # |y_hat| = (batch_size, 1, output_size)
    y_hats += [y_hat]

    if is_greedy:
        y = torch.topk(y_hat, 1, dim = -1)[1].squeeze(-1)
    else:
        # Take a random sampling based on the multinoulli distribution.
        y = torch.multinomial(y_hat.exp().view(batch_size, -1), 1)
    # Put PAD if the sample is done.
    y = y.masked_fill_((1. - is undone).byte(), data_loader.PAD)
    is undone = is undone * torch.ne(y, data_loader.EOS).float()
    # |y| = (batch_size, 1)
    # |is undone| = (batch_size, 1)
    indice += [y]

    y_hats = torch.cat(y_hats, dim = 1)
    indice = torch.cat(indice, dim = -1)
    # |y_hat| = (batch_size, length, output_size)
    # |indice| = (batch_size, length)

return y_hats, indice

```

## Beam Search

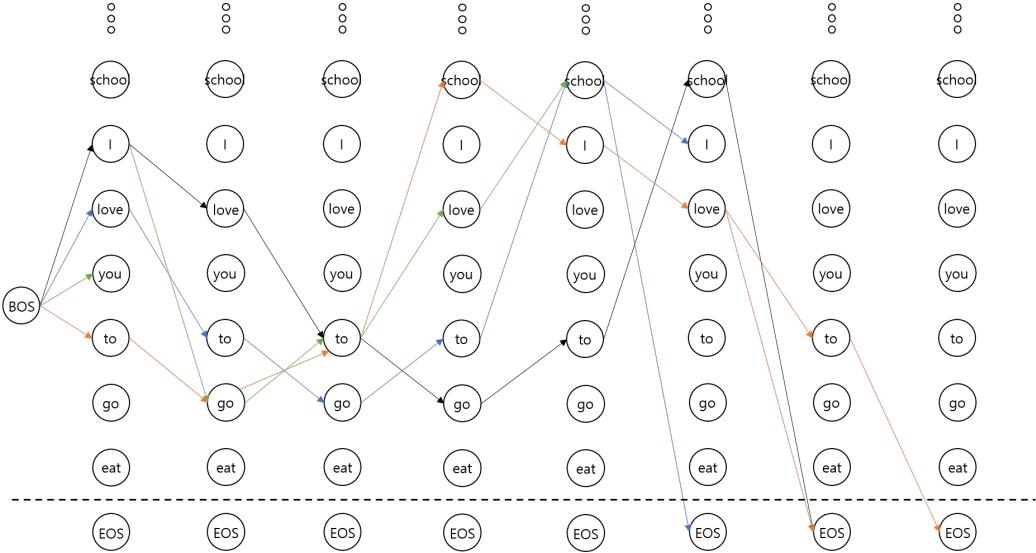


Figure 13:

하지만 우리는 자료구조, 알고리즘 수업에서 배웠다시피, greedy 알고리즘은 굉장히 쉽고 간편하지만, 최적의(optimal) 해를 보장하지 않습니다. 따라서 최적의 해에 가까워지기 위해서 우리는 약간의 트릭을 첨가합니다. Beam Size 만큼의 후보를 더 추적하는 것입니다.

현재 time-step에서 Top-k개를 뽑아서 (여기서 k는 beam size와 같습니다) 다음 time-step에 대해서 k번 추론을 수행합니다. 그리고 총  $k * |V|$  개의 softmax 결과 값 중에서 다시 top-k개를 뽑아 다음 time-step으로 넘깁니다. ( $|V|$ 는 Vocabulary size) 여기서 중요한 점은 두가지입니다.

$$\begin{aligned}\hat{y}_t^k &= \underset{k-\text{th}}{\operatorname{argmax}} \hat{Y}_t \\ \hat{Y}_t &= f_{\theta}(X, y_{<t}^1) \cup f_{\theta}(X, y_{<t}^2) \cup \dots \cup f_{\theta}(X, y_{<t}^k) \\ X &= \{x_1, x_2, \dots, x_n\}\end{aligned}$$

1. 누적 확률을 사용하여 top-k를 뽑습니다. 이때, 보통 로그 확률을 사용하므로 현재 time-step 까지의 로그확률에 대한 합을 추적하고 있어야 합니다.
2. top-k를 뽑을 때, 현재 time-step에 대해 k번 계산한 모든 결과물 중에서 뽑습니다.

Beam Search를 사용하면 좀 더 넓은 경로에 대해서 search를 수행하므로 당연히 좀 더 나은 성능을 보장합니다. 하지만, beam size만큼 번역을 더 수행해야 하기 때문에 속도에 저하가 있습니다. 다행히도 우리는 이 작업을 매 time-step마다 임시로 그때그때 mini-batch로 만들어 수행하기 때문에, 병렬로 계산할 수 있고, 이로 인해 약간의 속도 저하만 생기게 됩니다.

아래는 [Cho et al.2016]에서 주장한 beam search의 성능향상에 대한 실험 결과입니다. 샘플링 방법은 단순한 greedy 탐색보다 더 좋은 성능을 제공하지만, beam search가 가장 좋은 성능을 보여줍니다. 특기할 점은 기계번역 문제에서는 보통 beam size를 10 이하로 사용한다는 것입니다.

Strategy	# Chains	Valid Set		Test Set	
		NLL	BLEU	NLL	BLEU
Ancestral Sampling	50	22.98	15.64	26.25	16.76
Greedy Decoding	-	27.88	15.50	26.49	16.66
Beamsearch	5	20.18	17.03	22.81	18.56
Beamsearch	10	19.92	17.13	22.44	18.59

Figure 14: En-Cz: 12m training sentence pairs [Cho, arXiv 2016]

### How to implement

하나의 샘플에 대해서 추론을 수행하기 위한 beam search를 나타낸 그림입니다. 여기서 beam size는 3이 되는 것을 알 수 있습니다. 보통 beam search는 EOS가 beam size의 크기만큼 추론 될 때까지 수행 됩니다. 즉, 아래에서는 3개 이상의 EOS가 출현하면 beam search는 종료 됩니다. 아래의 그림에서는 마지막 직전의 time-step에서 1개의 EOS가 발생한 것을 볼 수 있고, EOS로부터 다시 이어지는 추론 대신에, 다른 2개의 추론 중에서 top-3를 다시 선택 하는 것을 볼 수 있습니다. Top-k개를 뽑을 때는 누적 확률이 높은 순서대로 뽑게 됩니다.

좀 더 자세하게 예를 들면,  $|V| = 30,000$ 이고  $k = 3$ 일 때, 한 time-step의 예측 결과의 갯수는  $|V| \times k = 90,000$ 가 됩니다. 이 9만개의 softmax 레이어 출력 유닛(vocabulary의 각 단어)들의 각각의 확률 값에 이전 time-step에서 선택된  $k$ 개의 누적 확률 값을 더해 최종 누적 확률 값 90,000개를 얻습니다. 여기서 총

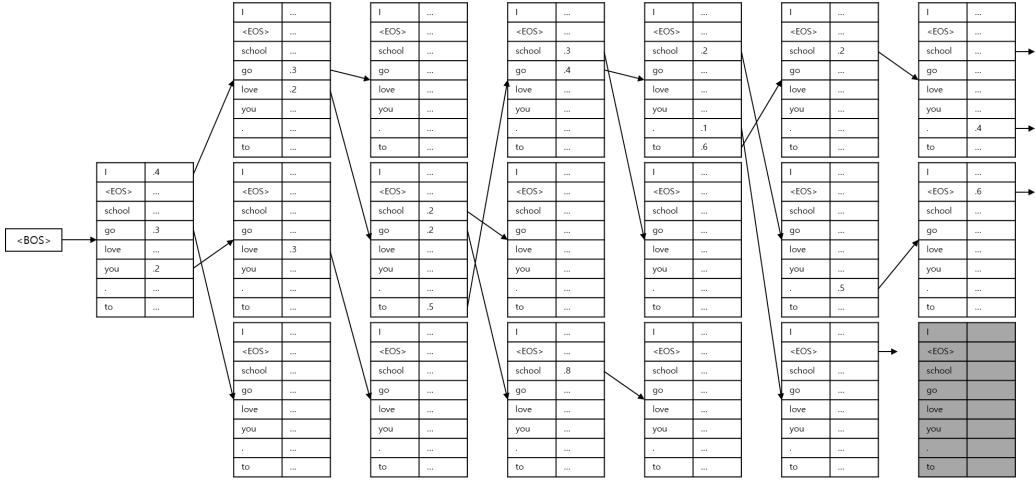


Figure 15: 1 sample beam search (beam size=3)

90,000개의 누적 확률 값 중에서 top-k개를 뽑아 다음 time-step의 예측을 위한 디코더의 입력으로 사용하게 됩니다.

### Length Penalty

위의 탐색 알고리즘을 직접 짜서 수행시켜 보면 한가지 문제점이 발견됩니다. 현재 time-step 까지의 확률을 모두 곱(로그 확률의 경우에는 합)하기 때문에 문장이 길어질수록 확률이 낮아진다는 점입니다. 따라서 짧은 문장일수록 더 높은 점수를 획득하는 경향이 있습니다. 우리는 이러한 현상을 방지하기 위해서 예측한 문장의 길이에 따른 페널티를 주어 탐색이 조기 종료되는 것을 막습니다. 수식은 아래와 같습니다. 불행히도 우리는 2개의 추가적인 hyper-parameter가 필요합니다.

$$\log \tilde{P}(\hat{Y}|X) = \log P(\hat{Y}|X) \times \text{penalty}$$

$$\text{penalty} = \frac{(1 + \text{length})^\alpha}{(1 + \beta)^\alpha}$$

where  $\beta$  is hyper-parameter of minimum length.

## Code

SingleBeamSearchSpace Class

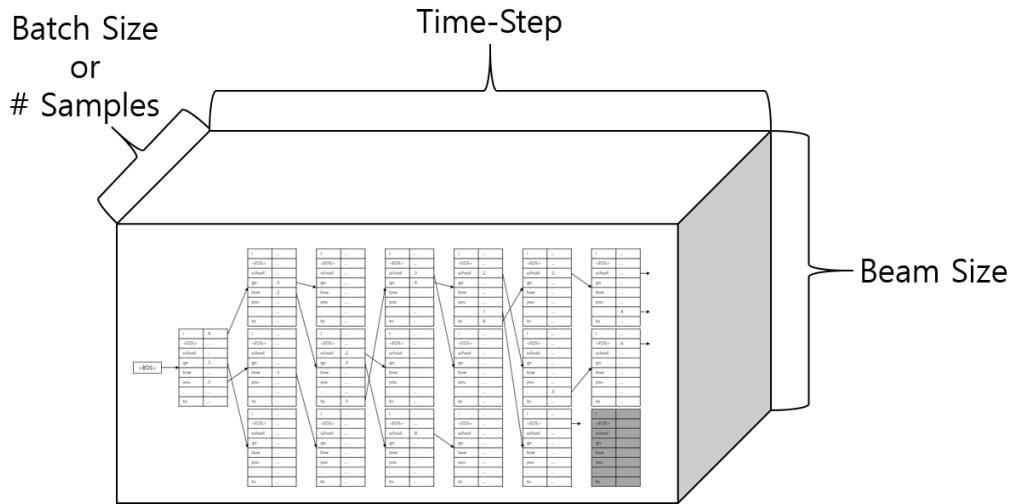


Figure 16:

Initialization

```
from operator import itemgetter

import torch
import torch.nn as nn

import data_loader

LENGTH_PENALTY = 1.2
MIN_LENGTH = 5

class SingleBeamSearchSpace():

    def __init__(self, hidden, h_t_tilde = None, beam_size = 5, max_length = 255)
        self.beam_size = beam_size
        self.max_length = max_length
```

```

super(SingleBeamSearchSpace, self).__init__()

# To put data to same device.
self.device = hidden[0].device
# Inferred word index for each time-step. For now, initialized with init
self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) +
# Index origin of current beam.
self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.device)
# Cumulative log-probability for each beam.
self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')]) * (beam_
# 1 if it is done else 0
self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

# We don't need to remember every time-step of hidden states: prev_hidde
# What we need is remember just last one.
# Future work: make this class to deal with any necessary information fo

# |hidden[0]| = (n_layers, 1, hidden_size)
self.prev_hidden = torch.cat([hidden[0]] * beam_size, dim = 1)
self.prev_cell = torch.cat([hidden[1]] * beam_size, dim = 1)
# |prev_hidden| = (n_layers, beam_size, hidden_size)
# |prev_cell| = (n_layers, beam_size, hidden_size)

# |h_t_tilde| = (batch_size = 1, 1, hidden_size)
self.prev_h_t_tilde = torch.cat([h_t_tilde] * beam_size, dim = 0) if h_t_t
# |prev_h_t_tilde| = (beam_size, 1, hidden_size)

self.current_time_step = 0
self.done_cnt = 0

def get_length_penalty(self, length, alpha = LENGTH_PENALTY, min_length = MIN_
# Calculate length-penalty, because shorter sentence usually have bigger
# Thus, we need to put penalty for shorter one.
p = (1 + length) ** alpha / (1 + min_length) ** alpha

return p

```

```

def is_done(self):
    # Return 1, if we had EOS more than 'beam_size'-times.
    if self.done_cnt >= self.beam_size:
        return 1
    return 0

def get_batch(self):
    y_hat = self.word_indice[-1].unsqueeze(-1)
    hidden = (self.prev_hidden, self.prev_cell)
    h_t_tilde = self.prev_h_t_tilde

    # |y_hat| = (beam_size, 1)
    # |hidden| = (n_layers, beam_size, hidden_size)
    # |h_t_tilde| = (beam_size, 1, hidden_size) or None
    return y_hat, hidden, h_t_tilde

def collect_result(self, y_hat, hidden, h_t_tilde):
    # |y_hat| = (beam_size, 1, output_size)
    # |hidden| = (n_layers, beam_size, hidden_size)
    # |h_t_tilde| = (beam_size, 1, hidden_size)
    output_size = y_hat.size(-1)

    self.current_time_step += 1

    # Calculate cumulative log-probability.
    # First, fill -inf value to last cumulative probability, if the beam is
    # Second, expand -inf filled cumulative probability to fit to 'y_hat'.
    # Third, add expanded cumulative probability to 'y_hat'
    cumulative_prob = y_hat + self.cumulative_probs[-1].masked_fill_(self.mask)
    # Now, we have new top log-probability and its index. We picked top index
    # Be aware that we picked top-k from whole batch through 'view(-1)'.
    top_log_prob, top_indice = torch.topk(cumulative_prob.view(-1), self.beam_size)
    # |top_log_prob| = (beam_size)
    # |top_indice| = (beam_size)

    self.word_indice += [top_indice.fmod(output_size)] # Because we picked from
    self.prev_beam_indice += [top_indice.div(output_size).long()] # Also, we

```

```

# Add results to history boards.
self.cumulative_probs += [top_log_prob]
self.masks += [torch.eq(self.word_indice[-1], data_loader.EOS)] # Set finished flag
self.done_cnt += self.masks[-1].float().sum() # Calculate a number of finished sentences

# Set hidden states for next time-step, using 'index_select' method.
self.prev_hidden = torch.index_select(hidden[0], dim = 1, index = self.prev_h_tilde)
self.prev_cell = torch.index_select(hidden[1], dim = 1, index = self.prev_h_tilde)
self.prev_h_t_tilde = torch.index_select(h_t_tilde, dim = 0, index = self.prev_h_tilde)

def get_n_best(self, n = 1):
    sentences = []
    probs = []
    founds = []

    for t in range(len(self.word_indice)): # for each time-step,
        for b in range(self.beam_size): # for each beam,
            if self.masks[t][b] == 1: # if we had EOS on this time-step and b-th beam
                # Take a record of penaltified log-probability.
                probs += [self.cumulative_probs[t][b] / self.get_length_penalty(t)]
                founds += [(t, b)]

    # Also, collect log-probability from last time-step, for the case of EOS
    for b in range(self.beam_size):
        if self.cumulative_probs[-1][b] != -float('inf'):
            if not (len(self.cumulative_probs) - 1, b) in founds:
                probs += [self.cumulative_probs[-1][b]]
                founds += [(t, b)]

    # Sort and take n-best.
    sorted_founds_with_probs = sorted(zip(founds, probs),
                                      key = itemgetter(1),
                                      reverse = True
                                      )[:n]

    probs = []

    for (end_index, b), prob in sorted_founds_with_probs:
        sentence = []

```

```

# Trace from the end.
for t in range(end_index, 0, -1):
    sentence = [self.word_indice[t][b]] + sentence
    b = self.prev_beam_indice[t][b]

sentences += [sentence]
probs += [prob]

return sentences, probs

```

Initialization

```

def __init__(self, hidden, h_t_tilde = None, beam_size = 5, max_length = 255):
    self.beam_size = beam_size
    self.max_length = max_length

    super(SingleBeamSearchSpace, self).__init__()

    # To put data to same device.
    self.device = hidden[0].device
    # Inferred word index for each time-step. For now, initialized with init
    self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) +
    # Index origin of current beam.
    self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.device)
    # Cumulative log-probability for each beam.
    self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')]) * (beam_
    # 1 if it is done else 0
    self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

    # We don't need to remember every time-step of hidden states: prev_hidden
    # What we need is remember just last one.
    # Future work: make this class to deal with any necessary information for

    # |hidden[0]| = (n_layers, 1, hidden_size)
    self.prev_hidden = torch.cat([hidden[0]] * beam_size, dim = 1)
    self.prev_cell = torch.cat([hidden[1]] * beam_size, dim = 1)

```

```

# |prev_hidden| = (n_layers, beam_size, hidden_size)
# |prev_cell| = (n_layers, beam_size, hidden_size)

# |h_t_tilde| = (batch_size = 1, 1, hidden_size)
self.prev_h_t_tilde = torch.cat([h_t_tilde] * beam_size, dim = 0) if h_t_tilde is not None
# |prev_h_t_tilde| = (beam_size, 1, hidden_size)

self.current_time_step = 0
self.done_cnt = 0

```

Implement Length Penalty

```

def get_length_penalty(self, length, alpha = LENGTH_PENALTY, min_length = MIN_LENGTH):
    # Calculate length-penalty, because shorter sentence usually have bigger p.
    # Thus, we need to put penalty for shorter one.
    p = (1 + length) ** alpha / (1 + min_length) ** alpha

    return p

```

Mark if is done

```

def is_done(self):
    # Return 1, if we had EOS more than 'beam_size'-times.
    if self.done_cnt >= self.beam_size:
        return 1
    return 0

```

Generate Fabricated Mini-batch

```

def get_batch(self):
    y_hat = self.word_indice[-1].unsqueeze(-1)
    hidden = (self.prev_hidden, self.prev_cell)
    h_t_tilde = self.prev_h_t_tilde

    # |y_hat| = (beam_size, 1)
    # |hidden| = (n_layers, beam_size, hidden_size)

```

```

# |h_t_tilde| = (beam_size, 1, hidden_size) or None
return y_hat, hidden, h_t_tilde

```

Collect the Result and Pick Top-K

```

def collect_result(self, y_hat, hidden, h_t_tilde):
    # |y_hat| = (beam_size, 1, output_size)
    # |hidden| = (n_layers, beam_size, hidden_size)
    # |h_t_tilde| = (beam_size, 1, hidden_size)
    output_size = y_hat.size(-1)

    self.current_time_step += 1

    # Calculate cumulative log-probability.
    # First, fill -inf value to last cumulative probability, if the beam is
    # Second, expand -inf filled cumulative probability to fit to 'y_hat'.
    # Third, add expanded cumulative probability to 'y_hat'
    cumulative_prob = y_hat + self.cumulative_probs[-1].masked_fill_(self.mask)
    # Now, we have new top log-probability and its index. We picked top inde
    # Be aware that we picked top-k from whole batch through 'view(-1)'.
    top_log_prob, top_indice = torch.topk(cumulative_prob.view(-1), self.beam_
    # |top_log_prob| = (beam_size)
    # |top_indice| = (beam_size)

    self.word_indice += [top_indice.fmod(output_size)] # Because we picked fr
    self.prev_beam_indice += [top_indice.div(output_size).long()] # Also, we

    # Add results to history boards.
    self.cumulative_probs += [top_log_prob]
    self.masks += [torch.eq(self.word_indice[-1], data_loader.EOS)] # Set fin
    self.done_cnt += self.masks[-1].float().sum() # Calculate a number of fin

    # Set hidden states for next time-step, using 'index_select' method.
    self.prev_hidden = torch.index_select(hidden[0], dim = 1, index = self.pre
    self.prev_cell = torch.index_select(hidden[1], dim = 1, index = self.prev_
    self.prev_h_t_tilde = torch.index_select(h_t_tilde, dim = 0, index = self

```

## Back-trace the History

```
def get_n_best(self, n = 1):
    sentences = []
    probs = []
    founds = []

    for t in range(len(self.word_indice)): # for each time-step,
        for b in range(self.beam_size): # for each beam,
            if self.masks[t][b] == 1: # if we had EOS on this time-step and b
                # Take a record of penaltified log-probability.
                probs += [self.cumulative_probs[t][b] / self.get_length_penalty()]
                founds += [(t, b)]

    # Also, collect log-probability from last time-step, for the case of EOS
    for b in range(self.beam_size):
        if self.cumulative_probs[-1][b] != -float('inf'):
            if not (len(self.cumulative_probs) - 1, b) in founds:
                probs += [self.cumulative_probs[-1][b]]
                founds += [(t, b)]

    # Sort and take n-best.
    sorted_founds_with_probs = sorted(zip(founds, probs),
                                      key = itemgetter(1),
                                      reverse = True
                                      )[:n]

    probs = []

    for (end_index, b), prob in sorted_founds_with_probs:
        sentence = []

        # Trace from the end.
        for t in range(end_index, 0, -1):
            sentence = [self.word_indice[t][b]] + sentence
            b = self.prev_beam_indice[t][b]

        sentences += [sentence]
        probs += [prob]
```

```
    return sentences, probs
```

### Mini-batch Parallelized Beam-Search

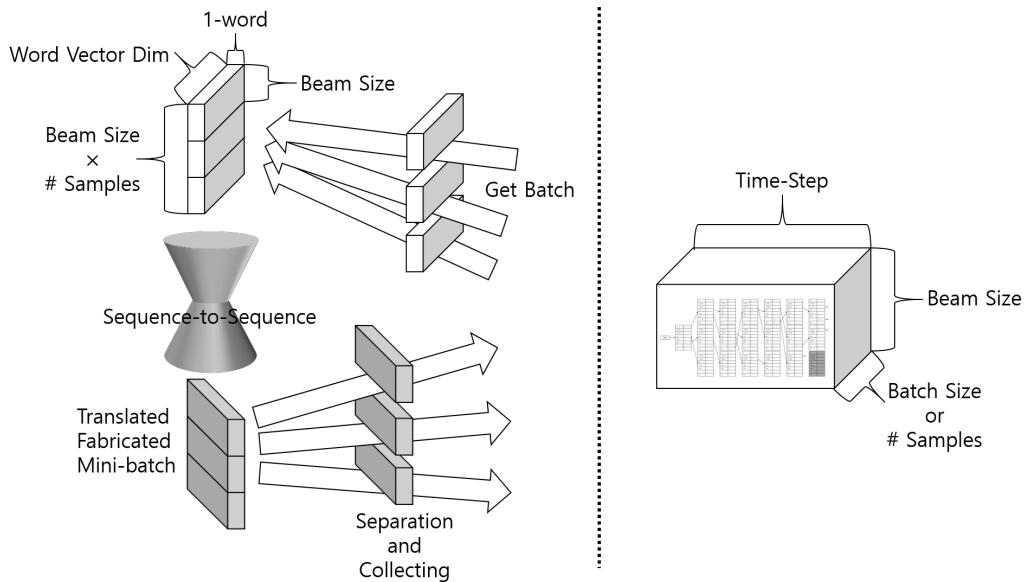


Figure 17:

```
def batch_beam_search(self, src, beam_size = 5, max_length = 255, n_best = 1)
    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src
        batch_size = x.size(0)

    emb_src = self.emb_src(x)
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
```

```

# |h_src| = (batch_size, length, hidden_size)
h_0_tgt, c_0_tgt = h_0_tgt
h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h)
c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h)
# |h_0_tgt| = (n_layers, batch_size, hidden_size)
h_0_tgt = (h_0_tgt, c_0_tgt)

# initialize 'SingleBeamSearchSpace' as many as batch_size
spaces = [SingleBeamSearchSpace((h_0_tgt[0][:, i, :].unsqueeze(1),
                                 h_0_tgt[1][:, i, :].unsqueeze(1)),
                                 None,
                                 beam_size,
                                 max_length = max_length
                               ) for i in range(batch_size)]
done_cnt = [space.is_done() for space in spaces]

length = 0
# Run loop while sum of 'done_cnt' is smaller than batch_size, or length
while sum(done_cnt) < batch_size and length <= max_length:
    # current_batch_size = sum(done_cnt) * beam_size

    # Initialize fabricated variables.
    # As far as batch-beam-search is running,
    # temporary batch-size for fabricated mini-batch is 'beam_size'-time
    fab_input, fab_hidden, fab_cell, fab_h_t_tilde = [], [], [], []
    fab_h_src, fab_mask = [], []

    # Build fabricated mini-batch in non-parallel way.
    # This may cause a bottle-neck.
    for i, space in enumerate(spaces):
        if space.is_done() == 0: # Batchfy only if the inference for the
            y_hat_, (hidden_, cell_), h_t_tilde_ = space.get_batch()

            fab_input += [y_hat_]
            fab_hidden += [hidden_]
            fab_cell += [cell_]
            if h_t_tilde_ is not None:
                fab_h_t_tilde += [h_t_tilde_]

```

```

    else:
        fab_h_t_tilde = None

        fab_h_src += [h_src[i, :, :]] * beam_size
        fab_mask += [mask[i, :]] * beam_size

    # Now, concatenate list of tensors.
    fab_input = torch.cat(fab_input, dim = 0)
    fab_hidden = torch.cat(fab_hidden, dim = 1)
    fab_cell = torch.cat(fab_cell, dim = 1)
    if fab_h_t_tilde is not None:
        fab_h_t_tilde = torch.cat(fab_h_t_tilde, dim = 0)
    fab_h_src = torch.stack(fab_h_src)
    fab_mask = torch.stack(fab_mask)
    # |fab_input| = (current_batch_size, 1)
    # |fab_hidden| = (n_layers, current_batch_size, hidden_size)
    # |fab_cell| = (n_layers, current_batch_size, hidden_size)
    # |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
    # |fab_h_src| = (current_batch_size, length, hidden_size)
    # |fab_mask| = (current_batch_size, length)

    emb_t = self.emb_dec(fab_input)
    # |emb_t| = (current_batch_size, 1, word_vec_dim)

    fab_decoder_output, (fab_hidden, fab_cell) = self.decoder(emb_t, fab_h_src)
    # |fab_decoder_output| = (current_batch_size, 1, hidden_size)
    context_vector = self.attn(fab_h_src, fab_decoder_output, fab_mask)
    # |context_vector| = (current_batch_size, 1, hidden_size)
    fab_h_t_tilde = self.tanh(self.concat(torch.cat([fab_decoder_output, context_vector], dim = 2)))
    # |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
    y_hat = self.generator(fab_h_t_tilde)
    # |y_hat| = (current_batch_size, 1, output_size)

    # separate the result for each sample.
    cnt = 0
    for space in spaces:
        if space.is_done() == 0:
            # Decide a range of each sample.

```

```

        from_index = cnt * beam_size
        to_index = from_index + beam_size

        # pick k-best results for each sample.
        space.collect_result(y_hat[from_index:to_index],
                             (fab_hidden[:, from_index:to_index],
                              fab_cell[:, from_index:to_index],
                              fab_h_t_tilde[from_index:to_index]))
        cnt += 1

    done_cnt = [space.is_done() for space in spaces]
    length += 1

    # pick n-best hypothesis.
    batch_sentences = []
    batch_probs = []

    # Collect the results.
    for i, space in enumerate(spaces):
        sentences, probs = space.get_n_best(n_best)

        batch_sentences += [sentences]
        batch_probs += [probs]

    return batch_sentences, batch_probs

```

## Evaluation

번역기의 성능을 평가하는 방법은 크게 두 가지로 나눌 수 있습니다. 정성적(implicit) 평가와 정량적(explicit) 평가 방식입니다.

## Implicit Evaluation

정성평가 방식은 보통 사람이 번역된 문장을 채점하는 형태로 이루어집니다. 사람은 선입견 등이 채점하는데 있어서 방해요소로 작용될 수 있기 때문에, 보통은 블라인드 테스트를 통해서 채점합니다. 이를 위해서 여러개의 다른 알고리즘을 통해 (또는 경쟁사의) 여러 번역결과를 누구의 것인지 밝히지 않은 채, 채점하여 우열을 가립니다. 이 방식은 가장 정확하다고 할 수 있지만, 자원과 시간이 많이 드는 단점이 있습니다.

## Explicit Evaluation

위의 단점 때문에, 보통은 자동화 된 정량평가를 주로 수행합니다. 두 평가를 모두 주기적으로 수행하되, 정성평가의 평가 주기를 좀 더 길게 가져가거나, 무언가 확실한 성능의 개선이 이루어졌을 때 수행하는 편입니다.

### Cross Entropy and Perplexity

신경망 기계번역도 기본적으로 매 time-step마다 최고 확률을 갖는 단어를 선택(분류) 하는 작업이기 때문에 기본적으로 분류(classification)작업에 속합니다. 따라서 Cross Entropy를 손실함수(loss function)로 사용합니다. 이전 섹션에서 언급하였듯이, 신경망 기계번역도 조건부 언어모델이기 때문에 perplexity를 통해 성능을 측정할 수 있습니다. 그러므로 이전 언어모델 챕터에서 다루었듯이, cross entropy loss 값에  $\exp$ 를 취하여 perplexity(PPL) 값을 얻을 수 있습니다.

### BLEU

위의 PPL은 우리가 사용하는 손실함수 cross entropy와 직결되어 바로 알 수 있는 간편함이 있지만, 사실 안타깝게도 실제 번역기의 성능과 완벽한 비례관계에 있다고 할 수는 없습니다. Cross entropy의 수식을 해석 해 보면, 각 time-step 별 실제 정답에 해당하는 단어의 확률만 채점하기 때문입니다.

원문	I	love	to	go	to	school	.
index	0	1	2	3	4	5	6
정답	나는	학교에	가는	것을	좋아한다	.	.
번역1	학교에	가는	것을	좋아한다	나는	.	.
번역2	나는	오락실에	가는	것을	싫어한다	.	.

예를 들어, 번역1은 cross entropy loss에 의하면 매우 높은 loss값을 가집니다. 하지만 번역2는 번역1에 비해 완전 틀린 번역이지만 loss가 훨씬 낮을 겁니다. 따라서 실제 번역문의 품질과 cross entropy 사이에는 (특히 teacher forcing 방식이 더해져) 괴리가 있습니다. 이러한 간극을 줄이기 위해 여러가지 방법들이 제시되었습니다 – METEOR, BLEU. 이번 섹션은 그 중 가장 널리 쓰이는 BLEU에 대해 짚고 넘어가겠습니다.

$$BLEU = \text{brevity-penalty} * \prod_{n=1}^N p_n^{w_n}$$

where brevity penalty =  $\min(1, \frac{|\text{prediction}|}{|\text{reference}|})$

and  $p_n$  is precision of  $n$ -gram and  $w_n$  is weight that  $w_n = \left(\frac{1}{2}\right)^n$ .

BLEU는 정답 문장과 예측 문장 사이에 일치하는 n-gram의 갯수의 비율의 기하평균에 따라 점수가 매겨집니다. brevity penalty는 예측 된 번역문이 정답 문장보다 짧을 경우 점수가 좋아지는 것을 방지하기 위함입니다. 보통 위 수식의 결과 값에 100을 곱하여 0에서 100사이의 숫자로 점수를 표현합니다. 실제 위의 예제 '번역1'에서 나타난 2-gram의 출현 빈도를 세어 간단하게 BLEU를 측정 하여 보겠습니다.

2-gram	count	hit count
BOS 학교에	1	0
학교에 가는	1	1
가는 것을	1	1
것을 좋아한다	1	1
좋아한다 나는	1	0
나는 .	1	0
. EOS	1	1
합계	7	4

따라서 2-gram의 BLEU 점수는  $4/7$ 이 됩니다. 이번에는 '번역2'에 대한 2-gram BLEU를 측정 해 보겠습니다.

2-gram	count	hit count
BOS 나는	1	1

2-gram	count	hit count
나는 오락실에	1	0
오락실에 가는	1	0
가는 것을	1	1
것을 싫어한다	1	0
싫어한다.	1	0
. EOS	1	1
합계	7	3

'번역2'의 2-gram BLEU는 3/7가 나왔습니다. 그러므로 (brevity penalty나 1-gram, 2-gram, 3-gram, 4-gram의 점수를 평균내지는 않았지만) 2-gram에 한해서  $4/7 > 3/7$ 이므로 '번역1'이 더 잘 번역되었다고 볼 수 있습니다. 즉, 위의 예제에서 '번역1'이 '번역2'보다 일치하는 n-gram이 더 많으므로 더 높은 BLEU 점수를 획득할 수 있습니다. 이와 같이 BLEU는 대체로 실제 정성평가의 결과와 일치하는 경향 있다고 여겨집니다.

위에서 볼 수 있듯이, 우리는 성능 평가 결과를 해석할 때 Perplexity(=Loss)는 낮을수록 좋고, BLEU는 높을수록 좋다고 합니다. 앞으로 설명할 알고리즘들의 성능을 평가할 때 참고 바랍니다. 실제 성능을 측정하기 위해서는 보통 SMT 프레임워크인 MOSES의 multi-bleu.perl을 주로 사용합니다.

## Full Source Code for Neural Machine Translation via RNN Sequence-to-Sequence

github repo url: <https://github.com/kh-kim/simple-nmt>

### train.py

```
import argparse, sys

import torch
import torch.nn as nn

from data_loader import DataLoader
```

```

import data_loader
from simple_nmt.seq2seq import Seq2Seq
import simple_nmt.trainer as trainer

def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True, help = 'Model file name to save. Add')
    p.add_argument('-train', required = True, help = 'Training set file name except')
    p.add_argument('-valid', required = True, help = 'Validation set file name except')
    p.add_argument('-lang', required = True, help = 'Set of extention represents T')
    p.add_argument('-gpu_id', type = int, default = -1, help = 'GPU ID to train. C')

    p.add_argument('-batch_size', type = int, default = 32, help = 'Mini batch size')
    p.add_argument('-n_epochs', type = int, default = 18, help = 'Number of epochs')
    p.add_argument('-print_every', type = int, default = 1000, help = 'Number of g')
    p.add_argument('-early_stop', type = int, default = -1, help = 'The training w')

    p.add_argument('-max_length', type = int, default = 80, help = 'Maximum length')
    p.add_argument('-dropout', type = float, default = .2, help = 'Dropout rate. I')
    p.add_argument('-word_vec_dim', type = int, default = 512, help = 'Word embedde')
    p.add_argument('-hidden_size', type = int, default = 768, help = 'Hidden size')
    p.add_argument('-n_layers', type = int, default = 4, help = 'Number of layers')

    p.add_argument('-max_grad_norm', type = float, default = 5., help = 'Threshold')
    p.add_argument('-adam', action = 'store_true', help = 'Use Adam instead of usi')
    p.add_argument('-lr', type = float, default = 1., help = 'Initial learning rat')
    p.add_argument('-min_lr', type = float, default = .000001, help = 'Minimum learn')
    p.add_argument('-lr_decay_start_at', type = int, default = 10, help = 'Start l')
    p.add_argument('-lr_slow_decay', action = 'store_true', help = 'Decay learning')
    p.add_argument('-lr_decay_rate', type = float, default = .5, help = 'Learning')

    config = p.parse_args()

    return config

def overwrite_config(config, prev_config):

```

```

# This method provides a compatibility for new or missing arguments.
for key in vars(prev_config).keys():
    if '-%s' % key not in sys.argv or key == 'model':
        if vars(config).get(key) is not None:
            vars(config)[key] = vars(prev_config)[key]
    else:
        # Missing argument
        print('WARNING!!! Argument "-%s" is not found in current arguments')
else:
    # Argument value is change from saved model.
    print('WARNING!!! Argument "-%s" is not loaded from saved model.\n')

return config

if __name__ == "__main__":
    config = define_argparser()

    import os.path
    # If the model exists, load model and configuration to continue the training
    if os.path.isfile(config.model):
        saved_data = torch.load(config.model)

        prev_config = saved_data['config']
        config = overwrite_config(config, prev_config)
        config.lr = saved_data['current_lr']
    else:
        saved_data = None

    # Load training and validation data set.
    loader = DataLoader(config.train,
                        config.valid,
                        (config.lang[:2], config.lang[-2:]),
                        batch_size = config.batch_size,
                        device = config.gpu_id,
                        max_length = config.max_length
                        )

```

```

input_size = len(loader.src.vocab) # Encoder's embedding layer input size
output_size = len(loader.tgt.vocab) # Decoder's embedding layer input size and output size
# Declare the model
model = Seq2Seq(input_size,
                  config.word_vec_dim, # Word embedding vector size
                  config.hidden_size, # LSTM's hidden vector size
                  output_size,
                  n_layers = config.n_layers, # number of layers in LSTM
                  dropout_p = config.dropout # dropout-rate in LSTM
                  )

# Default weight for loss equals to 1, but we don't need to get loss for PAD.
# Thus, set a weight for PAD to zero.
loss_weight = torch.ones(output_size)
loss_weight[data_loader.PAD] = 0.
# Instead of using Cross-Entropy loss, we can use Negative Log-Likelihood(NLL)
criterion = nn.NLLLoss(weight = loss_weight, size_average = False)

print(model)
print(criterion)

# Pass models to GPU device if it is necessary.
if config.gpu_id >= 0:
    model.cuda(config.gpu_id)
    criterion.cuda(config.gpu_id)

# If we have loaded model weight parameters, use that weights for declared model.
if saved_data is not None:
    model.load_state_dict(saved_data['model'])

# Start training. This function maybe equvalant to 'fit' function in Keras.
trainer.train_epoch(model,
                     criterion,
                     loader.train_iter,
                     loader.valid_iter,
                     config,
                     start_epoch = saved_data['epoch'] if saved_data is not None else 0,
                     others_to_save = {'src_vocab': loader.src.vocab, 'tgt_vocab': loader.tgt.vocab})

```

)

## translate.py

```
import argparse, sys
from operator import itemgetter

import torch
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from simple_nmt.seq2seq import Seq2Seq
import simple_nmt.trainer as trainer

def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True, help = 'Model file name to use')
    p.add_argument('-gpu_id', type = int, default = -1, help = 'GPU ID to use. -1'

    p.add_argument('-batch_size', type = int, default = 128, help = 'Mini batch size')
    p.add_argument('-max_length', type = int, default = 255, help = 'Maximum sequence length')
    p.add_argument('-n_best', type = int, default = 1, help = 'Number of best inference results')
    p.add_argument('-beam_size', type = int, default = 5, help = 'Beam size for beam search')

    config = p.parse_args()

    return config

def read_text():
    # This method gets sentences from standard input and tokenize those.
    lines = []

    for line in sys.stdin:
        if line.strip() != '':
            lines += [line.strip().split(' ')]
```

```

    return lines

def to_text(indice, vocab):
    # This method converts index to word to show the translation result.
    lines = []

    for i in range(len(indice)):
        line = []
        for j in range(len(indice[i])):
            index = indice[i][j]

            if index == data_loader.EOS:
                #line += ['<EOS>']
                break
            else:
                line += [vocab.itos[index]]

        line = ' '.join(line)
        lines += [line]

    return lines

if __name__ == '__main__':
    config = define_argparser()

    # Load saved model.
    saved_data = torch.load(config.model)

    # Load configuration setting in training.
    train_config = saved_data['config']
    # Load vocabularies from the model.
    src_vocab = saved_data['src_vocab']
    tgt_vocab = saved_data['tgt_vocab']

    # Initialize dataloader, but we don't need to read training & test corpus.
    # What we need is just load vocabularies from the previously trained model.
    loader = DataLoader()

```

```

loader.load_vocab(src_vocab, tgt_vocab)
input_size = len(loader.src.vocab)
output_size = len(loader.tgt.vocab)

# Declare sequence-to-sequence model.
model = Seq2Seq(input_size,
                 train_config.word_vec_dim,
                 train_config.hidden_size,
                 output_size,
                 n_layers = train_config.n_layers,
                 dropout_p = train_config.dropout
                )
model.load_state_dict(saved_data['model']) # Load weight parameters from the
model.eval() # We need to turn-on the evaluation mode, which turns off all d

# We don't need to draw a computation graph, because we will have only infer
torch.set_grad_enabled(False)

# Put models to device if it is necessary.
if config.gpu_id >= 0:
    model.cuda(config.gpu_id)

# Get sentences from standard input.
lines = read_text()

with torch.no_grad(): # Also, declare again to prevent to get gradients.
    while len(lines) > 0:
        # Since packed_sequence must be sorted by decreasing order of length
        # sorting by length in mini-batch should be restored by original ord
        # Therefore, we need to memorize the original index of the sentence.
        sorted_lines = lines[:config.batch_size]
        lines = lines[config.batch_size:]
        lengths = [len(_) for _ in sorted_lines]
        orders = [i for i in range(len(sorted_lines))]

        sorted_tuples = sorted(zip(sorted_lines, lengths, orders), key = itemg
sorted_lines = [sorted_tuples[i][0] for i in range(len(sorted_tuples))]
lengths = [sorted_tuples[i][1] for i in range(len(sorted_tuples))]
```

```

orders = [sorted_tuples[i][2] for i in range(len(sorted_tuples))]

# Converts string to list of index.
x = loader.src.numericalize(loader.src.pad(sorted_lines), device = 'cu

if config.beam_size == 1:
    # Take inference for non-parallel beam-search.
    y_hat, indice = model.search(x)
    output = to_text(indice, loader.tgt.vocab)

    sorted_tuples = sorted(zip(output, orders), key = itemgetter(1))
    output = [sorted_tuples[i][0] for i in range(len(sorted_tuples))]

    sys.stdout.write('\n'.join(output) + '\n')
else:
    # Take mini-batch parallelized beam search.
    batch_indice, _ = model.batch_beam_search(x,
                                                beam_size = config.beam_size,
                                                max_length = config.max_length,
                                                n_best = config.n_best)

    # Restore the original orders.
    output = []
    for i in range(len(batch_indice)):
        output += [to_text(batch_indice[i], loader.tgt.vocab)]
    sorted_tuples = sorted(zip(output, orders), key = itemgetter(1))
    output = [sorted_tuples[i][0] for i in range(len(sorted_tuples))]

    for i in range(len(output)):
        sys.stdout.write('\n'.join(output[i]) + '\n')

```

## data\_loader.py

```

import os
from torchtext import data, datasets

```

```

PAD = 1
BOS = 2
EOS = 3

class DataLoader():

    def __init__(self, train_fn = None,
                 valid_fn = None,
                 exts = None,
                 batch_size = 64,
                 device = 'cpu',
                 max_vocab = 99999999,
                 max_length = 255,
                 fix_length = None,
                 use_bos = True,
                 use_eos = True,
                 shuffle = True
                 ):

        super(DataLoader, self).__init__()

        self.src = data.Field(sequential = True,
                             use_vocab = True,
                             batch_first = True,
                             include_lengths = True,
                             fix_length = fix_length,
                             init_token = None,
                             eos_token = None
                             )

        self.tgt = data.Field(sequential = True,
                             use_vocab = True,
                             batch_first = True,
                             include_lengths = True,
                             fix_length = fix_length,
                             init_token = '<BOS>' if use_bos else None,
                             eos_token = '<EOS>' if use_eos else None
                             )

```

```

if train_fn is not None and valid_fn is not None and exts is not None:
    train = TranslationDataset(path = train_fn, exts = exts,
                               fields = [('src', self.src), ('tgt', self.tgt),
                                          max_length = max_length
                                         ])
    valid = TranslationDataset(path = valid_fn, exts = exts,
                               fields = [('src', self.src), ('tgt', self.tgt),
                                          max_length = max_length
                                         ])

    self.train_iter = data.BucketIterator(train,
                                         batch_size = batch_size,
                                         device = 'cuda:%d' % device if device is not None else None,
                                         shuffle = shuffle,
                                         sort_key=lambda x: len(x.tgt),
                                         sort_within_batch = True
                                         )
    self.valid_iter = data.BucketIterator(valid,
                                         batch_size = batch_size,
                                         device = 'cuda:%d' % device if device is not None else None,
                                         shuffle = False,
                                         sort_key=lambda x: len(x.tgt),
                                         sort_within_batch = True
                                         )

    self.src.build_vocab(train, max_size = max_vocab)
    self.tgt.build_vocab(train, max_size = max_vocab)

def load_vocab(self, src_vocab, tgt_vocab):
    self.src.vocab = src_vocab
    self.tgt.vocab = tgt_vocab

class TranslationDataset(data.Dataset):
    """Defines a dataset for machine translation."""

    @staticmethod
    def sort_key(ex):

```

```

    return data.interleave_keys(len(ex.src), len(ex.trg))

def __init__(self, path, exts, fields, max_length=None, **kwargs):
    """Create a TranslationDataset given paths and fields.

    Arguments:
        path: Common prefix of paths to the data files for both languages.
        exts: A tuple containing the extension to path for each language.
        fields: A tuple containing the fields that will be used for data
            in each language.
    Remaining keyword arguments: Passed to the constructor of
        data.Dataset.

    """
    if not isinstance(fields[0], (tuple, list)):
        fields = [('src', fields[0]), ('trg', fields[1])]

    if not path.endswith('.'):
        path += '.'

    src_path, trg_path = tuple(os.path.expanduser(path + x) for x in exts)

    examples = []
    with open(src_path) as src_file, open(trg_path) as trg_file:
        for src_line, trg_line in zip(src_file, trg_file):
            src_line, trg_line = src_line.strip(), trg_line.strip()
            if max_length and max_length < max(len(src_line.split()), len(trg_line.split())):
                continue
            if src_line != '' and trg_line != '':
                examples.append(data.Example.fromlist(
                    [src_line, trg_line], fields))

    super(TranslationDataset, self).__init__(examples, fields, **kwargs)

if __name__ == '__main__':
    import sys
    loader = DataLoader(sys.argv[1], sys.argv[2], (sys.argv[3], sys.argv[4]), batch_size=128, shuffle=True)
    print(len(loader.src.vocab))

```

```

print(len(loader.tgt.vocab))

for batch_index, batch in enumerate(loader.train_iter):
    print(batch.src)
    print(batch.tgt)

    if batch_index > 1:
        break

```

### simple\_nmt/seq2seq.py

```

import numpy as np
import torch
import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence as pack
from torch.nn.utils.rnn import pad_packed_sequence as unpack

import data_loader
from simple_nmt.search import SingleBeamSearchSpace

class Attention(nn.Module):

    def __init__(self, hidden_size):
        super(Attention, self).__init__()

        self.linear = nn.Linear(hidden_size, hidden_size, bias = False)
        self.softmax = nn.Softmax(dim = -1)

    def forward(self, h_src, h_t_tgt, mask = None):
        # |h_src| = (batch_size, length, hidden_size)
        # |h_t_tgt| = (batch_size, 1, hidden_size)
        # |mask| = (batch_size, length)

        query = self.linear(h_t_tgt.squeeze(1)).unsqueeze(-1)
        # |query| = (batch_size, hidden_size, 1)

        weight = torch.bmm(h_src, query).squeeze(-1)

```

```

# |weight| = (batch_size, length)
if mask is not None:
    # Set each weight as -inf, if the mask value equals to 1.
    # Since the softmax operation makes -inf to 0, masked weights would
    # Thus, if the sample is shorter than other samples in mini-batch, t
    weight.masked_fill_(mask, -float('inf'))
weight = self.softmax(weight)

context_vector = torch.bmm(weight.unsqueeze(1), h_src)
# |context_vector| = (batch_size, 1, hidden_size)

return context_vector

class Encoder(nn.Module):

    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
        super(Encoder, self).__init__()

        # Be aware of value of 'batch_first' parameter.
        # Also, its hidden_size is half of original hidden_size, because it is b
        self.rnn = nn.LSTM(word_vec_dim, int(hidden_size / 2), num_layers = n_layer

    def forward(self, emb):
        # |emb| = (batch_size, length, word_vec_dim)

        if isinstance(emb, tuple):
            x, lengths = emb
            x = pack(x, lengths.tolist(), batch_first = True)

        # Below is how pack_padded_sequence works.
        # As you can see, PackedSequence object has information about mini-b
        #
        # a = [torch.tensor([1,2,3]), torch.tensor([3,4])]
        # b = torch.nn.utils.rnn.pad_sequence(a, batch_first=True)
        # >>>
        # tensor([[ 1,  2,  3],
        #         [ 3,  4,  0]])
        # torch.nn.utils.rnn.pack_padded_sequence(b, batch_first=True, lengt

```

```

# >>>PackedSequence(data=tensor([ 1,  3,  2,  4,  3]), batch_sizes=[1, 3, 2, 2, 1])
else:
    x = emb

y, h = self.rnn(x)
# |y| = (batch_size, length, hidden_size)
# |h[0]| = (num_layers * 2, batch_size, hidden_size / 2)

if isinstance(emb, tuple):
    y, _ = unpack(y, batch_first = True)

return y, h

class Decoder(nn.Module):

def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
    super(Decoder, self).__init__()

    # Be aware of value of 'batch_first' parameter and 'bidirectional' parameter
    self.rnn = nn.LSTM(word_vec_dim + hidden_size, hidden_size, num_layers = n_layers,
                       batch_first = True, bidirectional = False)

def forward(self, emb_t, h_t_1_tilde, h_t_1):
    # |emb_t| = (batch_size, 1, word_vec_dim)
    # |h_t_1_tilde| = (batch_size, 1, hidden_size)
    # |h_t_1[0]| = (n_layers, batch_size, hidden_size)
    batch_size = emb_t.size(0)
    hidden_size = h_t_1[0].size(-1)

    if h_t_1_tilde is None:
        # If this is the first time-step,
        h_t_1_tilde = emb_t.new(batch_size, 1, hidden_size).zero_()

    # Input feeding trick.
    x = torch.cat([emb_t, h_t_1_tilde], dim = -1)

    # Unlike encoder, decoder must take an input for sequentially.
    y, h = self.rnn(x, h_t_1)

```

```

    return y, h

class Generator(nn.Module):

    def __init__(self, hidden_size, output_size):
        super(Generator, self).__init__()

        self.output = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim = -1)

    def forward(self, x):
        # |x| = (batch_size, length, hidden_size)

        y = self.softmax(self.output(x))
        # |y| = (batch_size, length, output_size)

        # Return log-probability instead of just probability.
        return y

class Seq2Seq(nn.Module):

    def __init__(self, input_size, word_vec_dim, hidden_size, output_size, n_layers,
                 dropout_p):
        self.input_size = input_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        super(Seq2Seq, self).__init__()

        self.emb_src = nn.Embedding(input_size, word_vec_dim)
        self.emb_dec = nn.Embedding(output_size, word_vec_dim)

        self.encoder = Encoder(word_vec_dim, hidden_size, n_layers = n_layers, drop =
        self.decoder = Decoder(word_vec_dim, hidden_size, n_layers = n_layers, drop =
        self.attn = Attention(hidden_size)

```

```

        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.tanh = nn.Tanh()
        self.generator = Generator(hidden_size, output_size)

    def generate_mask(self, x, length):
        mask = []

        max_length = max(length)
        for l in length:
            if max_length - l > 0:
                # If the length is shorter than maximum length among samples,
                # set last few values to be 1s to remove attention weight.
                mask += [torch.cat([x.new_ones(1, l).zero_(), x.new_ones(1, (max_
else:
                # If the length of the sample equals to maximum length among sam
                # set every value in mask to be 0.
                mask += [x.new_ones(1, l).zero_()]

        mask = torch.cat(mask, dim = 0).byte()

    return mask

def merge_encoder_hiddens(self, encoder_hiddens):
    new_hiddens = []
    new_cells = []

    hiddens, cells = encoder_hiddens

    # i-th and (i+1)-th layer is opposite direction.
    # Also, each direction of layer is half hidden size.
    # Therefore, we concatenate both directions to 1 hidden size layer.
    for i in range(0, hiddens.size(0), 2):
        new_hiddens += [torch.cat([hiddens[i], hiddens[i + 1]], dim = -1)]
        new_cells += [torch.cat([cells[i], cells[i + 1]], dim = -1)]

    new_hiddens, new_cells = torch.stack(new_hiddens), torch.stack(new_cells)

    return (new_hiddens, new_cells)

```

```

def forward(self, src, tgt):
    batch_size = tgt.size(0)

    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        # Based on the length information, gererate mask to prevent that short
        # sentences are padded with zeros.
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src

    if isinstance(tgt, tuple):
        tgt = tgt[0]

    # Get word embedding vectors for every time-step of input sentence.
    emb_src = self.emb_src(x)
    # |emb_src| = (batch_size, length, word_vec_dim)

    # The last hidden state of the encoder would be a initial hidden state of
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    # |h_src| = (batch_size, length, hidden_size)
    # |h_0_tgt| = (n_layers * 2, batch_size, hidden_size / 2)

    # Merge bidirectional to uni-directional
    # We need to convert size from (n_layers * 2, batch_size, hidden_size /
    # Thus, the converting operation will not working with just 'view' method.
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hid
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hid
    # You can use 'merge_encoder_hiddens' method, instead of using above 3 lines.
    # 'merge_encoder_hiddens' method works with non-parallel way.
    # h_0_tgt = self.merge_encoder_hiddens(h_0_tgt)

    # |h_src| = (batch_size, length, hidden_size)
    # |h_0_tgt| = (n_layers, batch_size, hidden_size)

```

```

    h_0_tgt = (h_0_tgt, c_0_tgt)

    emb_tgt = self.emb_dec(tgt)
    # |emb_tgt| = (batch_size, length, word_vec_dim)
    h_tilde = []

    h_t_tilde = None
    decoder_hidden = h_0_tgt
    # Run decoder until the end of the time-step.
    for t in range(tgt.size(1)):
        # Teacher Forcing: take each input from training set, not from the
        # Because of Teacher Forcing, training procedure and inference procedure
        # Of course, because of sequential running in decoder, this causes some
        emb_t = emb_tgt[:, t, :].unsqueeze(1)
        # |emb_t| = (batch_size, 1, word_vec_dim)
        # |h_t_tilde| = (batch_size, 1, hidden_size)

        decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decoder_hidden)
        # |decoder_output| = (batch_size, 1, hidden_size)
        # |decoder_hidden| = (n_layers, batch_size, hidden_size)

        context_vector = self.attn(h_src, decoder_output, mask)
        # |context_vector| = (batch_size, 1, hidden_size)

        h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_vector])))
        # |h_t_tilde| = (batch_size, 1, hidden_size)

        h_tilde += [h_t_tilde]

    h_tilde = torch.cat(h_tilde, dim = 1)
    # |h_tilde| = (batch_size, length, hidden_size)

    y_hat = self.generator(h_tilde)
    # |y_hat| = (batch_size, length, output_size)

    return y_hat

def search(self, src, is_greedy = True, max_length = 255):

```

```

mask = None
x_length = None
if isinstance(src, tuple):
    x, x_length = src
    mask = self.generate_mask(x, x_length)
else:
    x = src
batch_size = x.size(0)

emb_src = self.emb_src(x)
h_src, h_0_tgt = self.encoder((emb_src, x_length))
h_0_tgt, c_0_tgt = h_0_tgt
h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h_size)
c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h_size)
h_0_tgt = (h_0_tgt, c_0_tgt)

# Fill a vector, which has 'batch_size' dimension, with BOS value.
y = x.new(batch_size, 1).zero_() + data_loader.BOS
is undone = x.new_ones(batch_size, 1).float()
decoder_hidden = h_0_tgt
h_t_tilde, y_hats, indice = None, [], []

# Repeat a loop while sum of 'is undone' flag is bigger than 0, or current length is bigger than max_length.
while is undone.sum() > 0 and len(indice) < max_length:
    # Unlike training procedure, take the last time-step's output during decoding.
    emb_t = self.emb_dec(y)
    # |emb_t| = (batch_size, 1, word_vec_dim)

    decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decoder_hidden)
    context_vector = self.attn(h_src, decoder_output, mask)
    h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_vector])))
    y_hat = self.generator(h_t_tilde)
    # |y_hat| = (batch_size, 1, output_size)
    y_hats += [y_hat]

    if is_greedy:
        y = torch.topk(y_hat, 1, dim = -1)[1].squeeze(-1)
    else:

```

```

        # Take a random sampling based on the multinoulli distribution.
        y = torch.multinomial(y_hat.exp().view(batch_size, -1), 1)
        # Put PAD if the sample is done.
        y = y.masked_fill_((1. - is undone).byte(), data_loader.PAD)
        is undone = is undone * torch.ne(y, data_loader.EOS).float()
        # |y| = (batch_size, 1)
        # |is undone| = (batch_size, 1)
        indice += [y]

        y_hats = torch.cat(y_hats, dim = 1)
        indice = torch.cat(indice, dim = -1)
        # |y_hat| = (batch_size, length, output_size)
        # |indice| = (batch_size, length)

    return y_hats, indice

def batch_beam_search(self, src, beam_size = 5, max_length = 255, n_best = 1):
    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src
    batch_size = x.size(0)

    emb_src = self.emb_src(x)
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    # |h_src| = (batch_size, length, hidden_size)
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hid_size)
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.hid_size)
    # |h_0_tgt| = (n_layers, batch_size, hidden_size)
    h_0_tgt = (h_0_tgt, c_0_tgt)

    # initialize 'SingleBeamSearchSpace' as many as batch_size
    spaces = [SingleBeamSearchSpace((h_0_tgt[0][:, i, :]).unsqueeze(1),

```

```

        h_0_tgt[1][:, i, :].unsqueeze(1)),
        None,
        beam_size,
        max_length = max_length
    ) for i in range(batch_size)]
done_cnt = [space.is_done() for space in spaces]

length = 0
# Run loop while sum of 'done_cnt' is smaller than batch_size, or length
while sum(done_cnt) < batch_size and length <= max_length:
    # current_batch_size = sum(done_cnt) * beam_size

    # Initialize fabricated variables.
    # As far as batch-beam-search is running,
    # temporary batch-size for fabricated mini-batch is 'beam_size'-time
    fab_input, fab_hidden, fab_cell, fab_h_t_tilde = [], [], [], []
    fab_h_src, fab_mask = [], []

    # Build fabricated mini-batch in non-parallel way.
    # This may cause a bottle-neck.
    for i, space in enumerate(spaces):
        if space.is_done() == 0: # Batchfy only if the inference for the
            y_hat_, (hidden_, cell_), h_t_tilde_ = space.get_batch()

            fab_input += [y_hat_]
            fab_hidden += [hidden_]
            fab_cell += [cell_]
            if h_t_tilde_ is not None:
                fab_h_t_tilde += [h_t_tilde_]
            else:
                fab_h_t_tilde = None

            fab_h_src += [h_src[i, :, :]] * beam_size
            fab_mask += [mask[i, :]] * beam_size

    # Now, concatenate list of tensors.
    fab_input = torch.cat(fab_input, dim = 0)
    fab_hidden = torch.cat(fab_hidden, dim = 1)

```

```

fab_cell = torch.cat(fab_cell, dim = 1)
if fab_h_t_tilde is not None:
    fab_h_t_tilde = torch.cat(fab_h_t_tilde, dim = 0)
fab_h_src = torch.stack(fab_h_src)
fab_mask = torch.stack(fab_mask)
# |fab_input| = (current_batch_size, 1)
# |fab_hidden| = (n_layers, current_batch_size, hidden_size)
# |fab_cell| = (n_layers, current_batch_size, hidden_size)
# |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
# |fab_h_src| = (current_batch_size, length, hidden_size)
# |fab_mask| = (current_batch_size, length)

emb_t = self.emb_dec(fab_input)
# |emb_t| = (current_batch_size, 1, word_vec_dim)

fab_decoder_output, (fab_hidden, fab_cell) = self.decoder(emb_t, fab_h_src)
# |fab_decoder_output| = (current_batch_size, 1, hidden_size)
context_vector = self.attn(fab_h_src, fab_decoder_output, fab_mask)
# |context_vector| = (current_batch_size, 1, hidden_size)
fab_h_t_tilde = self.tanh(self.concat(torch.cat([fab_decoder_output,
# |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
y_hat = self.generator(fab_h_t_tilde)
# |y_hat| = (current_batch_size, 1, output_size)

# separate the result for each sample.
cnt = 0
for space in spaces:
    if space.is_done() == 0:
        # Decide a range of each sample.
        from_index = cnt * beam_size
        to_index = from_index + beam_size

        # pick k-best results for each sample.
        space.collect_result(y_hat[from_index:to_index],
                            (fab_hidden[:, from_index:to_index],
                             fab_cell[:, from_index:to_index],
                             fab_h_t_tilde[from_index:to_index])
                            )

```

```

        cnt += 1

    done_cnt = [space.is_done() for space in spaces]
    length += 1

    # pick n-best hypothesis.
    batch_sentences = []
    batch_probs = []

    # Collect the results.
    for i, space in enumerate(spaces):
        sentences, probs = space.get_n_best(n_best)

        batch_sentences += [sentences]
        batch_probs += [probs]

    return batch_sentences, batch_probs

```

## simple\_nmt/trainer.py

```

import time
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.utils as torch_utils

import utils

def get_loss(y, y_hat, criterion, do_backward = True):
    # |y| = (batch_size, length)
    # |y_hat| = (batch_size, length, output_size)
    batch_size = y.size(0)

    loss = criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguous())
    if do_backward:

```

```

        loss.div(batch_size).backward()

    return loss

def train_epoch(model, criterion, train_iter, valid_iter, config, start_epoch = 1
    current_lr = config.lr

    lowest_valid_loss = np.inf
    no_improve_cnt = 0

    for epoch in range(start_epoch, config.n_epochs + 1):
        if config.adam:
            optimizer = optim.Adam(model.parameters(), lr = current_lr)
        else:
            optimizer = optim.SGD(model.parameters(), lr = current_lr)
        print("current learning rate: %f" % current_lr)
        print(optimizer)

        sample_cnt = 0
        total_loss, total_word_count, total_parameter_norm, total_grad_norm = 0, 0
        start_time = time.time()
        train_loss = np.inf

        for batch_index, batch in enumerate(train_iter):
            # You have to reset the gradients of all model parameters before to
            optimizer.zero_grad()

            current_batch_word_cnt = torch.sum(batch.tgt[1])
            x = batch.src
            # Raw target variable has both BOS and EOS token.
            # The output of sequence-to-sequence does not have BOS token.
            # Thus, remove BOS token for reference.
            y = batch.tgt[0][:, 1:]
            # |x| = (batch_size, length)
            # |y| = (batch_size, length)

            # Take feed-forward
            # Similar as before, the input of decoder does not have EOS token.

```



```

# In orther to avoid gradient exploding, we apply gradient clipping.
torch_utils.clip_grad_norm_(model.parameters(), config.max_grad_norm)
# Take a step of gradient descent.
optimizer.step()

sample_cnt += batch.tgt[0].size(0)
if sample_cnt >= len(train_iter.dataset.examples):
    break

sample_cnt = 0
total_loss, total_word_count = 0, 0

with torch.no_grad(): # In validation, we don't need to get gradients.
    model.eval() # Turn-on the evaluation mode.

    for batch_index, batch in enumerate(valid_iter):
        current_batch_word_cnt = torch.sum(batch.tgt[1])
        x = batch.src
        y = batch.tgt[0][:, 1:]
        # |x| = (batch_size, length)
        # |y| = (batch_size, length)

        # Take feed-forward
        y_hat = model(x, batch.tgt[0][:, :-1])
        # |y_hat| = (batch_size, length, output_size)

        loss = get_loss(y, y_hat, criterion, do_backward = False)

        total_loss += float(loss)
        total_word_count += int(current_batch_word_cnt)

        sample_cnt += batch.tgt[0].size(0)
        if sample_cnt >= len(valid_iter.dataset.examples):
            break

    # Print result of validation.
    avg_loss = total_loss / total_word_count
    print("valid loss: %.4f\tPPL: %.2f" % (avg_loss, np.exp(avg_loss)))

```

```

if lowest_valid_loss > avg_loss:
    lowest_valid_loss = avg_loss
    no_improve_cnt = 0

    # Altough there is an improvement in last epoch, we need to decay learning rate.
    if epoch >= config.lr_decay_start_at:
        current_lr = max(config.min_lr, current_lr * config.lr_decay_rate)
    else:
        # Decrease learing rate if there is no improvement.
        current_lr = max(config.min_lr, current_lr * config.lr_decay_rate)
        no_improve_cnt += 1

    # Again, turn-on the training mode.
    model.train()

# Set a filename for model of last epoch.
# We need to put every information to filename, as much as possible.
model_fn = config.model.split(".")
model_fn = model_fn[:-1] + [">%02d" % epoch, "%.2f-%.2f" % (train_loss, np.mean(valid_losses))]

# PyTorch provides efficient method for save and load model, which uses
to_save = {"model": model.state_dict(),
            "config": config,
            "epoch": epoch + 1,
            "current_lr": current_lr
            }
if others_to_save is not None: # Add others if it is necessary.
    for k, v in others_to_save.items():
        to_save[k] = v
torch.save(to_save, '.'.join(model_fn))

# Take early stopping if it meets the requirement.
if config.early_stop > 0 and no_improve_cnt > config.early_stop:
    break

```

## simple\_nmt/search.py

```
from operator import itemgetter

import torch
import torch.nn as nn

import data_loader

LENGTH_PENALTY = 1.2
MIN_LENGTH = 5

class SingleBeamSearchSpace():

    def __init__(self, hidden, h_t_tilde = None, beam_size = 5, max_length = 255):
        self.beam_size = beam_size
        self.max_length = max_length

        super(SingleBeamSearchSpace, self).__init__()

        # To put data to same device.
        self.device = hidden[0].device
        # Inferred word index for each time-step. For now, initialized with init
        self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) +
        # Index origin of current beam.
        self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.device)
        # Cumulative log-probability for each beam.
        self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')]) * (beam_
        # 1 if it is done else 0
        self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

        # We don't need to remember every time-step of hidden states: prev_hidden
        # What we need is remember just last one.
        # Future work: make this class to deal with any necessary information for

        # |hidden[0]| = (n_layers, 1, hidden_size)
        self.prev_hidden = torch.cat([hidden[0]] * beam_size, dim = 1)
```

```

        self.prev_cell = torch.cat([hidden[1]] * beam_size, dim = 1)
        # |prev_hidden| = (n_layers, beam_size, hidden_size)
        # |prev_cell| = (n_layers, beam_size, hidden_size)

        # |h_t_tilde| = (batch_size = 1, 1, hidden_size)
        self.prev_h_t_tilde = torch.cat([h_t_tilde] * beam_size, dim = 0) if h_t_tilde is not None
        # |prev_h_t_tilde| = (beam_size, 1, hidden_size)

        self.current_time_step = 0
        self.done_cnt = 0

    def get_length_penalty(self, length, alpha = LENGTH_PENALTY, min_length = MIN_LENGTH):
        # Calculate length-penalty, because shorter sentence usually have bigger penalty.
        # Thus, we need to put penalty for shorter one.
        p = (1 + length) ** alpha / (1 + min_length) ** alpha

        return p

    def is_done(self):
        # Return 1, if we had EOS more than 'beam_size'-times.
        if self.done_cnt >= self.beam_size:
            return 1
        return 0

    def get_batch(self):
        y_hat = self.word_indice[-1].unsqueeze(-1)
        hidden = (self.prev_hidden, self.prev_cell)
        h_t_tilde = self.prev_h_t_tilde

        # |y_hat| = (beam_size, 1)
        # |hidden| = (n_layers, beam_size, hidden_size)
        # |h_t_tilde| = (beam_size, 1, hidden_size) or None
        return y_hat, hidden, h_t_tilde

    def collect_result(self, y_hat, hidden, h_t_tilde):
        # |y_hat| = (beam_size, 1, output_size)
        # |hidden| = (n_layers, beam_size, hidden_size)
        # |h_t_tilde| = (beam_size, 1, hidden_size)

```

```

output_size = y_hat.size(-1)

self.current_time_step += 1

# Calculate cumulative log-probability.
# First, fill -inf value to last cumulative probability, if the beam is
# Second, expand -inf filled cumulative probability to fit to 'y_hat'. (
# Third, add expanded cumulative probability to 'y_hat'
cumulative_prob = y_hat + self.cumulative_probs[-1].masked_fill_(self.mask)
# Now, we have new top log-probability and its index. We picked top inde
# Be aware that we picked top-k from whole batch through 'view(-1)'.
top_log_prob, top_indice = torch.topk(cumulative_prob.view(-1), self.beam_
# |top_log_prob| = (beam_size)
# |top_indice| = (beam_size)

self.word_indice += [top_indice.fmod(output_size)] # Because we picked fr
self.prev_beam_indice += [top_indice.div(output_size).long()] # Also, we

# Add results to history boards.
self.cumulative_probs += [top_log_prob]
self.masks += [torch.eq(self.word_indice[-1], data_loader.EOS)] # Set fin
self.done_cnt += self.masks[-1].float().sum() # Calculate a number of fin

# Set hidden states for next time-step, using 'index_select' method.
self.prev_hidden = torch.index_select(hidden[0], dim = 1, index = self.pre
self.prev_cell = torch.index_select(hidden[1], dim = 1, index = self.prev_
self.prev_h_t_tilde = torch.index_select(h_t_tilde, dim = 0, index = self.p

def get_n_best(self, n = 1):
    sentences = []
    probs = []
    founds = []

    for t in range(len(self.word_indice)): # for each time-step,
        for b in range(self.beam_size): # for each beam,
            if self.masks[t][b] == 1: # if we had EOS on this time-step and b
                # Take a record of penaltified log-probability.
                probs += [self.cumulative_probs[t][b] / self.get_length_penalty(t, b)]
            else:
                founds.append(1)
                break
        if sum(founds) == n:
            break
    return sentences, probs

```

```

        founds += [(t, b)]

# Also, collect log-probability from last time-step, for the case of EOS
for b in range(self.beam_size):
    if self.cumulative_probs[-1][b] != -float('inf'):
        if not (len(self.cumulative_probs) - 1, b) in founds:
            probs += [self.cumulative_probs[-1][b]]
            founds += [(t, b)]


# Sort and take n-best.
sorted_founds_with_probs = sorted(zip(founds, probs),
                                  key = itemgetter(1),
                                  reverse = True
                                  )[:n]
probs = []

for (end_index, b), prob in sorted_founds_with_probs:
    sentence = []

    # Trace from the end.
    for t in range(end_index, 0, -1):
        sentence = [self.word_indice[t][b]] + sentence
        b = self.prev_beam_indice[t][b]

    sentences += [sentence]
    probs += [prob]

return sentences, probs

```

utils.py

```

import torch

def get_grad_norm(parameters, norm_type = 2):
    parameters = list(filter(lambda p: p.grad is not None, parameters))

```

```

total_norm = 0

try:
    for p in parameters:
        param_norm = p.grad.data.norm(norm_type)
        total_norm += param_norm ** norm_type
    total_norm = total_norm ** (1. / norm_type)
except Exception as e:
    print(e)

return total_norm

def get_parameter_norm(parameters, norm_type = 2):
    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.data.norm(norm_type)
            total_norm += param_norm ** norm_type
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm

```

## Advanced Topic on Neural Machine Translation



Figure 1: Christopher Manning – Image from Web

## Multi-lingual Neural Machine Translation

이제부터는 기계번역의 성능을 끌어올리기 위한 고급 기법들을 설명하고자 합니다. 코드를 직접 구현하고 실습을 해 보기보단, 논문을 소개하는 위주가 될 것 입니다. 앞으로 소개할 기술(논문)들은 일부는 기계번역에만 적용 가능한 기술들도 있지만, 자연어 생성 또는 시퀀스 데이터 생성에 응용될 수 있는 기술들도 있습니다.

기존의 번역 시스템이 Seq2seq를 필두로 어느정도 안정된 성능을 제공함에 따라서, 이를 활용한 여러가지 추가적인 연구 주제가 생겨났습니다. 하나의 end2end 모델에서 여러 언어쌍의 번역을 동시에 제공하는 multi-lingual 기계번역이 그 주제중에 하나입니다.

## Zero-shot Learning

이 흥미로운 방식은 [Johnson et al. 2016]에서 제안 되었습니다. 이 방식의 특징은 여러 언어쌍의 parallel 코퍼스를 하나의 모델에 훈련하면 부가적으로 parallel 코퍼스에 존재하지 않은 언어쌍도 번역이 가능하다는 것 입니다. 즉, 한번도 기계번역 모델에게 데이터를 보여준 적이 없지만, 해당 언어쌍 번역을 처리할 수 있기 때문에, zero-shot learning이라는 이름이 붙었습니다. 뿐만 아니라, parallel 코퍼스가 적은 경우에도 부가적인 효과를 발휘 합니다.

방법은 너무 간단합니다. 아래와 같이 기존 parallel 코퍼스의 맨 앞에 특수 토큰을 삽입함으로써 완성됩니다. 삽입된 토큰에 따라서 target 언어가 결정됩니다.

- Hello, how are you? → Hola, ¿c□mo est□s?
- <2es> Hello, how are you? → Hola, ¿c□mo est□s?

실험의 목표는 단순히 Multi-lingual end2end model을 구현하는 것이 아닌, 다른 언어쌍의 코퍼스를 활용하여 특정 언어쌍 번역기의 성능을 올릴 수 있는가에 대한 관점도 있습니다. 이에 따라 실험은 크게 4가지 관점에서 수행되었습니다.

번호	방법	설명
1	Many to One	다수의 언어를 encoder에 넣고 훈련시킵니다.
2	One to Many	다수의 언어를 decoder에 넣고 훈련시킵니다.
3	Many to Many	다수의 언어를 encoder와 decoder에 모두 넣고 훈련시킵니다.
4	Zero-shot Translation	위의 방법으로 훈련된 모델에서 zero-shot translation의 성능을 평가합니다.

언어가 다른 코퍼스를 하나로 합치다보면 양이 다르기 때문에 이에 대한 대처 방법도 정의 되어야 합니다. 따라서 아래의 실험들에서 oversampling 기법의 사용 유무도 같이 실험이 되었습니다. Oversampling 기법은 양이 적은 코퍼스를 양이 많은 코퍼스에 양과 비슷하도록 (데이터를 반복시켜) 양을 늘려 맞춰주는 방법을 말합니다.

### Many to One

이 실험에서는 전체적으로 성능이 향상 된 것을 볼 수 있습니다. 하단의 일본어, 한국어, 스페인어, 포르투갈어 실험의 경우에는 모두 oversampling을 기준으로 실험되었습니다.

Many to One: BLEU scores on various data sets for single language pair and multilingual models.

	Model	Single	Multi	Diff
WMT German→English (oversampling)	30.43	30.59	+0.16	
WMT French→English (oversampling)	35.50	35.73	+0.23	
WMT German→English (no oversampling)	30.43	30.54	+0.11	
WMT French→English (no oversampling)	35.50	36.77	+1.27	
Prod Japanese→English	23.41	23.87	+0.46	
Prod Korean→English	25.42	25.47	+0.05	
Prod Spanish→English	38.00	38.73	+0.73	
Prod Portuguese→English	44.40	45.19	+0.79	

Figure 2: Many to One

## One to Many

One to Many: BLEU scores on various data sets for single language pair and multilingual models.

	Model	Single	Multi	Diff
WMT English→German (oversampling)	24.67	24.97	+0.30	
WMT English→French (oversampling)	38.95	36.84	-2.11	
WMT English→German (no oversampling)	24.67	22.61	-2.06	
WMT English→French (no oversampling)	38.95	38.16	-0.79	
Prod English→Japanese	23.66	23.73	+0.07	
Prod English→Korean	19.75	19.58	-0.17	
Prod English→Spanish	34.50	35.40	+0.90	
Prod English→Portuguese	38.40	38.63	+0.23	

Figure 3: One to Many

이 실험에서는 이전 실험과 달리 성능의 향상이 있다고 보기 힘듭니다. 게다가 oversampling과 관련해서 코퍼스의 양이 적은 영어/독일어 코퍼스는 oversampling의 이득을 본 반면, 양이 충분한 영어/프랑스어 코퍼스의 경우에는 oversampling을 하면 더 큰 손해를 보는 것을 볼 수 있습니다.

## Many to Many

이 실험에서도 대부분의 실험결과가 성능의 하락으로 이어졌습니다. (그렇지만 절대적인 BLEU 수치는 쓸만합니다.)

Many to Many: BLEU scores on various data sets for single language pair and multilingual models.

	Model	Single	Multi	Diff
WMT English→German (oversampling)	24.67	24.49	-0.18	
WMT English→French (oversampling)	38.95	36.23	-2.72	
WMT German→English (oversampling)	30.43	29.84	-0.59	
WMT French→English (oversampling)	35.50	34.89	-0.61	
WMT English→German (no oversampling)	24.67	21.92	-2.75	
WMT English→French (no oversampling)	38.95	37.45	-1.50	
WMT German→English (no oversampling)	30.43	29.22	-1.21	
WMT French→English (no oversampling)	35.50	35.93	+0.43	
Prod English→Japanese	23.66	23.12	-0.54	
Prod English→Korean	19.75	19.73	-0.02	
Prod Japanese→English	23.41	22.86	-0.55	
Prod Korean→English	25.42	24.76	-0.66	
Prod English→Spanish	34.50	34.69	+0.19	
Prod English→Portuguese	38.40	37.25	-1.15	
Prod Spanish→English	38.00	37.65	-0.35	
Prod Portuguese→English	44.40	44.02	-0.38	

Figure 4: Many to Many

Portuguese→Spanish BLEU scores using various models.

	Model	Zero-shot	BLEU
(a)	PBMT bridged	no	28.99
(b)	NMT bridged	no	30.91
(c)	NMT Pt→Es	no	31.50
(d)	Model 1 (Pt→En, En→Es)	yes	21.62
(e)	Model 2 (En↔{Es, Pt})	yes	24.75
(f)	Model 2 + incremental training	no	31.77

Figure 5: Zero-shot Translation

## Zero-shot Translation

이 실험은 Zero-shot learning의 성능을 평가하였습니다. bridged 방법은 중간 언어를 영어로 하여 *Portuguese → English → Spanish* 2단계에 걸쳐 번역을 한 경우를 말합니다. (PBMT 방식은 SMT방식 중의 하나입니다.) *NMT Pt → Es*는 단순 parallel 코퍼스를 활용하여 기존의 방법대로 훈련한 baseline입니다.

모델1은 *Pt → En*, *En → Es*를 한 모델에 훈련한 버전입니다. 그리고 모델2는 *En ↔ Pt*, *En ↔ Es* 코퍼스를 한 모델에 훈련한 버전입니다. 모델2는 총 4가지 코퍼스를 훈련 한 점을 주의해야 합니다.

마지막으로 모델2 + incremental 학습 방식은 (c) 보다 적은양의 parallel 코퍼스를 기준에 훈련한 모델2에 추가적으로 훈련한 모델입니다.

비록 모델1과 모델2는 훈련 중에 한번도 *Pt → Es* 데이터를 보지 못했지만, 20이 넘는 BLEU를 보여주는 것을 알 수 있습니다. 하지만 bridge 방식의 (a), (b) 보다 성능이 떨어지는 것을 알 수 있습니다. 다행히도 (f)의 경우에는 (c)보다 (큰 차이는 아니지만) 성능이 뛰어난 것을 알 수 있습니다. 따라서 우리는 parallel 코퍼스의 양이 얼마 되지 않는 언어쌍의 번역기를 훈련할 때에 위와 같은 방법을 통해서 성능을 끌어올릴 수 있음을 알 수 있습니다.

## Conclusion

앞서 다룬 monolingual 코퍼스를 활용하는 방법의 연장선상으로써 위와 같이 multi-lingual 기계번역 모델로서는 의의가 있지만, 성능에 있어서는 이득이 있었기 때문에 실제 사용에는 한계가 있습니다. 더군다나 훈련 데이터의 양이 적은 언어쌍에 대해서는 성능의 향상이 있지만 뒤 챕터에 설명할 방법들을 사용하면 그다지 좋은 방법은 아닙니다.

## Applications

우리는 합성 토큰을 추가하는 방식을 다른 곳에서도 응용할 수 있습니다. 다른 도메인의 데이터를 하나로 모아 번역기를 훈련시키는 과정에 사용 가능합니다. 예를 들어 코퍼스를 뉴스기사와 미드 자막에서 각각 모았다고 가정하면, 문어체와 대화체로 도메인을 나누어 특수 토큰을 추가하여 우리가 원하는대로 번역문의 말투를 바꾸어줄 수 있을 겁니다. 또는 마찬가지로 의료용과 법률용으로 나누어 번역기의 모드를 바꾸어줄 수 있을 겁니다.

# Improve Neural Machine Translation using Monolingual Corpora

번역 시스템을 훈련하기 위해서는 다양한 Parallel Corpus(병렬 말뭉치)가 필요합니다. 필자의 경험상 대략 300만 문장쌍이 있으면 완벽하지는 않지만 나름 쓸만한 번역기가 나오기 시작합니다. 하지만 인터넷에는 정말 수치로 정의하기도 힘들 정도의 monolingual corpus가 널려 있는데 반해서, 이러한 parallel corpus를 대량으로 얻는 것은 굉장히 어려운 일입니다. 또한, monolingual corpus가 그 양이 많기 때문에 실제 우리가 사용하는 언어의 확률분포에 좀 더 가까울 수 있고, 따라서 언어모델을 구성함에 있어서 훨씬 유리합니다. 즉, 앞으로 소개할 기법들은 다양한 monolingual corpus를 활용하여 해당 언어의 언어모델을 보완하는 것이 주 목적입니다. 이번 섹션은 이러한 값싼 monolingual corpus를 활용하여 신경망 기계번역 시스템의 성능을 쥐어짜는 방법들에 대해서 다룹니다.

## Language Model Ensemble

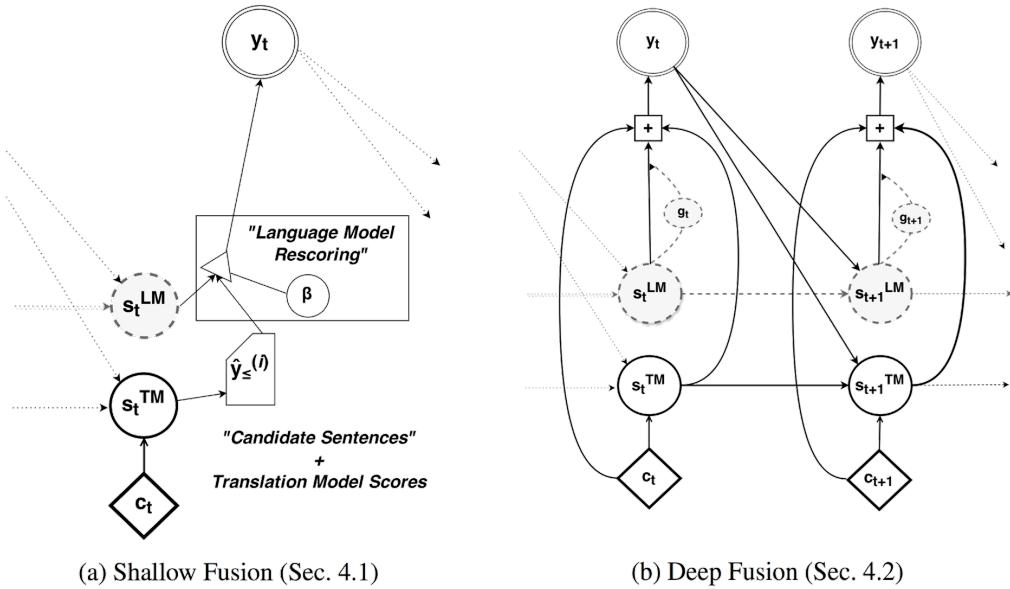


Figure 6: [Gulcehre et al. 2015]

이 방법은 Bengio 교수의 연구실에서 쓴 페이퍼인 [Gulcehre et al. 2015]에서 제안

된 방법입니다. 언어모델을 명시적으로 앙상블하여 디코더의 성능을 올리고자 하였습니다. 두 개의 다른 모델을 쓴 shallow fusion 방법 보다, 언어모델을 Seq2seq에 포함시켜 end2end 학습을 통해 한 개의 모델로 만든 deep fusion 방법이 좀 더 나은 성능을 나타냈습니다. 두 방식 모두 monolingual corpus를 활용하여 언어모델을 학습한 이후 실제 번역기를 훈련시킬 때에는 네트워크 파라미터 값을 고정한 상태로 seq2seq 모델을 훈련합니다.

	Development Set		Test Set			
	dev2010	tst2010	tst2011	tst2012	tst2013	Test 2014
<b>Previous Best (Single)</b>	15.33	17.14	18.77	18.62	18.88	-
<b>Previous Best (Combination)</b>	-	17.34	18.83	18.93	18.70	-
<b>NMT</b>	14.50	18.01	18.40	18.77	19.86	18.64
<b>NMT+LM (Shallow)</b>	14.44	17.99	18.48	18.80	19.87	18.66
<b>NMT+LM (Deep)</b>	<b>15.69</b>	<b>19.34</b>	<b>20.17</b>	<b>20.23</b>	<b>21.34</b>	<b>20.56</b>

Figure 7: [Gulcehre et al.2015]

성능상으로는 뒤에 다룰 내용들보다 성능 상의 이득이 적지만, 그 내용이 방법의 장점은 monolingual corpus를 전부 활용 할 수 있다는 것입니다.

## Dummy source sentence translation

아래의 내용들은 전부 Edinburgh 대학의 Nematus 번역시스템에서 제안되고 사용된 내용들입니다. 이 페이퍼[Sennrich et al.2015]의 저자인 Rico Sennrich는 좀 전의 내용처럼 명시적으로 언어모델을 앙상블하는 대신, 디코더로 하여금 monolingual corpus를 학습할 수 있게 하는 방법을 제안하였습니다. 예전 챕터에서 다루었듯이, 디코더는 Conditional Neural Network Language Model이라고 할 수 있는데, source 문장인  $X$ 를 빈 입력을 넣어줌으로써, (그리고 Attention등을 모두 dropout 시켜 끊어줌으로써) condition을 없애는 것이 이 방법의 핵심입니다. 저자는 이 방법을 사용하면 decoder가 monolingual corpus의 language model을 학습하는 것과 같다고 하였습니다.

## Back translation

그리고 같은 [Sennrich et al.2015]에서 좀 더 발전된 다른 방법을 제시하였습니다. 이 방법은 기존의 훈련된 반대 방향 번역기를 사용하여 monolingual corpus를

기계번역하여 합성(synthetic) parallel corpus를 만들어 이것을 훈련에 사용하는 방식입니다. 중요한 점은 기계번역에 의해 만들어진 합성 parallel corpus를 사용할 때, 반대 방향의 번역기의 훈련에 사용한다는 것입니다.

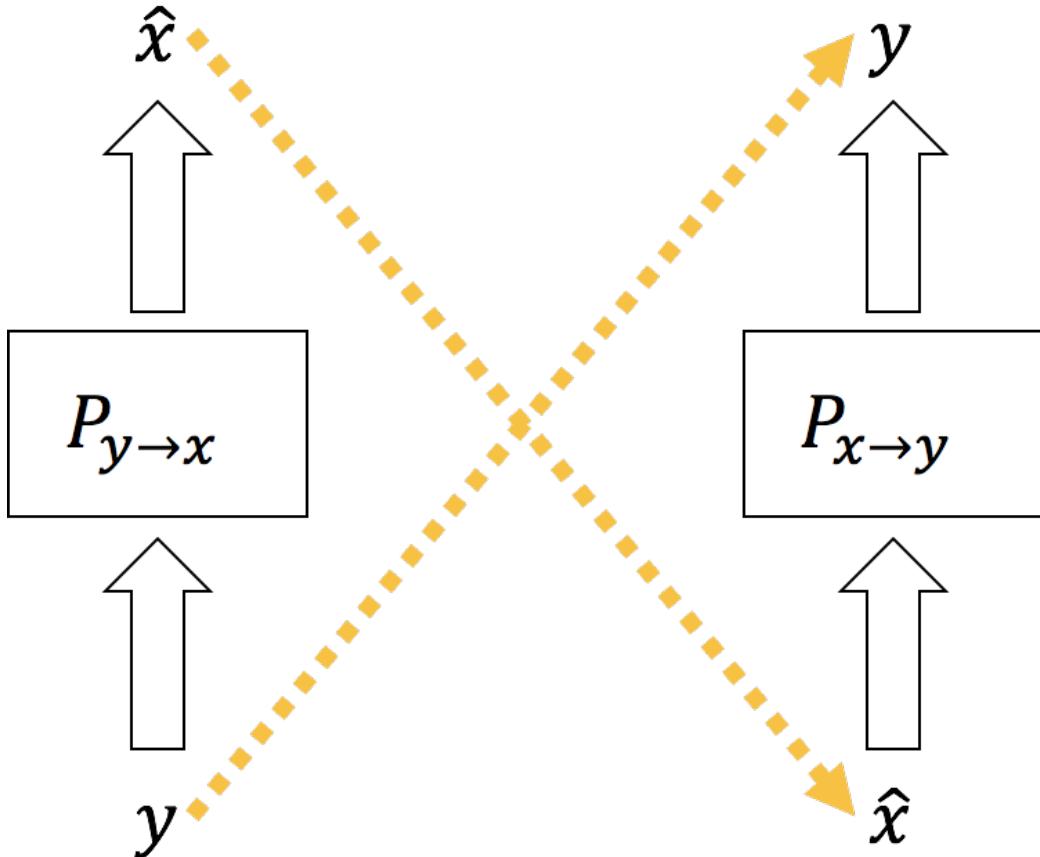


Figure 8: Back Translation 개요

$$\hat{\theta}_e = \underset{\theta}{\operatorname{argmax}} P_{f \rightarrow e}(e | \hat{f}; \theta_e)$$

where  $\hat{f} = \underset{f}{\operatorname{argmax}} P_{e \rightarrow f}(f | e; \theta_f)$

예를 들어, 한국어 monolingual corpus가 있을 때, 이것을 기존에 훈련된 한→영 번역기에 기계번역시켜 한–영 합성 parallel corpus를 만들고, 이것을 영→한

번역기를 훈련시키는데 사용하는 것 입니다. 이러한 방법의 특성 때문에 back translation이라고 명명되었습니다.

name	training instances	BLEU			
		newstest2014		newstest2015	
		single	ens-4	single	ens-4
parallel	37m (parallel)	19.9	20.4	22.8	23.6
+monolingual	49m (parallel) / 49m (monolingual)	20.4	21.4	23.2	24.6
+synthetic	44m (parallel) / 36m (synthetic)	<b>22.7</b>	<b>23.8</b>	<b>25.7</b>	<b>26.5</b>

Figure 9: [Sennrich et al.2015]

위의 테이블은 Dummy source translation(=monolingual)과 back translation(=합성) 방식에 대해서 성능을 실험한 결과입니다. 두 가지 방법 모두 사용하였을 때에 성능이 제법 향상된 것을 볼 수 있습니다. Parallel corpus와 거의 같은 양의 corpus가 각각 사용되었습니다. 위에서 언급했듯이, 명시적 언어모델 양상을 방식에서는 코퍼스 사용량의 제한이 없었지만, 이 방식에서는 기존의 parallel corpus와 차이 없이 섞어서 동시에 훈련에 사용하기 때문에, monolingual corpus의 양이 parallel corpus 보다 많아질 경우 주객전도 현상이 일어날 수 있습니다. 따라서 그 양을 제한하여 훈련에 사용하였습니다.

## Copied translation

이 방식은 같은 저자인 Rich Sennrich에 의해서 [Currey et al.2017] Copied Monolingual Data Improves Low-Resource Neural Machine Translation에서 제안 되었습니다. 기존의 Dummy source sentence translation 방식에서 좀 더 나아진 방식입니다. 기존의 방식 대신에 source 쪽과 target 쪽에 똑같은 데이터를 넣어 훈련시키는 것입니다. 기존의 dummy source sentence 방식은 encoder에서 decoder로 가는 길을 훈련 시에 dropout 처리 해주어야 했지만, 이 방식은 그럴 필요가 없어진 것이 장점입니다. 하지만 source 언어의 vocabulary에 target language의 vocabulary가 포함되어야 하는 불필요함을 감수해야 합니다.

## Conclusion

위와 같이 여러 방법들이 제안되었지만, 위의 방법 중에서는 구현 방법의 용이성과 효율성 때문에 back translation이 가장 많이 쓰이는 추세입니다. Back translation 기법은 간단한 방법임에도 불구하고 효과적으로 성능 향상을 얻을 수 있습니다.

system	TR→EN		EN→TR	
	2016	2017	2016	2017
baseline	20.0	19.7	13.2	14.7
+copied	20.2	19.7	13.8	15.6

Figure 10: [Sennrich at el.2017]

## Unsupervised Neural Machine Translation

[Artetxe at el.2017]

## Fully Convolutional Seq2seq

Neural Machine Translation의 최강자는 Google이라고 모두가 여기고 있을 때, Facebook이 과감하게 이 논문[Gehring at el.2017]을 들고 도전장을 내밀었습니다. RNN방식의 seq2seq 대신에 오직 convolutional layer만을 이용한 방식의 seq2seq를 들고 나와, 기존의 방식에 대비해서 성능과 속도 두마리 토끼를 모두 잡았다고 주장하였습니다.

### Architecture

#### Position Embedding

이 방식은 RNN을 기반으로 하지 않기 때문에 position embedding을 사용하였습니다. RNN을 사용하면 우리가 직접적으로 위치 정보를 명시하지 않아도 자연스럽게 위치정보가 encoding 되지만, convolutional layer의 경우에는 이것이 없기 때문에 직접 위치 정보를 주어야 하기 때문입니다.

따라서 word embedding vector와 같은 dimension의 position embedding vector를 구하여 매 time-step마다 더해준 뒤, 상위 layer로 feed forward하게 됩니다.

하지만 position embedding이 없다고 이 방식이 동작하지 않는 것은 아닙니다. Position embedding의 유무에 따라서 실험결과 BLEU가 최대 0.5 정도 차이가 나기도

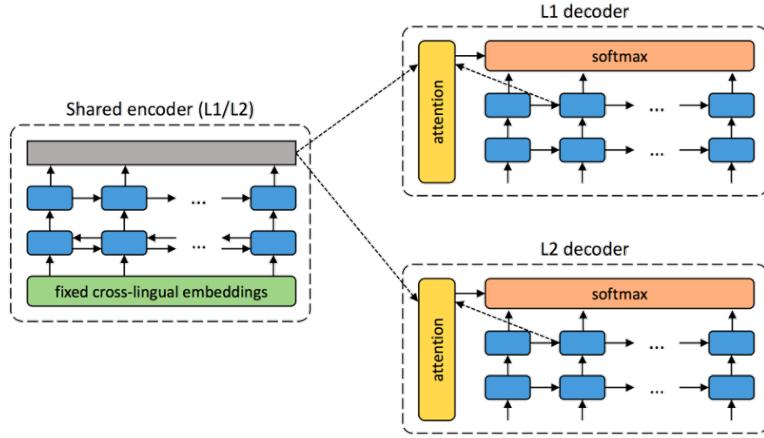


Figure 1: Architecture of the proposed system. For each sentence in language L1, the system is trained alternating two steps: *denoising*, which optimizes the probability of encoding a noised version of the sentence with the shared encoder and reconstructing it with the L1 decoder, and *back-translation*, which translates the sentence in inference mode (encoding it with the shared encoder and decoding it with the L2 decoder) and then optimizes the probability of encoding this translated sentence with the shared encoder and recovering the original sentence with the L1 decoder. Training alternates between sentences in L1 and L2, with analogous steps for the latter.

Figure 11:

Table 1: BLEU scores in newstest2014. Unsupervised systems are trained in the News Crawl monolingual corpus, semi-supervised systems are trained in the News Crawl monolingual corpus and 100,000 sentences from the News Commentary parallel corpus, and supervised systems (provided for comparison) are trained in the full parallel corpus, all from WMT 2014. For GNMT, we report the best single model scores from Wu et al. (2016).

		<b>FR-EN</b>	<b>EN-FR</b>	<b>DE-EN</b>	<b>EN-DE</b>
<b>Unsupervised</b>	1. Baseline (emb. nearest neighbor)	9.98	6.25	7.07	4.39
	2. Proposed (denoising)	7.28	5.33	3.64	2.40
	3. Proposed (+ backtranslation)	15.56	15.13	10.21	6.55
	4. Proposed (+ BPE)	15.56	14.36	10.16	6.89
<b>Semi-supervised</b>	5. Proposed (full) + 100k parallel	21.81	21.74	15.24	10.95
	6. Comparable NMT	20.48	19.89	15.04	11.05
<b>Supervised</b>	7. GNMT (Wu et al., 2016)	-	38.95	-	24.61

Figure 12:

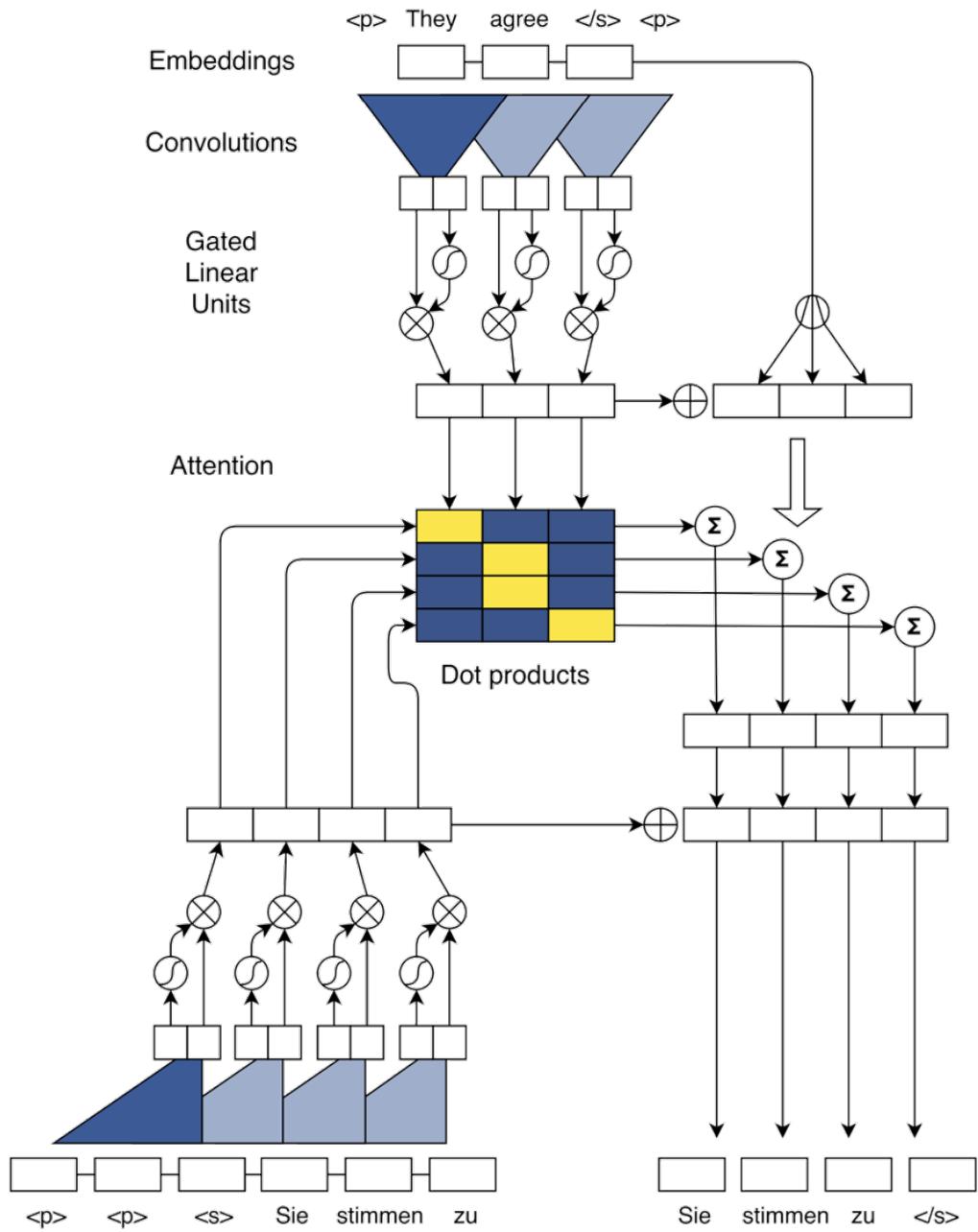


Figure 13: Fully Convolutional Sequence-to-Sequence 아키텍처

합니다.

### Convolutional Layer

Convolutional Layer를 사용한 encoder를 설명하기 이전에, 먼저 [Ranzato et al.2015]에서는 단순히 이전 layer의 결과값을 averaging하는 encoder를 제안하였습니다.

$$e_j = w_j + l_j, \quad z_j = \frac{1}{k} \sum_{t=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} e_{j+t}$$

where  $w_j$  is word vector and  $l_j$  is position embedding vector.

위와 같이 단순히 평균을 내는 것만으로도 어느정도의 성능을 낼 수 있었습니다. 만약 여기서 convolution filter를 사용하여 averaging 대신에 convolution 연산을 한다면 어떻게 될까요?

위의 물음에서 출발한 것이 이 논문의 핵심입니다. 따라서 kernel(or window) size  $k$ 인 convolution filter가  $d$ 개 channel의 입력을 받아서 convolution 연산을 수행하여  $2d$ 개 channel의 출력을 결과값으로 내놓습니다.

### Gated Linear Unit

이 논문에서는 [Dauphine et al.2016]에서 제안한 Gated Linear Unit(GLU)을 사용하였습니다.

$$v([A; B]) = A \otimes \sigma(B)$$

where  $A \in \mathbb{R}^d$  and  $B \in \mathbb{R}^d$   
thus  $[A; B] \in \mathbb{R}^{2d}$

GLU를 사용하여 직전 convolution layer에서의 결과값인 vector( $\in \mathbb{R}^{2d}$ )를 입력으로 삼아 gate 연산을 수행합니다. 이 연산은 LSTM이나 GRU에서의 gate들과 매우 비슷하게 동작을 수행합니다.

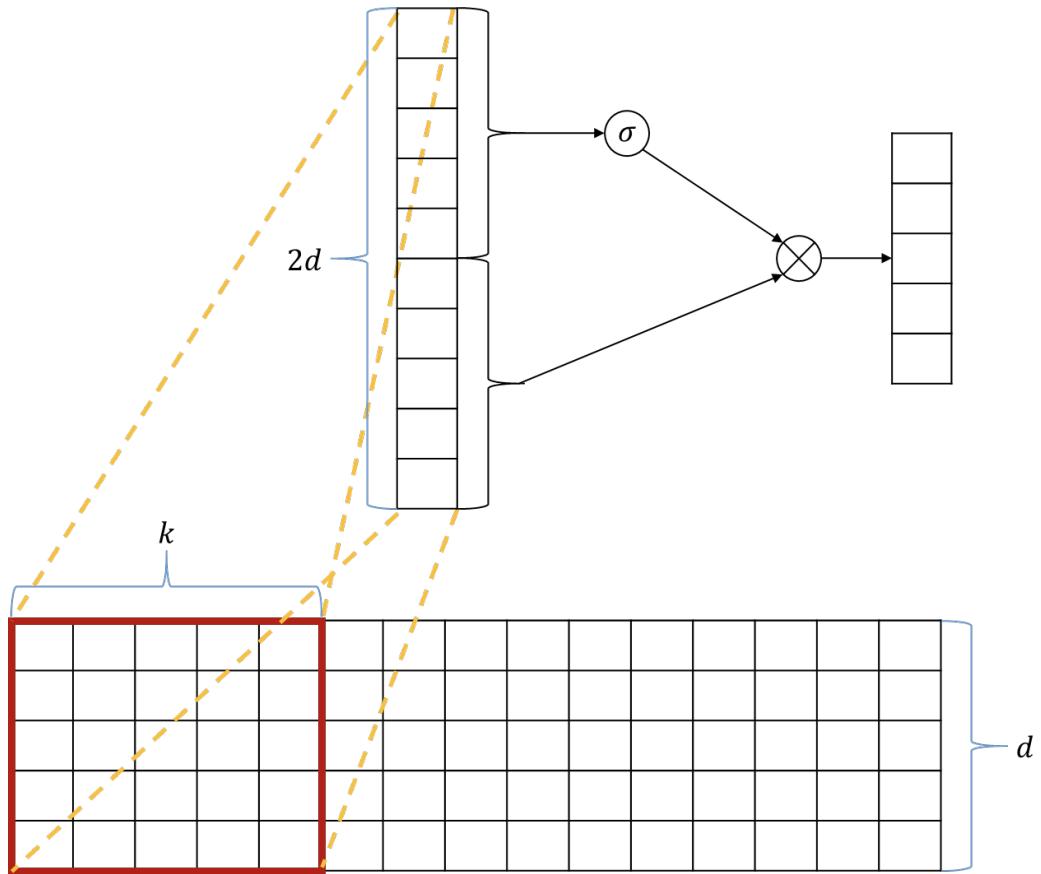


Figure 14: fconv 내에서 연산 과정을 도식화

## Attention

$z^u$ 를 인코더의 출력값,  $h_i^l$ 을 디코더의  $l$ 번째 layer의  $i$ 번째 결과값이라고 하고,  $g_i$ 를  $i - 1$ 번째 디코더의 출력값이라고 할 때, attention의 동작은 아래와 같습니다.

$$\begin{aligned} d_i^l &= W_d^l h_i^l + b_d^l + g_i \\ a_{ij}^l &= \frac{\exp(d_i^l z_j^u)}{\sum_{t=1}^m \exp(d_i^l z_t^u)} \\ c_i^l &= \sum_{j=1}^m a_{ij}^l (z_j^u + e_j) \end{aligned}$$

이렇게 구해진 context vector  $c_i^l$ 을 (기본적인 attention은 concatenate 하였던 것에 비해서) 아래와 같이  $h_i^l$ 에 그냥 더합니다. 그리고 이것을 다음 decoder layer의 입력으로 사용합니다.

$$\tilde{h}_i^l = h_i^l + c_i^l$$

이렇게  $l$ 번째 layer의 attention이 이루어지게 되는데, 이것을 매 layer마다 넣어주게 됩니다. 이전까지의 seq2seq는 attention layer가 전체 구조에서 한 개만 있었던 것에 비해서 Facebook이 제안한 이 구조에서는 모든 layer마다 나타날 수 있습니다. # Transformer (Attention is All You Need)

Facebook에서 CNN을 활용한 번역기에 대한 논문을 내며, 기존의 GNMT 보다 속도나 성능면에서 뛰어남을 자랑하자, 이에 질세라 Google에서 바로 곧이어 발표한 Attention is all you need [Vaswani et al. 2017] 논문입니다. 실제로 ArXiv에 Facebook이 5월에 해당 논문을 발표한데 이어서 6월에 이 논문이 발표되었습니다. 이 논문에서 Google은 아직까지 번역에 있어서 자신들의 기술력 우위성을 주장하였습니다.

## Architecture

“Attention is all you need”라는 제목의 논문답게 이 논문은 정말로 Attention만 구현해서 모든것을 해냅니다. 그리고 저자는 이 모델 구조를 Transformer라고 이름 붙였습니다. Encoder와 decoder를 설명하기에 앞서, sub-module부터 소개하겠습니다. Encoder와 decoder를 이루고 있는 sub-module은 크게 3가지로 나뉘어 집니다.

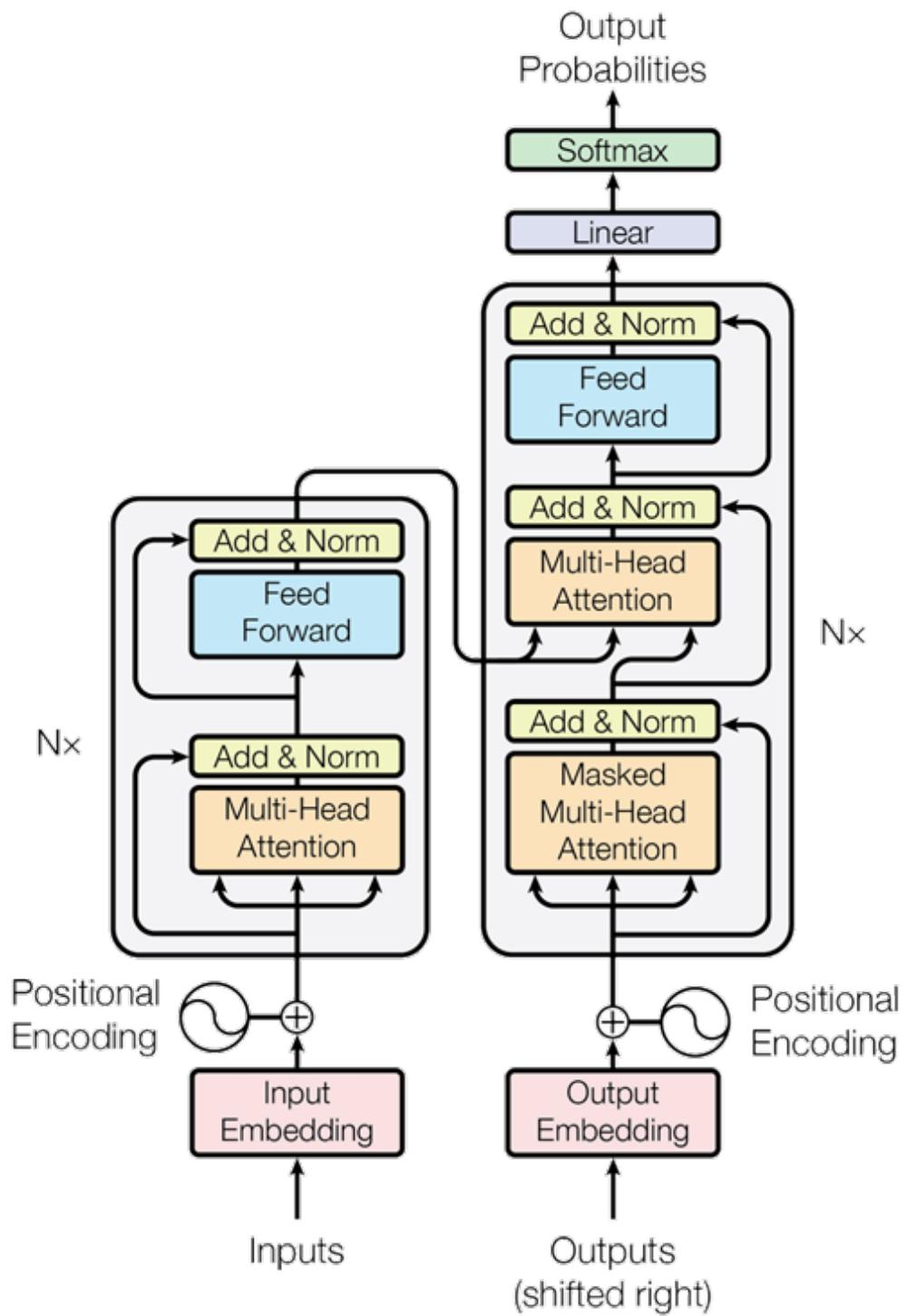


Figure 15: Transformer 아키텍처

---

명칭	역할
Self-attention	이전 layer의 output에 대해서 attention을 수행합니다.
Attention	Encoder의 output에 대해서 기존의 seq2seq와 같이 attention을 수행합니다.
Feed Forward Layer	attention layer을 거쳐 얻은 결과물을 최종적으로 정리합니다.

---

Encoder는 다수의 self-attention layer와 feed forward layer로 이루어져 있습니다. Decoder는 다수의 self-attention과 attention이 번갈아 나타나고, feed forward layer가 있습니다. 이처럼 Transformer는 구성되며 각 모듈에 대한 자세한 설명은 아래와 같습니다.

### Position Embedding

이전 Facebook 논문과 마찬가지로, RNN을 이용하지 않기 때문에, 위치정보를 단어와 함께 주는 것이 필요합니다. 따라서 Google에서도 마찬가지로 position embedding을 통해서 위치 정보를 나타내고자 하였으며, 그 수식은 약간 다릅니다.

$$\begin{aligned} \text{PE}(\text{pos}, 2i) &= \sin(\text{pos}/10000^{2i/d_{\text{model}}}) \\ \text{PE}(\text{pos}, 2i + 1) &= \cos(\text{pos}/10000^{2i/d_{\text{model}}}) \end{aligned}$$

Position embedding의 결과값의 dimension은 word embedding의 dimension과 같으며, 두 값을 더하여 encoder 또는 decoder의 입력으로 넘겨주게 됩니다.

### Attention

이 논문에서의 Attention 방식은 여러개의 attention으로 구성된 multi-head attention을 제안합니다. 마치 Convolution layer에서 여러개의 filter가 있어서 여러가지 다양한 feature를 뽑아 내는 것과 같은 원리라고 볼 수 있습니다.

기본적인 attention의 수식은 아래와 같습니다. 기본적인 attention은 원래 그냥 dot-product attention인데 scaled라는 이름이 붙은 이유는 key의 dimension인  $\sqrt{d_k}$ 로 나누어주었기 때문입니다. 이외에는 이전 섹션에서 다루었던 attention과 같습니다.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

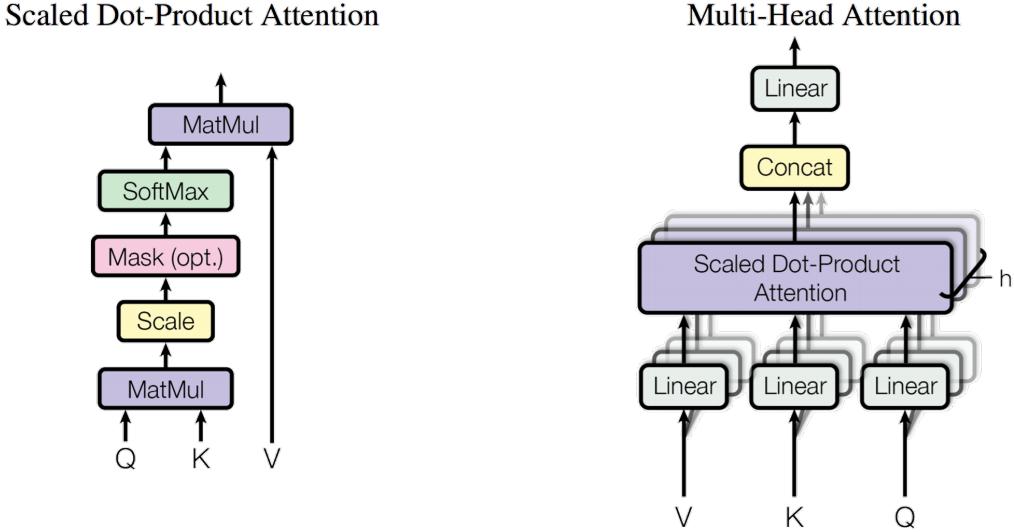


Figure 16: Transformer의 Attention 구성

이렇게 구성된 attention을 하나의 head로 삼아 Multi-Head Attention을 구성합니다.

$$\text{MultiHead}(Q, K, V) = [\text{head}_1; \text{head}_2; \dots; \text{head}_h]W^O$$

where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

where  $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ ,  $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ ,  
 $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$  and  $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$

$$d_k = d_v = d_{\text{model}}/h = 64$$

$$h = 8 \text{ and } d_{\text{model}} = 512$$

이때에 각 head의 Q, K, V마다 다른 W를 곱해줌으로써 각각 linear transformation 형태를 취해 줍니다. 즉, head마다 필요한 다른 정보(feature)를 attention을 통해 encoding 할 수 있게 됩니다. 해당 논문에서는 hidden size를 512로 하고 이를 8개의 head로 나누어 각 head의 hidden size는 64가 되도록 하였습니다.

실제 구현을 할 때에는 self attention의 경우에는 이전 layer의 출력값이 모두 Q, K, V를 이루게 됩니다. 같은 값이 Q, K, V로 들어가지만 linear transform을 해주기 때문에 상관이 없습니다. Decoder에서 수행하는 encoder에 대한 attention을 할

때에는, Q는 decoder의 이전 layer의 출력값이 되지만, K, V는 encoder의 출력값이 됩니다.

### Self Attention for Decoder

Decoder의 self-attention은 encoder의 그것과 조금 다릅니다. 이전 레이어의 출력값을 가지고 Q, K, V를 구성하는 것은 같지만, 약간의 제약이 더해졌습니다. 그 이유는 inference 할 때, 다음 time-step의 값을 알 수 없기 때문입니다. 따라서, self-attention을 하더라도 이전 time-step에 대해서만 접근이 가능하도록 해야 합니다. 이를 구현하기 위해서 scaled dot-product attention 계산을 할 때에 masking을 추가하여, 미래의 time-step에 대해서는 weight를 가질 수 없도록 하였습니다.

### Position-wise Feed Forward Layer

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \text{ where } d_{ff} = 2048$$

사실 여기에서 소개한 이 layer는 기존의 fully connected feed forward layer라기보단, kernel size가 1인 convolutional layer라고 볼 수 있습니다. Channel숫자가  $512 \rightarrow 2048$  으로 가는 convolution과,  $2048 \rightarrow 512$ 로 가는 convolution으로 이루어져 있는 것입니다.

### Evaluation

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	<b><math>3.3 \cdot 10^{18}</math></b>	
Transformer (big)	<b>28.4</b>	<b>41.8</b>	$2.3 \cdot 10^{19}$	

Figure 17: Transformer의 성능 비교

Google은 transformer를 통해서 State of the Art의 성능을 달성했다고 보고하였습니다. 뿐만 아니라, 기존의 RNN 및 Facebook의 ConvS2S보다 훨씬 빠른 속도로 훈련이 가능하다고 하였습니다. 실제로 위의 table을 보면, transformer의 training cost의 magnitude는  $10^{18}$ 으로, 대부분의 다른 방식  $10^{19}$ 와 급격한 차이를 보이는 것을 알 수 있습니다.

또 하나의 속도 개선의 원인은 input feeding의 부재입니다. RNN기반의 방식은 input feeding이 도입되면서 decoder를 훈련할 때 모든 time-step을 한번에 할 수 없게 되었습니다. 이로 인해서 대부분의 병목이 decoder에서 발생합니다. 하지만 transformer는 input feeding이 없기 때문에 한번에 모든 time-step에 대해서 계산할 수 있게 되었습니다.

비록 transformer가 최고 성능을 달성하기 했지만 그 모델 구조의 과격함 때문인지 (Facebook의 모델과 함께) 아직 주류로 편입되지 않았습니다. 아직 대부분의 논문들은 이 구조를 비교대상으로 논하기보다, RNN구조의 seq2seq를 대상으로 실험을 비교/진행 하곤 합니다.

## 읽을거리

# Reinforcement Learning for Natural Language Processing



Figure 1: Richard S. Sutton – Image from Web

# Reinforcement Learning for Natural Language Processing

## Discriminative learning vs Generative learning

2012년 이미지넷 대회(ImageNet competition)에서 딥러닝을 활용한 AlexNet이 우승을 차지한 아래로 컴퓨터 비전(Computer Vision), 음성인식(Speech Recognition), 자연어처리(Natural Language Processing) 등이 차례로 딥러닝에 의해 정복당해 왔습니다. 딥러닝이 뛰어난 능력을 보인 분야는 특히 분류(classification) 분야였습니다. 기존의 전통적인 방식과 달리 패턴 인식(pattern recognition) 분야에서는 압도적인 성능을 보여주었습니다. 이러한 분류(classification) 문제는 보통 Discriminative Learning에 속하는데 이를 일반화 하면 다음과 같습니다.

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(Y|X; \theta)$$

주어진  $x$ 에 대해서 최대의  $y$  값을 갖도록 하는 파라미터  $\theta$ 를 찾아내는 것입니다. 이러한 조건부 확률  $P(y|x)$  분포를 학습하는 것을 discriminative learning이라고 부릅니다. 하지만 이에 반해 Generative Learning은 확률분포  $P(x)$ 를 학습하는 것을 이릅니다. 따라서 Generative learning이 훨씬 더 학습하기 어렵습니다. 예를 들어

1. 사람의 생김새( $x$ )가 주어졌을 때 성별( $y|x$ )의 확률 분포를 배우는 것
2. 사람의 생김새( $x$ )와 성별( $y$ ) 자체의 확률 분포를 배우는 것

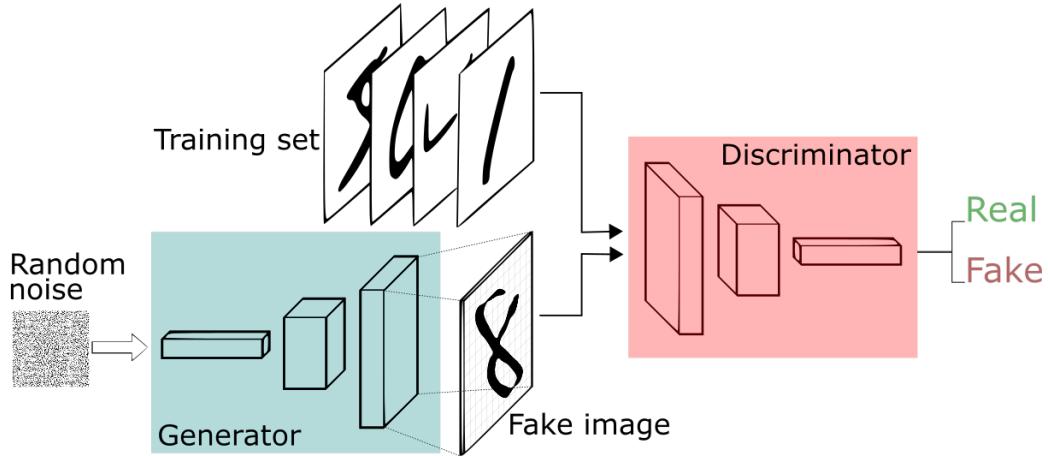
두 가지 경우를 비교하면 2번째가 훨씬 더 어려움을 알 수 있습니다. Discriminative learning은  $y$ 와  $x$ 와의 관계를 배우는 것이지만, generative learning은  $x$  (*and*  $y$ ) 자체를 배우는 것이기 때문입니다. 그리고 이것을 수식으로 일반화 하면 아래와 같습니다.

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(X; \theta)$$

사실 이제는 패턴인식과 같은 discriminative learning은 이제 딥러닝으로 너무나도 당연하게 잘 해결되기 때문에, 사람들의 관심과 연구 트렌드는 위와 같은 generative learning으로 집중되고 있습니다.

## Generative Adversarial Network (GAN)

2016년부터 주목받기 시작하여 2017년에 가장 큰 화제였던 분야는 단연 GAN이라고 말할 수 있습니다. Variational Auto Encoder(VAE)와 함께 Generative learning을 대표하는 방법 중에 하나입니다. GAN을 통해서 우리는 사실같은 이미지를 생성해내고 합성해내는 일들을 딥러닝을 통해 할 수 있게 되었습니다. 이러한 합성/생성된 이미지들을 통해, 자율주행과 같은 실생활에 중요하고 어렵지만 훈련 데이터셋을 얻기 힘든 문제들을 해결하는데 큰 도움을 얻을 수 있으리라고 예상됩니다. (실제로 GTA게임을 통해 자율주행을 훈련하려는 시도는 이미 유명합니다.)



Generative Adversarial Network overview – Image from web

$$\min_G \max_D \mathcal{L}(D, G) = \mathbb{E}_{x \sim p_r(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log (1 - D(G(z)))]$$

위와 같이 Generator( $G$ )와 Discriminator( $D$ ) 2개의 모델을 각기 다른 목표를 가지고 동시에 훈련시키는 것입니다.  $D$ 는 임의의 이미지를 입력으로 받아 이것이 실제 존재하는 이미지인지, 아니면 합성된 이미지인지 탐지 해 내는 역할을 합니다.  $G$ 는 어떤 이미지를 생성해 내되,  $D$ 를 속이는 이미지를 만들어 내는 것이 목표입니다. 이렇게 두 모델이 잘 균형을 이루며 min/max 게임을 펼치게 되면,  $G$ 는 결국 훌륭한 이미지를 합성해 내는 Generator가 됩니다.

### Why GAN is important?

마찬가지의 이유로 GAN 또한 주목받게 됩니다. 예를 들어, 생성된 이미지와 정답 이미지 간의 차이를 비교하는데 MSE(Mean Square Error)방식을 사용하게 되면,

결국 이미지는 MSE를 최소화 하기 위해서 자신의 학습했던 확률 분포의 중간으로 출력을 낼 수 밖에 없습니다. 예를 들어 사람의 얼굴을 일부 가리고 가려진 부분을 채워 넣도록 훈련한다면, MSE 손실함수(loss function) 아래에서는 각 픽셀마다 가능한 확률 분포의 평균값으로 채워 질 겁니다. 이것이 MSE를 최소화 하는 길이기 때문입니다. 하지만 우리는 그런 흐리멍텅한 이미지를 잘 생성된(채워진) 이미지라고 하지 않습니다. 따라서 사실적인 표현을 위해서는 MSE보다 정교한 목적함수(objective function)를 쓸 수 밖에 없습니다. GAN에서는 그러한 복잡한 함수를  $D$ 가 근사하여 해결한 것입니다.

## GAN과 NLP

위와 같이 GAN은 컴퓨터 비전(CV)분야에서 대성공을 이루었지만 자연어처리(NLP)에서는 적용이 어려웠습니다. 그 이유는 Natural Language 자체의 특성에 있습니다. 이미지라는 것은 어떠한 continuous한 값들로 채워진 2차원의 matrix입니다. 하지만 이와 달리 단어라는 것은 discrete한 symbol로써, 결국 언어라는 것은 어떠한 discrete한 값들의 순차적인 배열입니다. 비록 우리는 NNLM이나 NMT Decoder를 통해서 latent variable로써 언어의 확률을 모델링  $P(w_1, w_2, \dots, w_n)$  하고 있지만, 결국 언어를 나타내기 위해서는 해당 확률 모델에서 sampling(또는 argmax)을 하는 과정을 거쳐야 하고 이 과정은 미분이 불가능하거나 미분이 되더라도 gradient가 0이 되어 back-propagation이 불가 합니다.

$$\hat{w}_t = \operatorname{argmax} P(w_t | w_1, \dots, w_{t-1})$$

이러한 이유 때문에 Discriminator의 loss를 Generator에 전달 할 수가 없고, 따라서 GAN을 NLP에는 적용할 수 없는 인식이 지배적이었습니다. 하지만 강화학습을 사용함으로써 Adversarial learning을 NLP에도 우회적으로 사용할 수 있게 되었습니다.

참고로 Reparameterization Trick을 이용해 이 문제를 해결하려는 시도들도 있습니다. Gumbel Softmax [Jang et al. 2016]가 대표적입니다. 이를 활용하면 gradient를 전달 할 수 있기 때문에, policy gradient 없이 문제를 해결 할 수도 있습니다.

## Why we use RL?

위와 같이 GAN을 사용하기 위함 뿐만이 아니라, 강화학습은 매우 중요합니다. 어떠한 문제를 해결함에 있어서 cross entropy를 쓸 수 있는 분류(classification)

문제나, tensor간의 오류(error)를 구할 수 있는 MSE 등으로 정의 할 수 없는 복잡한 목적함수(objective function)들이 많이 존재하기 때문입니다. (비록 그동안 그러한 objective function으로 문제를 해결하였더라도 문제를 단순화 하여 접근한 것일 수도 있습니다.) 우리는 이러한 문제들을 강화학습을 통해 해결하거나 성능을 더 극대화 할 수 있습니다. 이를 위해서 잘 설계된 reward를 통해서 보다 복잡하고 정교한 task의 문제를 해결 할 수 있습니다.

이제 강화학습에 대해 간단히 다루고, 이를 NLP와 NMT에 어떻게 적용시키는지 다루어 보도록 하겠습니다.

## Math Basics

### Expectation

기대값(expectation)은 보상(reward)과 그 보상을 받을 확률을 곱한 값의 총 합을 통해 얻을 수 있습니다. 즉, reward에 대한 가중합(weighted sum)라고 볼 수 있습니다. 주사위의 경우에는 reward의 경우에는 1부터 6까지 받을 수 있지만, 각 reward에 대한 확률은 1/6로 동일합니다.

$$\text{expected reward from dice} = \sum_{x=1}^6 P(X = x) \times \text{reward}(x)$$

where  $P(x) = \frac{1}{6}, \forall x$  and  $\text{reward}(x) = x.$

따라서 실제 주사위의 기대값은 아래와 같이 3.5가 됩니다.

$$\frac{1}{6} \times (1 + 2 + 3 + 4 + 5 + 6) = 3.5$$

또한, 위의 수식은 아래와 같이 표현 할 수 있습니다.

$$\mathbb{E}_{X \sim P}[\text{reward}(x)] = \sum_{x=1}^6 P(X = x) \times \text{reward}(x) = 3.5$$

주사위의 경우에는 discrete variable을 다루는 확률 분포이고, continuous variable의 경우에는 적분을 통해 우리는 기대값을 구할 수 있습니다.

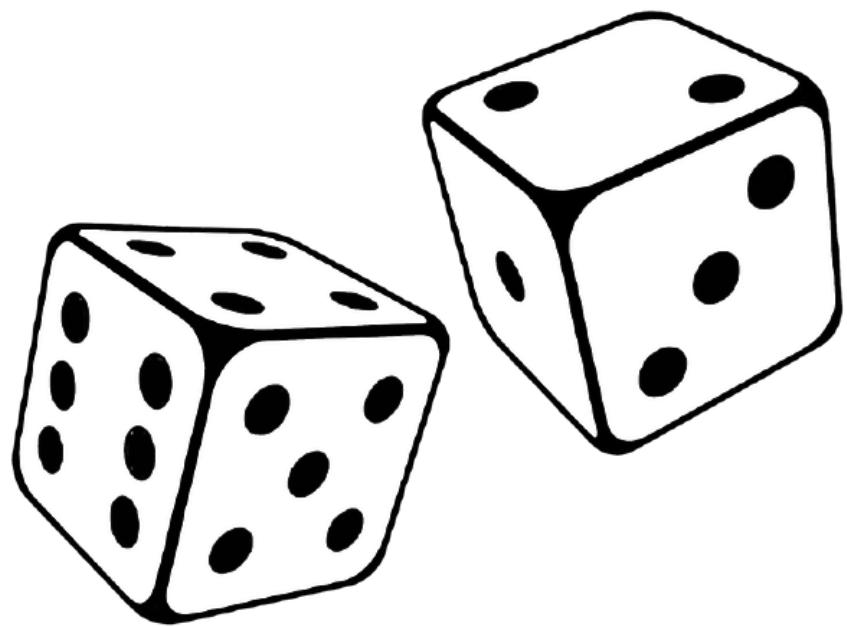


Figure 2:

## Monte Carlo Sampling

Monte Carlo Sampling은 난수를 이용하여 임의의 함수를 근사하는 방법입니다. 예를 들어 임의의 함수  $f$ 가 있을 때, 사실은 해당 함수가 Gaussian distribution을 따르고 있고, 충분히 많은 수의 random number  $x$ 를 생성하여,  $f(x)$ 를 구한다면,  $f(x)$ 의 분포는 역시 gaussian distribution을 따르고 있을 것 입니다. 이와 같이 임의의 함수에 대해서 Monte Carlo 방식을 통해 해당 함수를 근사할 수 있습니다.

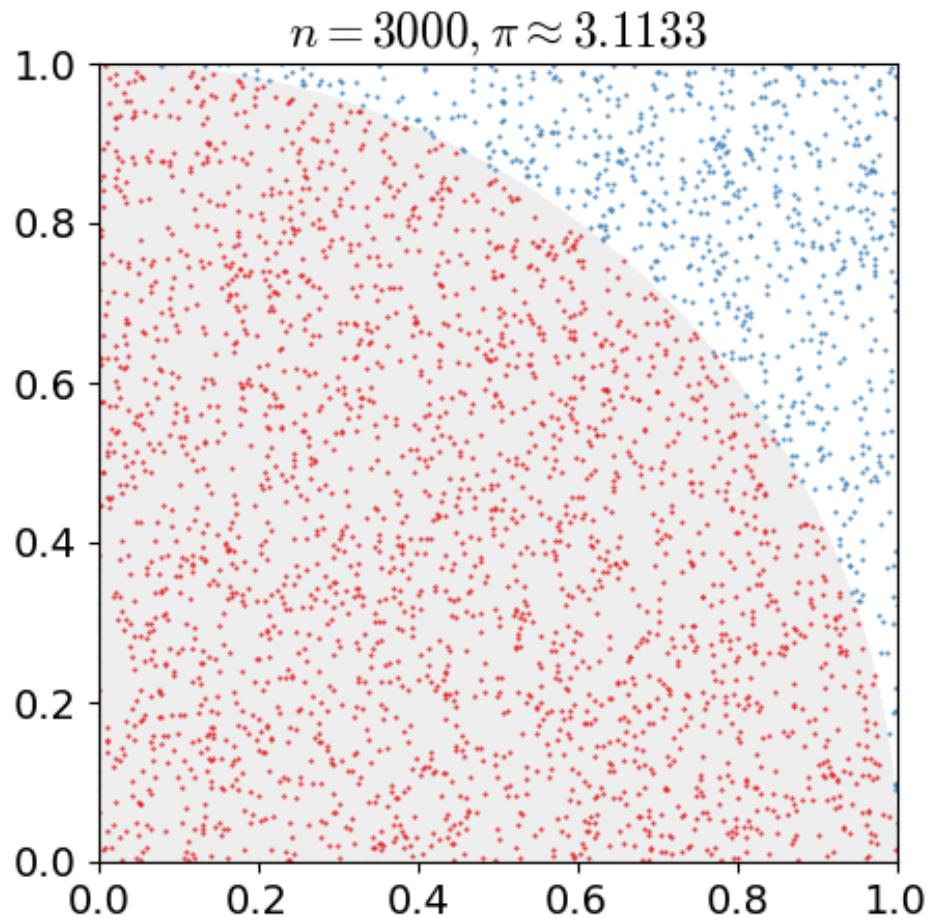


Figure 3: approximation of pi using Monte Carlo

따라서 Monte Carlo sampling을 사용하면 기대값(expectation) 내의 표현을 밖으로 꺼낼 수 있습니다. 즉, 주사위의 reward에 대한 기대값을 아래와 같이 간단히(simplify) 표현할 수 있습니다.

$$\mathbb{E}_{X \sim P}[\text{reward}(X)] \approx \frac{1}{N} \sum_{i=1}^N \text{reward}(x_i)$$

주사위 reward의 기대값은  $N$ 번 sampling한 주사위 값의 평균이라고 할 수 있습니다. 실제로  $N$ 이 무한대에 가까워질 수록 (커질 수록) 해당 값은 실제 기대값 3.5에 가까워질 것 입니다. 따라서 우리는 경우에 따라서  $N = 1$ 인 경우도 가정 해 볼 수 있습니다. 즉, 아래와 같은 수식이 될 수도 있습니다.

$$\mathbb{E}_{X \sim P}[\text{reward}(X)] \approx \text{reward}(x) = x$$

위와 같은 가정을 가지고 수식을 간단히 표현할 수 있게 되면, 이후 gradient를 구한다거나 할 때에 수식이 간단해져 매우 편리합니다. # Reinforcement Learning Basics

강화학습은 매우 방대하고 유서 깊은 학문입니다. 이 챕터에서 모든 내용을 상세하게 다루기엔 무리가 있습니다. 따라서 우리가 다룰 policy gradient를 알기 위해서 필요한 정도로 가볍게 짚고 넘어가고자 합니다. 더 자세한 내용은 Sutton 교수의 강화학습 책 Reinforcement Learning: An Introduction [Sutton et al.2017]을 참고하면 좋습니다.

## Universe

먼저, 강화학습은 어떤 형태로 구성이 되어 있는지 이야기 해 보겠습니다. 강화학습은 어떠한 객체가 주어진 환경에서 상황에 따라 어떻게 행동해야 할지 학습하는 방법에 대한 학문입니다. 그러므로 강화학습은 아래와 같은 요소들로 구성되어 동작 합니다.

처음 상황(state)  $S_t$  ( $t = 0$ )을 받아서 agent는 자신의 정책에 따라 행동(action)  $A_t$ 를 선택합니다. 그럼 environment는 agent로부터 action  $A_t$ 를 받아 보상(reward)  $R_{t+1}$ 과 새롭게 바뀐 상황(state)  $S_{t+1}$ 을 반환합니다. 그럼 agent는 다시 그것을 받아 action을 선택하게 됩니다. 따라서 아래와 같이 state, action reward가 시퀀스(sequence)로 주어지게 됩니다.

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, \dots$$

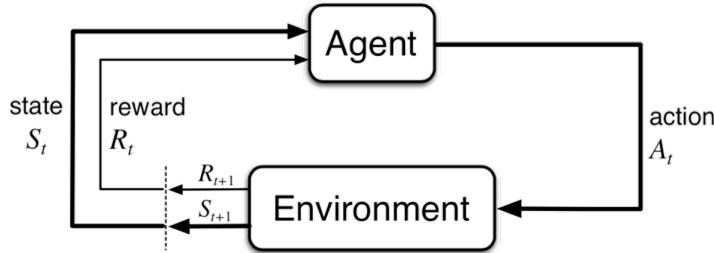


Figure 4:

특정 조건이 만족되면 environment는 이 시퀀스를 종료하고, 이를 하나의 에피소드(episode)라고 합니다. 반복되는 에피소드 하에서 agent를 강화학습을 통해 적절한 행동(보상을 최대로)을하도록 훈련하는 것이 우리의 목표입니다.

## Markov Decision Process

그리고 여기에 대해서 Markov Decision Process (MDP)라고 하는 성질을 도입합니다.

우리는 온세상의 현재(present)  $T = t$  이 순간을 하나의 상황(state)으로 정의하게 됩니다. 그럼 현재상황(present state)이 주어졌을 때, 미래  $T > t$ 는 과거  $T < t$ 로부터 독립(independent)이라고 가정 합니다. 그럼 이제 우리 세상은 Markov process(마코프 프로세스)상에서 움직인다고 할 수 있습니다. 이제 우리는 현재 상황에서 미래 상황으로 바뀔 확률을 아래와 같이 수식으로 표현할 수 있습니다.

$$P(S'|S)$$

여기서 Markov decision process(마코프 결정 프로세스)는 결정을 내리는 과정, 즉 행동을 선택하는 과정이 추가 된 것입니다. 풀어 설명하면, 현재 상황에서 어떤 행동을 선택 하였을 때 미래 상황으로 바뀔 확률이라고 할 수 있습니다. 따라서 그 수식은 아래와 같이 표현 할 수 있습니다.

$$P(S'|S, A)$$

이제 우리는 MDP 아래에서 environment(환경)와 agent(에이전트)가 state와 reward, action을 주고 받으며 나아가는 과정을 표현 할 수 있습니다.

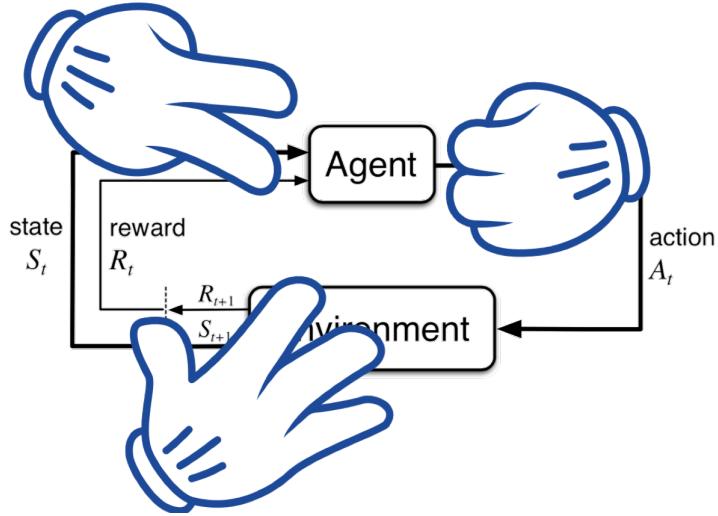


Figure 5:

## Reward

앞서, agent가 어떤 행동을 선택 하였을 때, 환경(environment)으로부터 보상(reward)을 받는다고 하였습니다. 이때 우리는  $G_t$ 를 어떤 시점으로부터 받은 보상의 총 합이라고 정의 합니다. 따라서  $G_t$ 는 아래와 같이 정의 됩니다.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

이때 우리는 discount factor  $\gamma$ 를 도입하여 수식을 다르게 표현 할 수도 있습니다.  $\gamma$ 는 0과 1 사이의 값으로, discount factor가 도입됨에 따라서 우리는 먼 미래의 보상보다 가까운 미래의 보상을 좀 더 중시해서 다룰 수 있게 됩니다.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

## Policy

Agent는 주어진 상황(state)에서 앞으로 받을 보상(reward)의 총 합을 최대로 하도록 행동해야 합니다. 마치 우리가 눈 앞의 즐거움을 참고 미래를 위해서 시험공부를

하듯이, 눈 앞의 작은 손해보다 면 미래까지의 보상의 총 합이 최대가 되는 것이 중요합니다. 따라서 agent는 어떠한 상황에서 어떻게 행동을 해야겠다라는 기준이 있어야 합니다.

사람도 상황에 따라서 즉흥적으로 임의의 행동을 선택하기보단 자신의 머릿속에 훈련되어 있는대로 행동하기 마련입니다. 무릎에 작은 충격이 왔을 때 다리를 들어올리는 '무릎반사'와 같은 무의식적인 행동에서부터, 파란불이 들어왔을 때 길을 건너는 행동, 어려운 수학 문제가 주어지면 답을 얻는 과정까지, 모두 주어진 상황에 대해서 행동해야 하는 기준이 있습니다.

정책(policy)은 이렇게 agent가 상황에 따라서 어떻게 행동을 해야 할 지 확률적으로 나타낸 기준입니다. 같은 상황이 주어졌을 때 항상 같은 행동만 반복하는 것이 아니라 확률적으로 행동을 선택한다고 할 수 있습니다. 물론 확률을 1로 표현하면 같은 행동만 반복하게 될 겁니다.

정책은 상황에 따른 가능한 행동에 대한 확률의 맵핑(mapping) 함수라고 할 수 있습니다. 수식으로는 아래와 같이 표현 합니다.

$$\pi(a|s) = P(A_t = a|S_t = s)$$

따라서 우리는 마음속의 정책에 따라 비가 오는 상황(state)에서 자장면과 짬뽕 중에 어떤 음식을 먹을지 확률적으로 선택 할 수 있고, 맑은 날에도 다른 확률 분포 중에서 선택 할 수 있습니다.

## Value Function

가치함수(value function)는 주어진 정책(policy)  $\pi$  아래에서 상황(state)  $s$ 에서부터 앞으로 얻을 수 있는 보상(reward) 총 합의 기대값을 표현합니다.  $v_\pi(s)$ 라고 표기하며, 아래와 같이 수식으로 표현 될 수 있습니다.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right], \forall s \in \mathcal{S}$$

앞으로 얻을 수 있는 보상의 총 합의 기대값은 기대누적보상(expected cumulative reward)라고 표현하기도 합니다.



Figure 6:

### Action–Value Function (Q–Function)

행동가치함수(action–value function)은 큐함수(Q–function)라고 불리기도 하며, 주어진 정책  $\pi$  아래 상황(state)  $s$ 에서 행동(action)  $a$ 를 선택 하였을 때 앞으로 얻을 수 있는 보상(reward)의 총 합의 기대값(expected cumulative reward, 기대누적보상)을 표현합니다. 가치함수는 어떤 상황  $s$ 에서 어떤 행동을 선택하는 것에 관계 없이 얻을 수 있는 누적 보상의 기대값이라고 한다면, 행동가치함수는 어떤 행동을 선택하는가에 대한 개념이 추가 된 것 입니다.

상황과 행동에 따른 기대누적보상을 나타내는 행동가치함수의 수식은 아래와 같습니다.

$$q_{\pi}(s, a) \doteq \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] = \mathbb{E}_{\pi} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right]$$

## Bellman Equation

Dynamic Programming

Policy Iteration

Policy Evaluation

$$\begin{aligned} v_{\pi}(s) &\doteq \mathbb{E}_{\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} P(s',r|s,a) \left[ r + \gamma v_{\pi}(s') \right] \end{aligned}$$

Policy Improvement

$$\begin{aligned} \pi'(s) &\doteq \operatorname{argmax}_a q_{\pi}(s,a) \\ &= \operatorname{argmax}_a \mathbb{E}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) | S_t = s, A_t = a] \\ &= \operatorname{argmax}_a \sum_{s',r} P(s',r|s,a) \left[ r + \gamma v_{\pi}(s') \right] \end{aligned}$$

Monte Carlo (MC) Methods

Temporal Difference (TD) Learning

Q-learning

우리는 올바른 행동가치함수를 알고 있다면, 어떠한 상황이 닥쳐도 항상 기대누적보상을 최대화(maximize)하는 행복한 선택을 할 수 있을 것 입니다. 따라서 행동가치함수를 잘 학습하는 것을 Q-learning(큐러닝)이라고 합니다.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ \overbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}^{\text{Target}} - \overbrace{Q(S_t, A_t)}^{\text{Current}} \right]$$

위 수식처럼 target과 current 사이의 차이를 줄이면, 결국 올바른 큐함수를 학습하게 될 것입니다.

### Deep Q-learning (DQN)

큐함수를 배울 때 상황(state) 공간과 행동(action) 공간이 너무 커서 상황과 행동이 희소한(sparse) 경우에는 문제가 생깁니다. 훈련 과정에서 희소성(sparseness)으로 인해 잘 볼 수 없기 때문입니다. 따라서 우리는 상황과 행동을 discrete한 별개의 값으로 다루되, 큐함수를 근사(approximation)함으로써 문제를 해결할 수 있습니다.

생각 해 보면, 아까 비가 올 때 자장면과 짬뽕을 선택하는 문제도, 비가 5mm가 오는것과 10mm가 오는 것은 비슷한 상황이며, 100mm 오는 것과는 상대적으로 다른 상황이라고 할 수 있습니다. 하지만 해가 짱짱한 맑은 날에 비해서 비가 5mm 오는 거소가 100mm 오는 것은 비슷한 상황이라고도 할 수 있습니다.

이처럼 상황과 행동을 근사하여 문제를 해결한다고 할 때, 신경망(neural network)은 매우 훌륭한 해결 방법이 될 수 있습니다. 딥마인드(DeepMind)는 신경망을 사용하여 근사한 큐러닝을 통해 아타리(Atari) 게임을 훌륭하게 플레이하는 강화학습 방법을 제시하였고, 이를 deep Q-learning (or DQN)이라고 이름 붙였습니다.

$$Q(S_t, A_t) \leftarrow \underbrace{Q(S_t, A_t)}_{\text{Approximated}} + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

위의 수식처럼 큐함수 부분을 신경망을 통해 근사(approximate)함으로써 희소성(sparseness)문제를 해결하였습니다. # Policy Based Reinforcement Learning

### Policy Gradients

Policy Gradients는 정책기반 강화학습(Policy based Reinforcement Learning) 방식에 속합니다. 알파고를 개발했던 DeepMind에 의해서 유명해진 Deep Q-Learning은 가치기반 강화학습(Value based Reinforcement Learning) 방식에 속합니다. 실제 딥러닝을 사용하여 두 방식을 사용 할 때에 가장 큰 차이점은, Value based방식은 인공신경망을 사용하여 어떤 action을 하였을 때에 얻을 수 있는 보상을 예측 하도록 훈련하는 것과 달리, policy based 방식은 인공신경망은 어떤 action을 할지 훈련되고 해당 action에 대한 보상(reward)를 back-propagation 할 때에

## Deep Reinforcement Learning in Atari

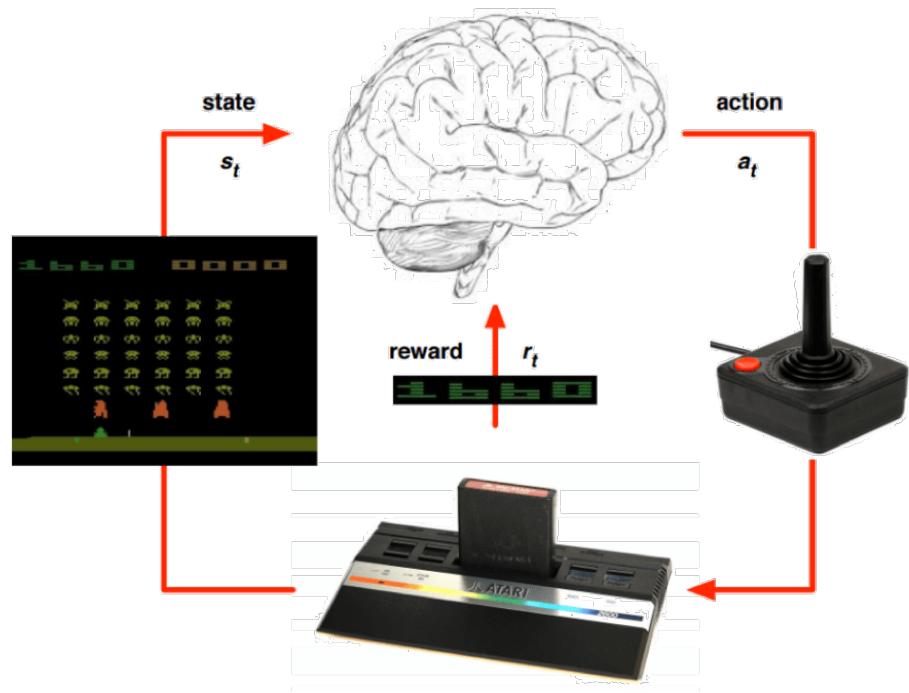


Figure 7:

gradient를 통해서 전달해 주는 것이 가장 큰 차이점입니다. 따라서 어떤 Deep Q-learning의 경우에는 action을 선택하는 것이 deterministic한 것에 비해서, Policy Gradient 방식은 action을 선택 할 때에 stochastic한 process를 거치게 됩니다. Policy Gradient에 대한 수식은 아래와 같습니다.

$$\pi_\theta(a|s) = P_\theta(a|s) = P(a|s; \theta)$$

위의  $\pi$ 는 정책(policy)을 의미합니다. 즉, 신경망  $\theta$ 는 현재 상황(state)  $s$ 가 주어졌을 때, 어떤 선택(action)  $a$ 를 해야할지 확률을 반환 합니다.

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}[r] = v_\theta(s_0) \\ &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \mathcal{R}_{s,a} \end{aligned}$$

우리의 목표(objective)는 최초 상황(initial state)에서의 기대누적보상(expected cumulative reward)을 최대(maximize)로 하도록 하는 policy( $\theta$ )를 찾는 것입니다. 최소화 하여야 하는 손실(loss)과 달리 보상(reward)은 최대화 하여야 하므로 기존의 gradient descent 대신에 gradient ascent를 사용하여 최적화(optimization)을 수행 합니다.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta)$$

Gradient ascent에 따라서,  $\nabla_\theta J(\theta)$ 를 구하여  $\theta$ 를 업데이트 해야 합니다. 여기서  $d(s)$ 는 markov chain의 stationary distribution으로써 시작점에 상관없이 전체의 trajecotry에서  $s$ 에 머무르는 시간의 proportion을 의미합니다.

$$\begin{aligned} \nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \end{aligned}$$

이때, 위의 로그 미분의 성질을 이용하여 아래와 같이  $\nabla_\theta J(\theta)$ 를 구할 수 있습니다. 이 수식을 해석하면 매 time-step 별 상황( $s$ )이 주어졌을 때 선택( $a$ )할 로그 확률의 gradient에, 그에 따른 보상(reward)을 곱한 값의 기대값이 됩니다.

$$\begin{aligned}
\nabla_{\theta} J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \nabla_{\theta} \pi_{\theta}(s, a) \mathcal{R}_{s,a} \\
&= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_{\theta}(s, a) \nabla_{\theta} \log \pi_{\theta}(s, a) \mathcal{R}_{s,a} \\
&= \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) r]
\end{aligned}$$

\_Policy Gradient Theorem\_에 따르면, 여기서 해당 time-step에 대한 즉각적인 reward( $r$ ) 대신에 episode의 종료까지의 총 reward, 즉  $Q$  function을 사용할 수 있습니다.

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi_{\theta}}(s, a)]$$

여기서 바로 Policy Gradients의 진가가 드러납니다. 우리는 policy network에 대해서 gradient를 구하지만,  $Q$ -function에 대해서는 gradient를 구할 필요가 없습니다. 즉, 미분의 가능 여부를 떠나서 임의의 어떠한 함수라도 보상 함수(reward function)로 사용할 수 있는 것입니다. 이렇게 어떠한 함수도 reward로 사용할 수 있게 됨에 따라, 기존의 단순히 cross entropy와 같은 손실 함수(loss function)에 학습(fitting) 시키는 대신에 좀 더 실제 문제에 부합하는 함수(번역의 경우에는 BLEU)를 사용하여  $\theta$ 를 훈련시킬 수 있게 되었습니다. 위의 수식에서 기대값 수식을 Monte Carlo sampling으로 대체하면 아래와 같이 parameter update를 수행 할 수 있습니다.

$$\theta \leftarrow \theta + \alpha Q^{\pi_{\theta}}(s_t, a_t) \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)$$

위의 수식을 좀 더 쉽게 설명 해 보면, Monte Carlo 방식을 통해 sampling 된 action들에 대해서 gradient를 구하고, 그 gradient에 reward를 곱하여 주는 형태입니다. 만약 샘플링 된 해당 action들이 좋은 (큰 양수) reward를 받았다면 learning rate  $\alpha$ 에 추가적인 곱셈을 통해서 더 큰 step으로 gradient ascending을 할 수 있을 겁니다. 하지만 negative reward를 받게 된다면, gradient의 반대방향으로 step을 갖도록 값이 곱해지게 될 겁니다. 따라서 해당 샘플링 된 action들이 앞으로는 잘 나오지 않도록 parameter  $\theta$ 가 update 됩니다.

따라서 실제 gradient에 따른 local minima(지역최소점)를 찾는 것이 아닌, 아래 그림과 같이 실제 reward-objective function에 따른 최적을 값을 찾게 됩니다. 하지만, 기존의 gradient는 방향과 크기를 나타낼 수 있었던 것에 비해서, policy gradients는 기존의 gradient의 방향에 크기(scalar)값을 곱해줌으로써 방향을 직접 지정해 줄 수는 없습니다. 따라서 실제 목적함수(objective function)에 따른 최적의

방향을 스스로 찾아갈 수는 없습니다. 그러므로 사실 훈련이 어렵고 비효율적인 단점을 갖고 있습니다.

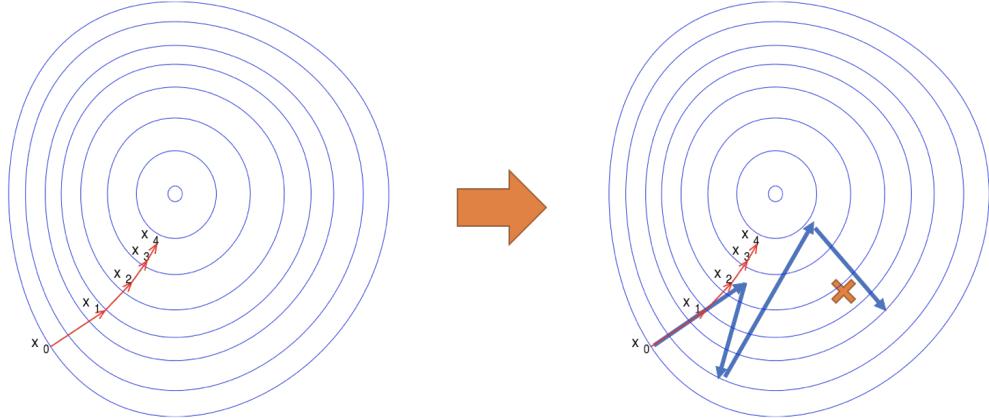


Figure 8:

Policy Gradient에 대한 자세한 설명은 원 논문인 [Sutton et al.1999], 또는 해당 저자가 쓴 텍스트북 Reinforcement Learning: An Introduction[Sutton et al.2017]을 참고하거나, DeepMind David Silver의 YouTube 강의를 참고하면 좋습니다.

### MLE vs RL(Policy Gradients)

여기서 reward의 역할을 좀 더 직관적으로 설명하고 넘어가도록 하겠습니다.

우리에게  $n$ 개의 시퀀스로 이루어진 입력을 받아,  $m$ 개의 시퀀스로 이루어진 출력을 하는 함수를 근사하는 것이 목표로 주어진다고 가정 해 봅니다. 그렇다면 시퀀스  $X$ 와  $Y$ 는  $\mathcal{B}$ 라는 dataset에 존재 합니다.

$$\begin{aligned}\mathcal{B} &= \{(X_i, Y_i)\}_{i=1}^N \\ X &= \{x_1, x_2, \dots, x_n\} \\ Y &= \{y_0, y_1, \dots, y_m\}\end{aligned}$$

근사하여 얻은 함수는 임의의 시퀀스  $X$ 가 주어졌을 때,  $\hat{Y}$ 를 반환하도록 잘 학습되어 있을 겁니다.

$$\hat{Y} = \operatorname{argmax}_Y P(Y|X)$$

그럼 해당 함수를 근사하기 위해서 우리는 parameter  $\theta$ 를 학습해야 합니다.  $\theta$ 는 아래와 같이 Maximum Likelihood Estimation(MLE)를 통해서 얻어질 수 있습니다.

$$\begin{aligned}\hat{\theta} &= \operatorname{argmax}_{\theta} P(\theta|X, Y) \\ &= \operatorname{argmax}_{\theta} P(Y|X; \theta)P(\theta)\end{aligned}$$

$\theta$ 에 대해 MLE를 수행하기 위해서 우리는 목적함수(objective function)을 아래와 같이 정의 합니다. 아래는 cross entropy loss를 목적함수로 정의 한 것 입니다. 우리의 목표는 목적함수를 최소화(minimize)하는 것입니다.

$$\begin{aligned}J(\theta) &= - \sum_{(X,Y) \in \mathcal{B}} P(Y|X) \log P(Y|X; \theta) \\ &= -\frac{1}{N} \sum_{(X,Y) \in \mathcal{B}} \sum_{i=0}^m \log P(y_i|X, y_{<i}; \theta)\end{aligned}$$

위의 수식에서  $P(Y|X)$ 는 훈련 데이터셋에 존재하므로 보통 1이라고 할 수 있습니다. 따라서 수식에서 생략 할 수 있습니다. 위에서 정의한 목적함수를 최소화 하여야 하기 때문에, gradient descent를 통해 지역최소점(local minima)를 찾아내어 전역최소점(global minima)에 근사(approximation)할 수 있습니다. 해당 수식은 아래와 같습니다.

$$\begin{aligned}\theta &\leftarrow \theta - \gamma \nabla J(\theta) \\ \theta &\leftarrow \theta + \gamma \frac{1}{N} \sum_{(X,Y) \in \mathcal{B}} \sum_{i=0}^m \nabla_{\theta} \log P(y_i|X, y_i; \theta)\end{aligned}$$

우리는 위의 수식에서 learning rate  $\gamma$ 를 통해 update step size를 조절 하는 것을 확인할 수 있습니다. 아래는 policy gradients에 기반하여 expected cumulative reward를 최대로 하는 gradient ascent 수식입니다. Reward를 최대화(maximization)해야 하기 때문에 gradient ascent를 사용하는 것을 볼 수 있습니다.

$$\begin{aligned}\theta &\leftarrow \theta + \alpha \nabla J(\theta) \\ \theta &\leftarrow \theta + \alpha Q^{\pi_{\theta}}(s_t, a_t) \nabla_{\theta} \log \pi_{\theta}(a_t|s_t)\end{aligned}$$

위의 수식에서도 이전 MLE의 gradient descent 수식과 마찬가지로,  $\alpha$ 와  $Q^{\pi_\theta}(s_t, a_t)$ 가 gradient 앞에 붙어서 learning rate 역할을 하는 것을 볼 수 있습니다. 따라서 reward에 따라서 해당 action들로부터 배우는 것을 더욱 강화하거나 반대방향으로 부정할 수 있는 것 입니다. 마치 좀 더 쉽게 비약적으로 설명하면 결과에 따라서 동적으로 learning rate를 알맞게 조절해 주는 것이라고 이해할 수 있습니다.

## REINFORCE with baseline

만약 위의 policy gradient를 수행 할 때, 보상이 항상 양수인 경우는 어떻게 동작할까요?

예를 들어 우리가 학교에서 100점 만점의 시험을 보았다고 가정해 보겠습니다. 시험 점수는 0점에서부터 100점까지 분포가 되어 평균 점수 근처에 있을 것입니다. 따라서 대부분의 학생들은 양의 보상을 받게 됩니다. 그럼 위의 기존 policy gradient는 항상 양의 보상을 받아 학생에게 박수쳐주며 해당 정책(policy)를 더욱 독려 할 것 입니다. 하지만 알고보면 평균점수 50점 일 때, 시험점수 10점은 매우 나쁜 점수라고 할 수 있습니다. 따라서 받수 받기보단 혼나서, 기존 정책(policy)의 반대방향으로 학습해야 합니다. 하지만 평균점수 50점일 때 시험점수 70점은 여전히 좋은 점수이고 박수 받아 마땅 합니다. 마찬가지로 평균 50점일 때 시험점수 90점은 70점보다 더 훌륭한 점수이고 박수갈채를 받아야 합니다.

주어진 상황에서 받아 마땅한 누적보상이 있기 때문에, 우리는 이를 바탕으로 현재 정책이 얼마나 훌륭한지 평가 할 수 있습니다. 이를 아래와 같이 policy gradient 수식으로 표현할 수 있습니다.

$$\theta \leftarrow \theta + \alpha \left( G_t - b(S_t) \right) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

## Reinforcement Learning on Natural Language Generation

### How to Apply

강화학습은 Markov Decision Process (MDP)상에서 정의되고 동작합니다. 따라서 여러 선택(action)들을 통해서 여러 상태(state)들을 옮겨다니며(transition) 에피소드가 구성되고, action과 state에 따라서 보상(reward)가 주어지며

누적(cumulated)되어, episode가 종료되면 누적보상(cumulative reward)를 얻을 수 있습니다.

따라서 이를 자연어처리에 적용하게 되면 텍스트 분류(text classification)와 같은 문제에 적용되기 보단 자연어생성(natural language generation)에 적용되게 됩니다. 이제까지 생성된 단어들의 시퀀스가 현재의 상황(state)이 될 것이며, 이제까지 생성된 단어를 기반으로 새롭게 선택하는 단어가 action이 될 것입니다. 이렇게 문장의 첫 단어(BOS, beginning of sentence)부터 문장의 끝 단어(EOS, end of sentence)까지 선택하는 과정(한 문장을 생성하는 과정)이 하나의 에피소드(episode)가 됩니다. 우리는 훈련 corpus에 대해서 문장을 생성하는 방법을 배워(즉, episode를 반복하여) 기대누적보상(expected cumulative reward)을 최대화(maximize)할 수 있도록 번역 신경망(강화학습에서는 정책망)  $\theta$ 를 훈련 하는 것입니다.

다시한번 정리를 위해, 기계번역에 강화학습을 대입시켜 보면, 현재의 상황(state)은 주어진 source 문장과 이전까지 생성(번역)된 단어들의 시퀀스가 될 것이고, action은 현재 state에 기반하여 새로운 단어를 선택 하는 것이 될 것입니다. 그리고 action을 선택하게 되면 다음 state는 deterministic하게 정해지게 됩니다. action을 선택한 후에 환경(environment)으로부터 즉각적인(immediate) 보상(reward)를 받지는 않으며, 모든 단어의 선택이 끝나고(EOS를 선택) 에피소드(episode)가 종료되면 BLEU 점수를 통해 보상을 받을 수 있습니다. 즉, 종료 시에 받는 보상값은 에피소드 누적보상값(cumulative reward)과 일치 합니다.

강화학습을 통해 모델을 훈련할 때, 훈련의 처음부터 강화학습만 적용하기에는 그 훈련방식이 비효율적이고 어려움이 크기 때문에, 보통 기존의 maximum likelihood estimation (MLE)를 통해 어느정도 학습이 된 신경망  $\theta$ 에 강화학습을 적용 합니다. – 강화학습은 탐험(exploration)을 통해 더 나은 정책의 가능성을 찾고, exploitation을 통해 그 정책을 발전시켜 나갑니다.

## Characteristics

우리는 이제까지 강화학습 중에서도 정책기반 학습 방식인 policy gradients에 대해서 간단히 다루어 보았습니다. 사실, policy gradients의 경우에도 소개한 방법 이외에도 발전된 방법들이 많이 있습니다. 예를 들어 Actor Critic의 경우에는 정책망( $\theta$ ) 이외에도 가치 네트워크( $W$ )를 따로 두어, episode의 종료까지 기다리지 않고 online으로 학습이 가능합니다. 여기에서 더욱 발전하여 기존의 단점을 보완한 A3C와 같은 다양한 방법들이 존재 합니다. – Reinforcement Learning: An Introduction[Sutton et al.2017]을 참고하세요.

하지만, 자연어처리에서의 강화학습은 이런 다양한 방법들을 굳이 사용하기보다는

간단한 REINFORCE with baseline를 사용하더라도 큰 문제가 없습니다. 이것은 자연어처리 분야의 특성에서 기인합니다. 강화학습을 자연어처리에 적용할 때는 아래와 같은 특성들이 있습니다.

1. 매우 많은 action들이 존재 합니다. 보통 다음 단어를 선택하는 것이 action이 되기 때문에, 선택 가능한 action set의 크기는 어휘(vocabulary) 사전의 크기와 같다고 볼 수 있습니다. 따라서 action set의 사이즈는 보통 몇만개가 되기 마련입니다.
2. 매우 많은 state들이 존재 합니다. 단어를 선택하는 것이 action이었다면, 이제까지 선택된 단어들의 시퀀스는 state가 되기 때문에, 여러 time-step을 거쳐 수많은 action(단어)들이 선택되었다면, 가능한 state의 경우의 수는 매우 커질 것입니다. –  $|state| = |action|^n$
3. 매우 많은 action들과 매우 많은 state들을 훈련 과정에서 모두 겪어보고 만나보는 것은 거의 불가능하다고 볼 수 있습니다. 또한, 추론(inference)과정에서 보지 못한(unseen) 샘플을 만나는 것은 매우 당연한 일이 될 것입니다. 따라서 이러한 희소성(sparseness)문제는 큰 골칫거리가 될 수 있습니다. 하지만 우리는 신경망(neural network)과 딥러닝을 통해서 이 문제를 해결 할 수 있습니다. – 이것은 Deep Reinforcement Learning이 큰 성공을 거두고 있는 이유이기도 합니다.
4. 강화학습을 자연어처리에 적용할 때에 쉬운 점도 있습니다. 대부분 하나의 문장을 생성하는 것이 하나의 에피소드(episode)가 되는데, 보통 문장의 길이는 길어봤자 80단어도 되기 힘들다는 것 입니다. 따라서 다른 분야의 강화학습보다 훨씬 쉬운 이점을 가지게 됩니다. 예를 들어 딥마인드(DeepMind)의 바둑(AlphaGo)이나 스타크래프트의 경우에는 하나의 에피소드가 끝나기까지 매우 긴 시간이 흐르게 됩니다. 따라서 에피소드 내에서 선택되었던 action들이 update되기 위해서는 매우 긴 에피소드가 끝나기를 기다려야 할 뿐만 아니라, 10분 전에 선택했던 action이 이 게임(game)의 승패에 얼마나 큰 영향을 미쳤는지 알아내는 것은 매우 어려운 일이 될 것입니다. 따라서, 자연어처리 분야가 다른 분야에 비해서 에피소드가 짧은 것은 매우 큰 이점으로 작용하여 정책망(policy network)을 훨씬 더 쉽게 훈련(training)시킬 수 있게 됩니다.
5. 대신에 문장 단위의 에피소드를 가진 강화학습에서는 보통 에피소드 중간에 reward를 얻기는 힘듭니다. 예를 들어 번역의 경우에는 각 time-step마다 단어를 선택할 때 즉각(immediate) reward를 얻지 못하고, 번역이 모두 끝난 이후 완성된 문장과 정답(reference) 문장을 비교하여 BLEU 점수를 누적 reward로 사용하게 됩니다. 마찬가지로 에피소드가 매우 길다면 이것은 매우 큰 문제가 되었겠지만, 다행히도 문장 단위의 에피소드 상에서는 큰 문제가 되지 않습니다.

이제 우리는 위의 특성들을 활용하여 강화학습을 성공적으로 기계번역에 적용한 방법들과 사례들을 살펴보고 실습 해 보도록 하겠습니다. # Supervised NMT

## Cross-entropy vs BLEU

$$L = -\frac{1}{|Y|} \sum_{y \in Y} P(y) \log P_\theta(y)$$

Cross entropy는 훌륭한 분류(classification) 문제에서 이미 훌륭한 손실함수(loss function)이지만 약간의 문제점을 가지고 있습니다. 자연어생성(NLG)을 위한 sequence-to-sequence의 훈련 과정에 적용하게 되면, 그 자체의 특성으로 인해서 우리가 평가하는 BLEU와의 괴리(discrepancy)가 생기게 됩니다. (자세한 내용은 이전 챕터 내용 참조 바랍니다.) 따라서 어찌보면 우리가 원하는 실제 기계번역의 목표(objective)와 달름으로 인해서 cross-entropy 자체에 오버피팅(over-fitting) 되는 효과가 생길 수 있습니다. 일반적으로 BLEU는 human evluation과 좋은 상관관계에 있다고 알려져 있지만, cross entropy는 이에 비해 낮은 상관관계를 가지기 때문입니다. 따라서 차라리 BLEU를 훈련 과정의 목적함수(objective function)로 사용하게 된다면 더 좋은 결과를 얻을 수 있을 것 입니다. 마찬가지로 다른 NLG 문제(요약 및 챗봇 등)에 대해서도 비슷한 접근을 생각 할 수 있습니다.

## Minimum Risk Training

위의 아이디어에서 출발한 논문[Shen et al.2015]<sup>o</sup>] Minimum Risk Training이라는 방법을 제안하였습니다. 이때에는 Policy Gradient를 직접적으로 사용하진 않았지만, 거의 비슷한 수식이 유도 되었다는 점이 매우 인상적입니다.

$$\hat{\theta}_{MLE} = argmin_\theta(\mathcal{L}(\theta))$$

where  $\mathcal{L}(\theta) = - \sum_{s=1}^S \log P(y^{(s)} | x^{(s)}; \theta)$

기존의 Maximum Likelihood Estimation (MLE)방식은 위와 같은 손실 함수(Loss function)를 사용하여  $|S|$ 개의 입력과 출력에 대해서 손실(loss)값을 구하고, 이를 최소화 하는  $\theta$ 를 찾는 것이 목표(objective)였습니다. 하지만 이 논문에서는 Risk를 아래와 같이 정의하고, 이를 최소화 하는 학습 방식을 Minimum Risk Training (MRT)라고 하였습니다.

$$\begin{aligned}\mathcal{R}(\theta) &= \sum_{s=1}^S E_{y|x^{(s)};\theta} [\Delta(y, y^{(s)})] \\ &= \sum_{s=1}^S \sum_{y \in \mathcal{Y}(\S^{(f)})} P(y|x^{(s)}; \theta) \Delta(y, y^{(s)})\end{aligned}$$

위의 수식에서  $\mathcal{Y}(x^{(s)})$ 는 full search space로써,  $s$ 번째 입력  $x^{(s)}$ 가 주어졌을 때, 가능한 정답의 집합을 의미합니다. 또한  $\Delta(y, y^{(s)})$ 는 입력과 파라미터( $\theta$ )가 주어졌을 때, sampling한  $y$ 와 실제 정답  $y^{(s)}$ 의 차이(error)값을 나타냅니다. 즉, 위 수식에 따르면 risk  $\mathcal{R}$ 은 주어진 입력과 현재 파라미터 상에서 얻은  $y$ 를 통해 현재 모델(함수)을 구하고, 동시에 이를 사용하여 risk의 기대값을 구한다고 볼 수 있습니다.

$$\hat{\theta}_{MRT} = \operatorname{argmin}_{\theta} (\mathcal{R}(\theta))$$

이렇게 정의된 risk를 최소화(minimize) 하도록 하는 것이 목표(objective)입니다. 사실 risk 대신에 reward로 생각하면, reward를 최대화(maximize) 하는 것이 목표가 됩니다. 결국은 risk를 최소화 할 때에는 gradient descent, reward를 최대화 할 때는 gradient ascent를 사용하게 되므로, 수식을 풀어보면 결국 완벽하게 같은 이야기라고 볼 수 있습니다. 따라서 실제 구현에 있어서는  $\Delta(y, y^{(s)})$  사용을 위해서 BLEU 점수에  $-1$ 을 곱하여 사용 합니다.

$$\begin{aligned}\tilde{\mathcal{R}}(\theta) &= \sum_{s=1}^S E_{y|x^{(s)};\theta,\alpha} [\Delta(y, y^{(s)})] \\ &= \sum_{s=1}^S \sum_{y \in \mathcal{S}(x^{(s)})} Q(y|x^{(s)}; \theta, \alpha) \Delta(y, y^{(s)})\end{aligned}$$

where  $\mathcal{S}(x^{(s)})$  is a sampled subset of the full search space  $\dagger(x^{(s)})$  and  $Q(y|x^{(s)}; \theta, \alpha)$  is a distribution defined on the subspace  $S(x^{(s)})$  :

$$Q(y|x^{(s)}; \theta, \alpha) = \frac{P(y|x^{(s)}; \theta)^{\alpha}}{\sum_{y' \in S(x^{(s)})} P(y'|x^{(s)}; \theta)^{\alpha}}$$

하지만 주어진 입력에 대한 가능한 정답에 대한 전체 space를 탐색(search)할 수는 없기 때문에, Monte Carlo를 사용하여 서브스페이스(sub-space)를

샘플링(sampling) 하는 것을 택합니다. 그리고 위의 수식에서  $\theta$ 에 대해서 미분을 수행합니다. 미분을 하여 얻은 수식은 아래와 같습니다.

$$\begin{aligned}\frac{\partial \tilde{R}(\theta)}{\partial \theta_i} &= \alpha \sum_{s=1}^S \mathbb{E}_{y|x^{(s)};\theta,\alpha} \left[ \frac{\partial P(y|x^{(s)};\theta)}{\partial \theta_i} \times (\Delta(y, y^{(s)}) - \mathbb{E}_{y'|x^{(s)};\theta,\alpha} [\Delta(y', y^{(s)})]) \right] \\ &= \alpha \sum_{s=1}^S \mathbb{E}_{y|x^{(s)};\theta,\alpha} \left[ \frac{\partial \log P(y|x^{(s)};\theta)}{\partial \theta_i} \times (\Delta(y, y^{(s)}) - \mathbb{E}_{y'|x^{(s)};\theta,\alpha} [\Delta(y', y^{(s)})]) \right] \\ &\approx \alpha \sum_{s=1}^S \frac{\partial \log P(y|x^{(s)};\theta)}{\partial \theta_i} \times (\Delta(y, y^{(s)}) - \frac{1}{K} \sum_{k=1}^K \Delta(y^{(k)}, y^{(s)}))\end{aligned}$$

$$\theta_{i+1} \leftarrow \theta_i - \frac{\partial \tilde{R}(\theta)}{\partial \theta_i}$$

이제 미분을 통해 얻은 MRT의 최종 수식을 해석 해 보겠습니다. 이해가 어렵다면 아래의 policy gradients 수식과 비교하며 따라가면 좀 더 이해가 수월할 수 있습니다.

- $s$ 번째 입력  $x^{(s)}$ 를 신경망  $\theta$ 에 넣어 얻은 로그확률  $\log P(y|x^{(s)};\theta)$ 을 미분하여 gradient를 얻습니다.
- 그리고  $\theta$ 로부터 샘플링(sampling) 한  $y$ 와 실제 정답  $y^{(s)}$ 와의 차이(여기서는 주로 BLEU에 -1을 곱하여 사용)값에서
- 또 다시  $\theta$ 로부터 샘플링하여 얻은  $y'$ 와 실제 정답  $y^{(s)}$ 와의 차이(마찬가지로 -BLEU)의 기대값을
- 빼 준 값을 risk로써 로그확률의 gradient에 곱해 줍니다.
- 이 과정을 전체 데이터셋(실제로는 mini-batch)  $S$ 에 대해서 수행한 후 합(summation)을 구하고 learning rate  $\alpha$ 를 곱 합니다.

최종적으로는 기대값 수식을 monte carlo sampling을 통해 제거할 수 있습니다.

아래는 policy gradients 수식입니다.

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) \times Q^{\pi_\theta}(s, a)] \\ \theta &\leftarrow \theta + \alpha \nabla_\theta J(\theta)\end{aligned}$$

MRT는 risk에 대해 minimize 해야 하기 때문에 gradient descent를 해 주는 것을 제외하면 똑같은 수식이 나오는 것을 알 수 있습니다.

System	Architecture	Training	Vocab	BLEU
<i>Existing end-to-end NMT systems</i>				
Bahdanau et al. (2015)	gated RNN with search	MLE	30K	28.45
Jean et al. (2015)	gated RNN with search		30K	29.97
Jean et al. (2015)	gated RNN with search + PosUnk		30K	33.08
Luong et al. (2015b)	LSTM with 4 layers		40K	29.50
Luong et al. (2015b)	LSTM with 4 layers + PosUnk		40K	31.80
Luong et al. (2015b)	LSTM with 6 layers		40K	30.40
Luong et al. (2015b)	LSTM with 6 layers + PosUnk		40K	32.70
Sutskever et al. (2014)	LSTM with 4 layers		80K	30.59
<i>Our end-to-end NMT systems</i>				
<i>this work</i>	gated RNN with search	MLE	30K	29.88
	gated RNN with search	MRT	30K	31.30
	gated RNN with search + PosUnk	MRT	30K	34.23

Table 7: Comparison with previous work on English-French translation. The BLEU scores are case-sensitive. “PosUnk” denotes Luong et al. (2015b)’s technique of handling rare words.

Figure 9:

위와 같이 훈련한 MRT에 대한 성능을 실험한 결과입니다. 기존의 MLE 방식에 비해서 BLEU가 1.5가량 상승한 것을 확인할 수 있습니다. 이처럼 MRT는 강화학습으로써의 접근을 전혀 하지 않고도, 수식적으로 policy gradients의 일종인 REINFORCE with baseline 수식을 이끌어내고 성능을 끌어올리는 방법을 제시한 점이 인상깊습니다.

## Implementation

우리는 아래의 방법을 통해 Minimum Risk Training을 PyTorch로 구현 할 겁니다.

1. 먼저 BLEU를 통해 얻은 reward에  $-1$ 을 곱해주어 risk로 변환 합니다.
2. 그리고 로그 확률에 risk를 곱해주고, 기존에 Negative Log Likelihood Loss (NLLLoss)를 사용했으므로 NLLLoss 값에  $-1$ 을 곱해주어 sum of positive log probability를 구합니다.
3. Summation 결과물에 대해서  $\theta$ 에 대해 미분을 수행하면, back-propagation을 통해서 신경망  $\theta$  전체에 gradient가 구해집니다.
4. 이 gradient를 사용하여 gradient descent를 통해 최적화(optimize)하도록 합니다.

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \sum_{s=1}^S \left( \log P(y|x^{(s)}; \theta) \times \left( \Delta(y, y^{(s)}) - \frac{1}{K} \sum_{k=1}^K \Delta(y^{(k)}, y^{(s)}) \right) \right)$$

where  $\Delta(\hat{y}, y) = -BLEU(\hat{y}, y)$

$$\theta \leftarrow \theta - \lambda \nabla_{\theta} J(\theta)$$

우리는 실험을 통해서 심지어  $K = 1$ 일 때도, MRT가 잘 동작함을 확인할 수 있습니다.

### Code

MRT(or RL)을 PyTorch를 사용하여 구현해 보도록 하겠습니다. 자세한 전체 코드는 이전의 NMT PyTorch 실습 코드의 git repository에서 다운로드 할 수 있습니다.

- git repo url: <https://github.com/kh-kim/simple-nmt>

### train.py

```
import argparse, sys

import torch
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from simple_nmt.seq2seq import Seq2Seq
import simple_nmt.trainer as trainer
import simple_nmt.rl_trainer as rl_trainer

def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True, help = 'Model file name to save. Ad')
    p.add_argument('-train', required = True, help = 'Training set file name except
```

```

p.add_argument('-valid', required = True, help = 'Validation set file name ex')
p.add_argument('-lang', required = True, help = 'Set of extention represents I')
p.add_argument('-gpu_id', type = int, default = -1, help = 'GPU ID to train. C')

p.add_argument('-batch_size', type = int, default = 32, help = 'Mini batch size')
p.add_argument('-n_epochs', type = int, default = 18, help = 'Number of epochs')
p.add_argument('-print_every', type = int, default = 1000, help = 'Number of g')
p.add_argument('-early_stop', type = int, default = -1, help = 'The training w')

p.add_argument('-max_length', type = int, default = 80, help = 'Maximum length')
p.add_argument('-dropout', type = float, default = .2, help = 'Dropout rate. D')
p.add_argument('-word_vec_dim', type = int, default = 512, help = 'Word embedde')
p.add_argument('-hidden_size', type = int, default = 768, help = 'Hidden size')
p.add_argument('-n_layers', type = int, default = 4, help = 'Number of layers')

p.add_argument('-max_grad_norm', type = float, default = 5., help = 'Threshold')
p.add_argument('-adam', action = 'store_true', help = 'Use Adam instead of us')
p.add_argument('-lr', type = float, default = 1., help = 'Initial learning rate')
p.add_argument('-min_lr', type = float, default = .000001, help = 'Minimum learn')
p.add_argument('-lr_decay_start_at', type = int, default = 10, help = 'Start l')
p.add_argument('-lr_slow_decay', action = 'store_true', help = 'Decay learning r')
p.add_argument('-lr_decay_rate', type = float, default = .5, help = 'Learning r')

p.add_argument('-rl_lr', type = float, default = .01, help = 'Learning rate fo')
p.add_argument('-n_samples', type = int, default = 1, help = 'Number of sample')
p.add_argument('-rl_n_epochs', type = int, default = 10, help = 'Number of epo')
p.add_argument('-rl_n_gram', type = int, default = 6, help = 'Maximum number o

config = p.parse_args()

return config

if __name__ == "__main__":
    config = define_argparser()

    import os.path
    # If the model exists, load model and configuration to continue the training
    if os.path.isfile(config.model):

```

```

    saved_data = torch.load(config.model)

    prev_config = saved_data['config']
    config = overwrite_config(config, prev_config)
    config.lr = saved_data['current_lr']

else:
    saved_data = None

# Load training and validation data set.
loader = DataLoader(config.train,
                     config.valid,
                     (config.lang[:2], config.lang[-2:]),
                     batch_size = config.batch_size,
                     device = config.gpu_id,
                     max_length = config.max_length
                     )

input_size = len(loader.src.vocab) # Encoder's embedding layer input size
output_size = len(loader.tgt.vocab) # Decoder's embedding layer input size and output size
# Declare the model
model = Seq2Seq(input_size,
                  config.word_vec_dim, # Word embedding vector size
                  config.hidden_size, # LSTM's hidden vector size
                  output_size,
                  n_layers = config.n_layers, # number of layers in LSTM
                  dropout_p = config.dropout # dropout-rate in LSTM
                  )

# Default weight for loss equals to 1, but we don't need to get loss for PAD
# Thus, set a weight for PAD to zero.
loss_weight = torch.ones(output_size)
loss_weight[data_loader.PAD] = 0.
# Instead of using Cross-Entropy loss, we can use Negative Log-Likelihood(NLL)
criterion = nn.NLLLoss(weight = loss_weight, size_average = False)

print(model)
print(criterion)

```

```

# Pass models to GPU device if it is necessary.
if config.gpu_id >= 0:
    model.cuda(config.gpu_id)
    criterion.cuda(config.gpu_id)

# If we have loaded model weight parameters, use that weights for declared m
if saved_data is not None:
    model.load_state_dict(saved_data['model'])

# Start training. This function maybe equivalent to 'fit' function in Keras.
trainer.train_epoch(model,
                     criterion,
                     loader.train_iter,
                     loader.valid_iter,
                     config,
                     start_epoch = saved_data['epoch'] if saved_data is not None
                     others_to_save = {'src_vocab': loader.src.vocab, 'tgt_vocab': loader.tgt.vocab})

# Start reinforcement learning.
if config.rl_n_epochs > 0:
    rl_trainer.train_epoch(model,
                           criterion, # Although it does not use cross-entropy loss
                           loader.train_iter,
                           loader.valid_iter,
                           config,
                           start_epoch = (saved_data['epoch'] - config.n_epochs)
                           others_to_save = {'src_vocab': loader.src.vocab, 'tgt_vocab': loader.tgt.vocab})

```

simple\_nmt/rl\_trainer.py

```

import time
import numpy as np
#from nltk.translate.bleu_score import sentence_bleu as score_func
from nltk.translate.bleu_score import sentence_bleu as score_func

import torch

```

```

import torch.nn as nn
import torch.optim as optim
import torch.nn.utils as torch_utils

import utils
import data_loader

def get_reward(y, y_hat, n_gram = 6):
    # This method gets the reward based on the sampling result and reference sentence
    # For now, we uses GLEU in NLTK, but you can used your own well-defined reward
    # In addition, GLEU is variation of BLEU, and it is more fit to reinforcement learning

    # Since we don't calculate reward score exactly as same as multi-bleu.perl,
    # (especialy we do have different tokenization,) I recommend to set n_gram to 6

    # |y| = (batch_size, length1)
    # |y_hat| = (batch_size, length2)

    scores = []

    # Actually, below is really far from parallized operations.
    # Thus, it may cause slow training.
    for b in range(y.size(0)):
        ref = []
        hyp = []
        for t in range(y.size(1)):
            ref += [str(int(y[b, t]))]
            if y[b, t] == data_loader.EOS:
                break

        for t in range(y_hat.size(1)):
            hyp += [str(int(y_hat[b, t]))]
            if y_hat[b, t] == data_loader.EOS:
                break

        # for nltk.bleu & nltk.gleu
        scores += [score_func([ref], hyp, max_len = n_gram) * 100.]

```

```

# for utils.score_sentence
#scores += [score_func(ref, hyp, 4, smooth = 1)[-1] * 100.]
scores = torch.FloatTensor(scores).to(y.device)
# |scores| = (batch_size)

return scores

def get_gradient(y, y_hat, criterion, reward = 1):
    # |y| = (batch_size, length)
    # |y_hat| = (batch_size, length, output_size)
    # |reward| = (batch_size)
    batch_size = y.size(0)

    # Before we get the gradient, multiply -reward for each sample and each time
    y_hat = y_hat * -reward.view(-1, 1, 1).expand(*y_hat.size())

    # Again, multiply -1 because criterion is NLLLoss.
    log_prob = -criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguous())
    log_prob.div(batch_size).backward()

    return log_prob

def train_epoch(model, criterion, train_iter, valid_iter, config, start_epoch = 1):
    current_lr = config.rl_lr

    highest_valid_bleu = -np.inf
    no_improve_cnt = 0

    # Print initial valid BLEU before we start RL.
    model.eval()
    total_reward, sample_cnt = 0, 0
    for batch_index, batch in enumerate(valid_iter):
        current_batch_word_cnt = torch.sum(batch.tgt[1])
        x = batch.src
        y = batch.tgt[0][:, 1:]
        batch_size = y.size(0)
        # |x| = (batch_size, length)
        # |y| = (batch_size, length)

```

```

# feed-forward
y_hat, indice = model.search(x, is_greedy = True, max_length = config.max_
# |y_hat| = (batch_size, length, output_size)
# |indice| = (batch_size, length)

reward = get_reward(y, indice, n_gram = config.rl_n_gram)

total_reward += float(reward.sum())
sample_cnt += batch_size
if sample_cnt >= len(valid_iter.dataset.examples):
    break
avg_bleu = total_reward / sample_cnt
print("initial valid BLEU: %.4f" % avg_bleu) # You can figure-out improvement
model.train() # Now, begin training.

# Start RL
for epoch in range(start_epoch, config.rl_n_epochs + 1):
    #optimizer = optim.Adam(model.parameters(), lr = current_lr)
    optimizer = optim.SGD(model.parameters(), lr = current_lr) # Default hyperparameter
    print("current learning rate: %f" % current_lr)
    print(optimizer)

    sample_cnt = 0
    total_loss, total_bleu, total_sample_count, total_word_count, total_parameters = 0, 0, 0, 0, 0
    start_time = time.time()
    train_bleu = np.inf

    for batch_index, batch in enumerate(train_iter):
        optimizer.zero_grad()

        current_batch_word_cnt = torch.sum(batch.tgt[1])
        x = batch.src
        y = batch.tgt[0][:, 1:]
        batch_size = y.size(0)
        # |x| = (batch_size, length)
        # |y| = (batch_size, length)

        # Take sampling process because set False for is_greedy.

```

```

y_hat, indice = model.search(x, is_greedy = False, max_length = config.max_length)
# Based on the result of sampling, get reward.
q_actor = get_reward(y, indice, n_gram = config.rl_n_gram)
# |y_hat| = (batch_size, length, output_size)
# |indice| = (batch_size, length)
# |q_actor| = (batch_size)

# Take samples as many as n_samples, and get average rewards for them.
# I figured out that n_samples = 1 would be enough.
baseline = []
with torch.no_grad():
    for i in range(config.n_samples):
        _, sampled_indice = model.search(x, is_greedy = False, max_length = config.max_length)
        baseline += [get_reward(y, sampled_indice, n_gram = config.rl_n_gram)]
    baseline = torch.stack(baseline).sum(dim = 0).div(config.n_samples)
    # |baseline| = (n_samples, batch_size) --> (batch_size)

# Now, we have relatively expected cumulative reward.
# Which score can be drawn from q_actor subtracted by baseline.
tmp_reward = q_actor - baseline
# |tmp_reward| = (batch_size)
# calculate gradients with back-propagation
get_gradient(indice, y_hat, criterion, reward = tmp_reward)

# simple math to show stats
total_loss += float(tmp_reward.sum())
total_bleu += float(q_actor.sum())
total_sample_count += batch_size
total_word_count += int(current_batch_word_cnt)
total_parameter_norm += float(utils.get_parameter_norm(model.parameters()))
total_grad_norm += float(utils.get_grad_norm(model.parameters()))

if (batch_index + 1) % config.print_every == 0:
    avg_loss = total_loss / total_sample_count
    avg_bleu = total_bleu / total_sample_count
    avg_parameter_norm = total_parameter_norm / config.print_every
    avg_grad_norm = total_grad_norm / config.print_every
    elapsed_time = time.time() - start_time

```



```

# |x| = (batch_size, length)
# |y| = (batch_size, length)

# feed-forward
y_hat, indice = model.search(x, is_greedy = True, max_length = config.max_length)
# |y_hat| = (batch_size, length, output_size)
# |indice| = (batch_size, length)

reward = get_reward(y, indice, n_gram = config.rl_n_gram)

total_reward += float(reward.sum())
sample_cnt += batch_size
if sample_cnt >= len(valid_iter.dataset.examples):
    break

avg_bleu = total_reward / sample_cnt
print("valid BLEU: %.4f" % avg_bleu)

if highest_valid_bleu < avg_bleu:
    highest_valid_bleu = avg_bleu
    no_improve_cnt = 0
else:
    no_improve_cnt += 1

model.train()

model_fn = config.model.split(".")
model_fn = model_fn[:-1] + [">%02d" % (config.n_epochs + epoch), "%.2f-%.4f"]

# PyTorch provides efficient method for save and load model, which uses
to_save = {"model": model.state_dict(),
            "config": config,
            "epoch": config.n_epochs + epoch + 1,
            "current_lr": current_lr
            }
if others_to_save is not None:
    for k, v in others_to_save.items():
        to_save[k] = v

```

```

    torch.save(to_save, f'{model_fn}'))

    if config.early_stop > 0 and no_improve_cnt > config.early_stop:
        break

```

## Unsupervised NMT

Supervised learning 방식은 높은 정확도를 자랑하지만 labeling 데이터가 필요하기 때문에 데이터 확보, 모델 및 시스템을 구축하는데 높은 비용과 시간이 소요됩니다. 하지만 Unsupervised Learning의 경우에는 데이터 확보에 있어서 훨씬 비용과 시간을 절감할 수 있기 때문에 좋은 대안이 될 수 있습니다.

### Parallel corpus vs Monolingual corpus

그러한 의미에서 parallel corpus에 비해서 확보하기 쉬운 monolingual corpus는 좋은 대안이 될 수 있습니다. 소량의 parallel corpus와 다량의 monolingual corpus를 결합하여 더 나은 성능을 확보할 수도 있을 것입니다. 이전 챕터에 다루었던 Back translation과 Copied translation에서 이와 관련하여 NMT의 성능을 고도화하는 방법을 보여주었습니다. 강화학습에서도 마찬가지로 unsupervised 방식을 적용하려는 시도들이 많이 보이고 있습니다. 다만, 대부분의 방식들은 아직 실제 field에서 적용하기에는 다소 효율성이 떨어집니다.

## Unsupervised NMT

위의 Dual Learning 논문과 달리 이 논문[Lample et al.2017]은 오직 Monolingual Corpus만을 사용하여 번역기를 제작하는 방법을 제안하였습니다. 따라서 Unsupervised NMT라고 할 수 있습니다.

이 논문의 핵심 아이디어는 아래와 같습니다. 제일 중요한 것은 encoder가 언어에 상관 없이 같은 내용일 경우에 같은 vector로 encoding할 수 있도록 훈련하는 것입니다. 이러한 encoder를 만들기 위해서 GAN이 도입되었습니다.

GAN을 NLP에 쓰지 못한다고 해 놓고 GAN을 썼다니 이게 무슨 소리인가 싶겠지만, encoder의 출력값인 vector에 대해서 GAN을 적용한 것이라 discrete한 값이 아닌 continuous한 값이기 때문에 가능한 것입니다.

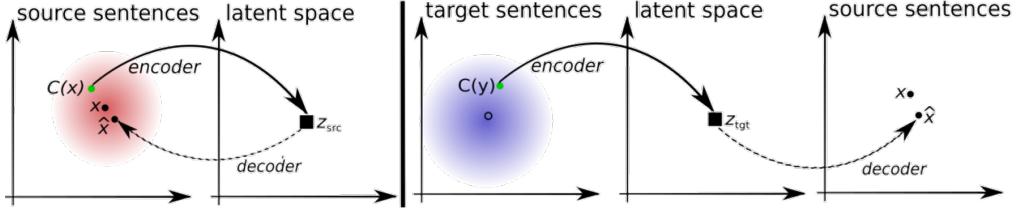


Figure 1: Toy illustration of the principles guiding the design of our objective function. Left (auto-encoding): the model is trained to reconstruct a sentence from a noisy version of it.  $x$  is the target,  $C(x)$  is the noisy input,  $\hat{x}$  is the reconstruction. Right (translation): the model is trained to translate a sentence in the other domain. The input is a noisy translation (in this case, from source-to-target) produced by the model itself,  $M$ , at the previous iteration ( $t$ ),  $y = M^{(t)}(x)$ . The model is symmetric, and we repeat the same process in the other language. See text for more details.

Figure 10:

이렇게 다른 언어일지라도 동일한 내용에 대해서는 같은 vector로 encoding하도록 훈련 된 encoder의 출력값을 가지고 decoder로 원래의 문장으로 잘 돌아오도록 해주는 것이 이 논문의 핵심 내용입니다.

특기 할 만한 점은 이 논문에서는 언어에 따라서 encoder와 decoder를 다르게 사용한 것이 아니라 언어에 상관없이 1개씩의 encoder와 decoder를 사용하였습니다. 또한 이 논문[Conneau et al., 2017]에서 제안한 word by word translation 방식으로 pre-training 한 모델을 사용합니다.

이 논문의 훈련은 3가지 관점에서 수행됩니다.

### Denoising Autoencoder

이전 챕터에서 다루었듯이 Seq2seq 모델도 결국 Autoencoder의 일종이라고 볼 수 있습니다. 그러한 관점에서 autoencoder(AE)로써 단순 복사(copy) task는 굉장히 쉬운 task에 속합니다. 그러므로 단순히 encoding 한 source sentence를 같은 언어의 문장으로 decoding 하는 것은 매우 쉬운 일이 될 것입니다. 따라서 AE에게 단순히 복사 작업을 지시하는 것이 아닌 noise를 섞어 준 source sentence에서 denoising을 하면서 reconstruction(복원)을 할 수 있도록 훈련해야 합니다. 따라서 이 task의 objective는 아래와 같습니다.

$$\mathcal{L}_{auto}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell) = \mathbb{E}_{x \sim \mathcal{D}_\ell, \hat{x} \sim d(e(C(x), \ell), \ell)} [\Delta(\hat{x}, x)]$$

$\hat{x} \sim d(e(C(x), \ell), \ell)$ 는 source sentence  $x$ 를  $C$ 를 통해 noise를 추가하고, 같은 언어  $\ell$ 로 encoding과 decoding을 수행한 것을 의미합니다.  $\Delta(\hat{x}, x)$ 는 원문과 복원된 문장과의 차이(error)를 나타냅니다.

### Noise Model

Noise Model  $C(x)$ 는 임의로 문장 내 단어들을 drop하거나, 순서를 섞어주는 일을 합니다. drop rate는 보통 0.1, 순서를 섞어주는 단어사이의 거리는 3정도가 적당한 것으로 설명 합니다.

### Cross Domain Training (Translation)

이번엔 이전 iteration의 모델  $M$ 에서 언어( $\ell_2$ )의 noisy translated된 문장( $y$ )을 다시 언어( $\ell_1$ ) source sentence로 원상복구 하는 task에 대한 objective입니다.

$$y = M(x)$$

$$\mathcal{L}_{cd}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_1, \ell_2) = \mathbb{E}_{x \sim \mathcal{D}_{\ell_1}, \hat{x} \sim d(e(C(y), \ell_2), \ell_1)} [\Delta(\hat{x}, x)]$$

### Adversarial Training

Encoder가 언어와 상관없이 항상 같은 분포로 hyper plane에 projection하는지 검사하기 위한 discriminator가 추가되어 Adversarial Training을 진행합니다.

Discriminator는 latent variable  $z$ 의 언어를 예측하여 아래의 cross-entropy loss를 minimize하도록 훈련됩니다.  $x_i, \ell_i$ 는 같은 언어(language pair)를 의미합니다.

$$\mathcal{L}_D(\theta_D | \theta, \mathcal{Z}) = -\mathbb{E}_{(x_i, \ell_i)} [\log p_D(\ell_i | e(x_i, \ell_i))]$$

따라서 encoder는 discriminator를 속일 수 있도록(fool) 훈련 되야 합니다.

$$\mathcal{L}_{adv}(\theta_{enc}, \mathcal{Z} | \theta_D) = -\mathbb{E}_{(x_i, \ell_i)} [\log p_D(\ell_j | e(x_i, \ell_i))]$$

where  $j = -(i - 1)$

위의 3가지 objective를 결합하면 Final Objective Function을 얻을 수 있습니다.

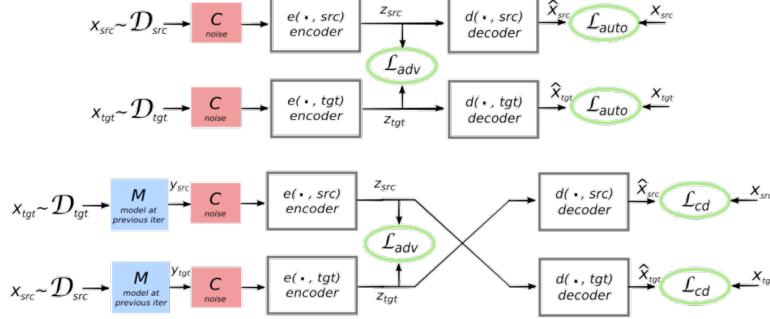


Figure 2: Illustration of the proposed architecture and training objectives. The architecture is a sequence to sequence model, with both encoder and decoder operating on two languages depending on an input language identifier that swaps lookup tables. Top (auto-encoding): the model learns to denoise sentences in each domain. Bottom (translation): like before, except that we encode from another language, using as input the translation produced by the model at the previous iteration (light blue box). The green ellipses indicate terms in the loss function.

Figure 11:

$$\begin{aligned} \mathcal{L}(\theta_{enc}, \theta_{dec}, \mathcal{Z}) = & \lambda_{auto} [\mathcal{L}_{auto}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{src}) + \mathcal{L}_{auto}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{tgt})] \\ & + \lambda_{cd} [\mathcal{L}_{cd}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{src}, \ell_{tgt}) + \mathcal{L}_{cd}(\theta_{enc}, \theta_{dec}, \mathcal{Z}, \ell_{tgt}, \ell_{src})] \\ & + \lambda_{adv} \mathcal{L}_{adv}(\theta_{enc}, \mathcal{Z} | \theta_D) \end{aligned}$$

$\lambda$ 를 통해서 선형결합(linear combination)을 취하여 기존의 손실함수에 추가 합니다. 이 과정을 pseudo code로 나타내면 아래와 같습니다.

이 논문에서 제안한 방식은 오직 단방향(monolingual) corpus만 존재할 때에 번역기를 만드는 방법에 대해서 다룹니다. 병렬(parallel) corpus가 없는 상황에서도 번역기를 만들 수 있다는 점은 매우 고무적이지만, 이 방법 자체만으로는 실제 필드에서 사용될 가능성은 적어 보입니다. 보통의 경우 필드에서 번역기를 구축하는 경우에는 병렬 corpus가 없는 경우는 드물고, 없다고 하더라도 단방향 corpus만으로 번역기를 구축하여 낮은 성능의 번역기를 확보하기보다는 비용을 들여 병렬 corpus를 직접 구축 한 후에, 병렬 corpus와 다수의 단방향 corpus를 합쳐 번역기를 구축하는 방향으로 나아갈 것이기 때문입니다.

---

**Algorithm 1** Unsupervised Training for Machine Translation

---

```
1: procedure TRAINING( $\mathcal{D}_{src}, \mathcal{D}_{tgt}, T$ )
2:   Infer bilingual dictionary using monolingual data (Conneau et al., 2017)
3:    $M^{(1)} \leftarrow$  unsupervised word-by-word translation model using the inferred dictionary
4:   for  $t = 1, T$  do
5:     using  $M^{(t)}$ , translate each monolingual dataset
6:     // discriminator training & model trainingl as in eq. 4
7:      $\theta_{discr} \leftarrow \arg \min \mathcal{L}_D, \quad \theta_{enc}, \theta_{dec}, \mathcal{Z} \leftarrow \arg \min \mathcal{L}$ 
8:      $M^{(t+1)} \leftarrow e^{(t)} \circ d^{(t)}$  // update MT model
9:   end for
10:  return  $M^{(t+1)}$ 
11: end procedure
```

---

Figure 12:

# Exploit Duality

## Exploit Duality

### What is Duality?

Duality란 무엇일까요? 우리가 보통 기계학습을 통해 학습하는 것은 어떤 도메인의 데이터  $X$ 를 받아서, 다른 도메인의 데이터  $Y$ 로 맵핑(mapping)해주는 함수를 근사(approximation)하는 것이라 할 수 있습니다. 따라서 대부분의 기계학습에 사용되는 데이터셋은 두 도메인 사이의 데이터로 구성되어있기 마련입니다.

Task( $D_1 \rightarrow D_2$ )	Domain 1	Domain 2	Task( $D_1 \leftarrow D_2$ )
기계번역	source 언어 문장	target 언어 문장	기계번역
음성인식	음성 신호	텍스트(transcript)	음성합성
이미지 분류	이미지	class	이미지 합성
요약	본문(content) 텍스트	제목(title) 텍스트	본문 생성

위와 같이 두 도메인 사이의 데이터의 관계를 배우는 방향에 따라서 음성인식이 되기도 하고, 음성합성이 되기도 합니다. 이러한 두 도메인 사이의 관계를 duality라고 우리는 정의 합니다. 대부분의 기계학습 문제들은 이와 같이 duality를 가지고 있는데, 특히 기계번역은 각 도메인의 데이터 간에 정보량의 차이가 거의 없는 것이 가장 큰 특징이자 장점입니다. 따라서 duality를 가장 적극적으로 활용할 수 있습니다.

## CycleGAN

먼저 좀 더 이해하기 쉬운 duality의 활용 예로, 컴퓨터 비전(Computer Vision)쪽 논문[Zhu et al. 2017]을 설명 해 볼까 합니다. Cycle GAN은 아래와 같이 unparallel image set이 여러개 있을 때, Set  $X$ 의 이미지를 Set  $Y$ 의 이미지로 합성/변환 시켜주는 방법입니다. 사진을 전체 구조는 유지하되 모네의 그림풍으로 바꾸어 주기도 하고, 말과 얼룩말을 서로 바꾸어 주기도 합니다. 겨울 풍경을 여름 풍경으로 바꾸어주기도 합니다.

아래에 이 방법을 도식화 하여 나타냈습니다. Set  $X$ 와 Set  $Y$  모두 각각 Generator( $G, F$ )와 Discriminator( $D_X, D_Y$ )를 가지고 있어서, min/max 게임을



Figure 1: Dr. Rico Sennrich

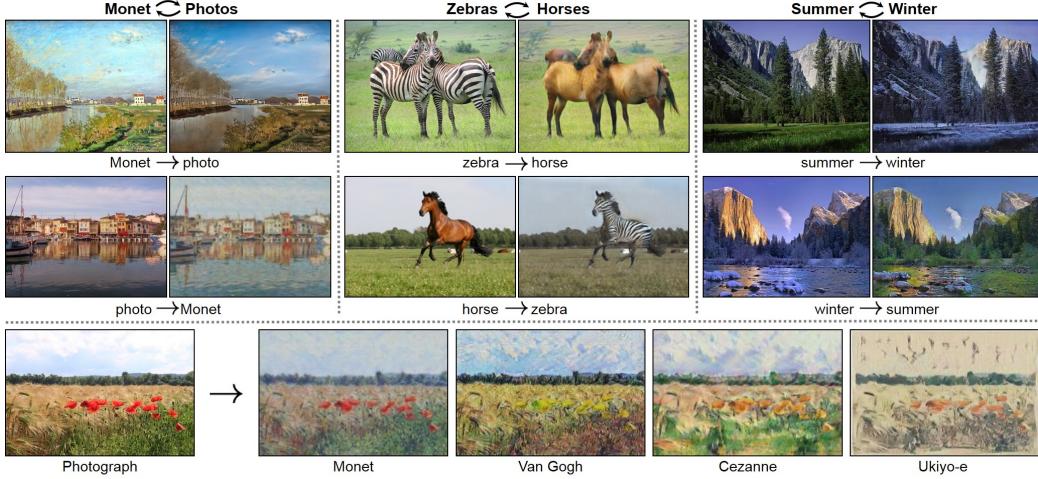


Figure 2: Cycle GAN – image from web

수행합니다.

$G$ 는  $x$ 를 입력으로 받아  $\hat{y}$ 으로 변환 해 냅니다. 그리고  $D_Y$ 는  $\hat{y}$  또는  $y$ 를 입력으로 받아 합성 유무(Real/Fake)를 판단 합니다. 마찬가지로  $F$ 는  $y$ 를 입력으로 받아  $\hat{x}$ 으로 변환 합니다. 이후에  $D_X$ 는  $\hat{x}$  또는  $x$ 를 입력으로 받아 합성 유무를 판단 합니다.

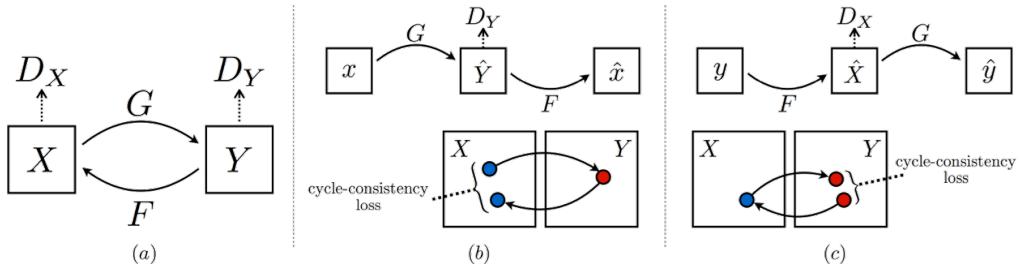


Figure 3: (a) Our model contains two mapping functions  $G : X \rightarrow Y$  and  $F : Y \rightarrow X$ , and associated adversarial discriminators  $D_Y$  and  $D_X$ .  $D_Y$  encourages  $G$  to translate  $X$  into outputs indistinguishable from domain  $Y$ , and vice versa for  $D_X$  and  $F$ . To further regularize the mappings, we introduce two *cycle consistency losses* that capture the intuition that if we translate from one domain to the other and back again we should arrive at where we started: (b) forward cycle-consistency loss:  $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$ , and (c) backward cycle-consistency loss:  $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$

Figure 3:

이 방식의 핵심은  $\hat{x}$ 나  $\hat{y}$ 를 합성 할 때에 기존의 Set  $X, Y$ 에 속하는 것 처럼 만들어내야 한다는 것 입니다. 이것을 기계번역에 적용 시켜 보면 어떻게 될까요?

## Dual Supervised Learning

이번에 소개할 방법은 Dual Supervised Learning (DSL) [Xia et al.2017] 입니다. 이 방법은 기존의 Teacher Forcing의 문제로 생기는 어려움을 강화학습을 사용하지 않고, Duality로부터 regularization term을 이끌어내어 해결하였습니다.

베이즈 정리(Bayes Theorem)에 따라서 우리는 아래의 수식이 언제나 성립함을 알고 있습니다.

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

$$P(Y|X)P(X) = P(X|Y)P(Y)$$

따라서 위의 수식을 따라서, 우리의 데이터셋을 통해 훈련한 모델들은 아래와 같은 수식을 만족해야 합니다.

$$P(x)P(y|x; \theta_{x \rightarrow y}) = P(y)P(x|y; \theta_{y \rightarrow x})$$

이 전제를 우리의 번역 훈련을 위한 목표에 적용하면 다음과 같습니다.

$$\text{objective1} : \min_{\theta_{x \rightarrow y}} \frac{1}{n} \sum_{i=1}^n \ell_1(f(x_i; \theta_{x \rightarrow y}), y_i),$$

$$\text{objective2} : \min_{\theta_{y \rightarrow x}} \frac{1}{n} \sum_{i=1}^n \ell_1(g(y_i; \theta_{y \rightarrow x}), x_i),$$

$$\text{s.t. } P(x)P(y|x; \theta_{x \rightarrow y}) = P(y)P(x|y; \theta_{y \rightarrow x}), \forall x, y.$$

위의 수식을 해석하면, 목표(objective1)은 베이즈 정리에 따른 제약조건을 만족함과 동시에,  $\ell_1$ 을 최소화(minimize) 하도록 해야 합니다.  $\ell_1$ 은 번역함수  $f$ 에 입력  $x_i$ 를 넣어 나온 반환값과  $y_i$  사이의 손실(loss)를 의미 합니다. 마찬가지로,  $\ell_2$ 도 번역함수  $g$ 에 대해 같은 작업을 수행하고 최소화하여 목표(objective2)를 만족해야 합니다.

$$\mathcal{L}_{duality} = ((\log \hat{P}(x) + \log P(y|x; \theta_{x \rightarrow y})) - (\log \hat{P}(y) + \log P(x|y; \theta_{y \rightarrow x})))^2$$

그러므로 우리는  $\mathcal{L}_{duality}$ 와 같이 베이즈 정리에 따른 제약조건의 양 변의 값의 차이를 최소화(minimize)하도록 하는 MSE 손실함수(loss function)을 만들 수

있습니다. 위의 수식에서 우리가 동시에 훈련시키는 신경망 네트워크 파라미터를 통해  $\log P(y|x; \theta_{x \rightarrow y})$ 와  $\log P(x|y; \theta_{y \rightarrow x})$ 를 구하고, 단방향(monolingual) corpus를 통해 별도로 이미 훈련시켜 놓은 언어모델을 통해  $\log \hat{P}(x)$ 와  $\log \hat{P}(y)$ 를 근사(approximation)할 수 있습니다.

이 부가적인 제약조건의 손실함수를 기존의 목적함수(objective function)에 추가하여 동시에 minimize 하도록 하면, 아래와 같이 표현 할 수 있습니다.

$$\begin{aligned}\theta_{x \rightarrow y} &\leftarrow \theta_{x \rightarrow y} - \gamma \nabla_{\theta_{x \rightarrow y}} \frac{1}{n} \sum_{j=1}^m [\ell_1(f(x_i; \theta_{x \rightarrow y}), y_i) + \lambda_{x \rightarrow y} \mathcal{L}_{duality}] \\ \theta_{y \rightarrow x} &\leftarrow \theta_{y \rightarrow x} - \gamma \nabla_{\theta_{y \rightarrow x}} \frac{1}{n} \sum_{j=1}^m [\ell_2(g(y_i; \theta_{y \rightarrow x}), x_i) + \lambda_{y \rightarrow x} \mathcal{L}_{duality}]\end{aligned}$$

여기서  $\lambda$ 는 Lagrange multipliers로써, 고정된 값의 hyper-parameter입니다. 실험 결과  $\lambda = 0.01$  일 때, 가장 좋은 성능을 나타낼 수 있었습니다.

*Table 2. Summary of some existing En→Fr translations*

Model	Brief description	BLEU
NMT[1]	<i>standard NMT</i>	33.08
MRT[2]	<i>Direct optimizing BLEU</i>	34.23
DSL	<i>Refer to Algorithm 1</i>	<b>34.84</b>
[1] (Jean et al., 2015); [2] (Shen et al., 2016)		

Figure 4:

위의 테이블과 같이 기존의 Teacher Forcing 아래의 cross entropy 방식([1]번)과 Minimum Risk Training(MRT) 방식([2]번) 보다 더 높은 성능을 보입니다.

이 방법은 강화학습과 같이 비효율적이고 훈련이 까다로운 방식을 벗어나서 regularization term을 추가하여 강화학습을 상회하는 성능을 얻어낸 것이 주목할 점이라고 할 수 있습니다.

# Dual Unsupervised Learning

## Dual Learning for Machine Translation

공교롭게도 CycleGAN과 비슷한 시기에 나온 논문[Xia et al.2016]이 있습니다. NLP의 특성상 CycleGAN처럼 직접적으로 gradient를 전달해 줄 수는 없었지만 기본적으로는 아주 비슷한 개념입니다. 짹이 없는 단방향(monolingual) corpus를 이용하여 성능을 극대화 하고자 하였습니다.

즉, monolingual sentence( $s$ )에 대해서 번역을 하고 그 문장( $s_{mid}$ )을 사용하여 복원을 하였을 때( $\hat{s}$ ) 원래의 처음 문장으로 돌아올 수 있도록(처음 문장과의 차이를 최소화 하도록) 훈련하는 것입니다. 이때, 번역된 문장  $s_{mid}$ 는 자연스러운 해당 언어의 문장이 되었는가도 중요한 지표가 됩니다.

위에서 설명한 알고리즘을 따라가 보겠습니다. 이 방법에서는 Set  $X$ , Set  $Y$  대신에 Language  $A$ , Language  $B$ 로 표기하고 있습니다.  $G_{A \rightarrow B}$ 의 파라미터  $\theta_{AB}$ 와  $F_{B \rightarrow A}$ 의 파라미터  $\theta_{BA}$ 가 등장합니다. 이  $G_{A \rightarrow B}, F_{B \rightarrow A}$ 는 모두 parallel corpus에 의해서 pre-training이 되어 있는 상태입니다. 즉, 기본적인 저성능의 번역기 수준이라고 가정합니다.

우리는 기존의 policy gradient와 마찬가지로 아래와 같은 파라미터 업데이트를 수행해야 합니다.

$$\begin{aligned}\theta_{AB} &\leftarrow \theta_{AB} + \gamma \nabla_{\theta_{AB}} \hat{\mathbb{E}}[r] \\ \theta_{BA} &\leftarrow \theta_{BA} + \gamma \nabla_{\theta_{BA}} \hat{\mathbb{E}}[r]\end{aligned}$$

$\hat{\mathbb{E}}[r]$ 을 각각의 파라미터에 대해서 미분 해 준 값을 더해주는 것을 볼 수 있습니다. 이 reward의 기대값은 아래와 같이 구할 수 있습니다.

$$\begin{aligned}r &= \alpha r_{AB} + (1 - \alpha)r_{BA} \\ r_{AB} &= LM_B(s_{mid}) \\ r_{BA} &= \log P(s|s_{mid}; \theta_{BA})\end{aligned}$$

위와 같이  $k$ 개의 sampling한 문장에 대해서 각기 방향에 대한 reward를 각각 구한 후, 이를 선형 결합(linear combination)을 취해줍니다. 이때,  $s_{mid}$ 는 sampling한 문장을 의미하고,  $LM_B$ 를 사용하여 해당 문장이 language  $B$ 의 집합에 잘 어울리는지를 따져 reward로 리턴합니다. 여기서 기존의 cross entropy를 사용할 수 없는 이유는

---

**Algorithm 1** The dual-learning algorithm

---

1: **Input:** Monolingual corpora  $D_A$  and  $D_B$ , initial translation models  $\Theta_{AB}$  and  $\Theta_{BA}$ , language models  $LM_A$  and  $LM_B$ , hyper-parameter  $\alpha$ , beam search size  $K$ , learning rates  $\gamma_{1,t}, \gamma_{2,t}$ .

2: **repeat**

3:      $t = t + 1$ .

4:     Sample sentence  $s_A$  and  $s_B$  from  $D_A$  and  $D_B$  respectively.

5:     Set  $s = s_A$ .  $\triangleright$  Model update for the game beginning from A.

6:     Generate  $K$  sentences  $s_{mid,1}, \dots, s_{mid,K}$  using beam search according to translation model  $P(\cdot|s; \Theta_{AB})$ .

7:     **for**  $k = 1, \dots, K$  **do**

8:         Set the language-model reward for the  $k$ th sampled sentence as  $r_{1,k} = LM_B(s_{mid,k})$ .

9:         Set the communication reward for the  $k$ th sampled sentence as  $r_{2,k} = \log P(s|s_{mid,k}; \Theta_{BA})$ .

10:        Set the total reward of the  $k$ th sample as  $r_k = \alpha r_{1,k} + (1 - \alpha)r_{2,k}$ .

11:     **end for**

12:     Compute the stochastic gradient of  $\Theta_{AB}$ :

$$\nabla_{\Theta_{AB}} \hat{E}[r] = \frac{1}{K} \sum_{k=1}^K [r_k \nabla_{\Theta_{AB}} \log P(s_{mid,k}|s; \Theta_{AB})].$$

13:     Compute the stochastic gradient of  $\Theta_{BA}$ :

$$\nabla_{\Theta_{BA}} \hat{E}[r] = \frac{1}{K} \sum_{k=1}^K [(1 - \alpha) \nabla_{\Theta_{BA}} \log P(s|s_{mid,k}; \Theta_{BA})].$$

14:     Model updates:

$$\Theta_{AB} \leftarrow \Theta_{AB} + \gamma_{1,t} \nabla_{\Theta_{AB}} \hat{E}[r], \Theta_{BA} \leftarrow \Theta_{BA} + \gamma_{2,t} \nabla_{\Theta_{BA}} \hat{E}[r].$$

15:     Set  $s = s_B$ .  $\triangleright$  Model update for the game beginning from B.

16:     Go through line 6 to line 14 symmetrically.

17: **until** convergence

---

Figure 5:

monolingual sentence이기 때문에 번역을 하더라도 정답을 알 수 없기 때문입니다. 또한 우리는 다수의 단방향(monolingual) corpus를 갖고 있기 때문에,  $LM$ 은 쉽게 만들어낼 수 있습니다.

$$\begin{aligned}\nabla_{\theta_{AB}} \hat{\mathbb{E}}[r] &= \frac{1}{K} \sum_{k=1}^K [r_k \nabla_{\theta_{AB}} \log P(s_{mid,k}|s; \theta_{AB})] \\ \nabla_{\theta_{BA}} \hat{\mathbb{E}}[r] &= \frac{1}{K} \sum_{k=1}^K [(1 - \alpha) \nabla_{\theta_{BA}} \log P(s|s_{mid,k}; \theta_{BA})]\end{aligned}$$

이렇게 얻어진  $\mathbb{E}[r]$ 를 각 파라미터에 대해서 미분하게 되면 위와 같은 수식을 얻을 수 있고, 상기 서술한 파라미터 업데이트 수식에 대입하면 됩니다. 비슷한 방식으로  $B \rightarrow A$ 를 구할 수 있습니다.

Table 1: Translation results of En↔Fr task. The results of the experiments using all the parallel data for training are provided in the first two columns (marked by “Large”), and the results using 10% parallel data for training are in the last two columns (marked by “Small”).

	En→Fr (Large)	Fr→En (Large)	En→Fr (Small)	Fr→En (Small)
NMT	29.92	27.49	25.32	22.27
pseudo-NMT	30.40	27.66	25.63	23.24
dual-NMT	<b>32.06</b>	<b>29.78</b>	<b>28.73</b>	<b>27.50</b>

Figure 6:

위의 테이블은 이 방법의 성능을 비교한 결과입니다. Pseudo-NMT는 이전 챕터에서 설명하였던 back-translation을 의미합니다. 그리고 그 방식보다 더 좋은 성능을 기록한 것을 볼 수 있습니다.

또한, 위 그래프에서 문장의 길이와 상관 없이 모든 구간에서 baseline NMT를 성능으로 압도하고 있는 것을 알 수 있습니다. 다만, 병렬(parallel) corpus의 양이 커질수록 단방향(monolingual) corpus에 의한 성능 향상의 폭이 줄어드는 것을 확인 할 수 있습니다.

이 방법은 강화학습과 Duality를 접목하여 적은 양의 병렬(parallel) corpus와 다수의 단방향(monolingual) corpus를 활용하여 번역기의 성능을 효과적으로 끌어올리는 방법을 제시하였다는 점에서 주목할 만 합니다.

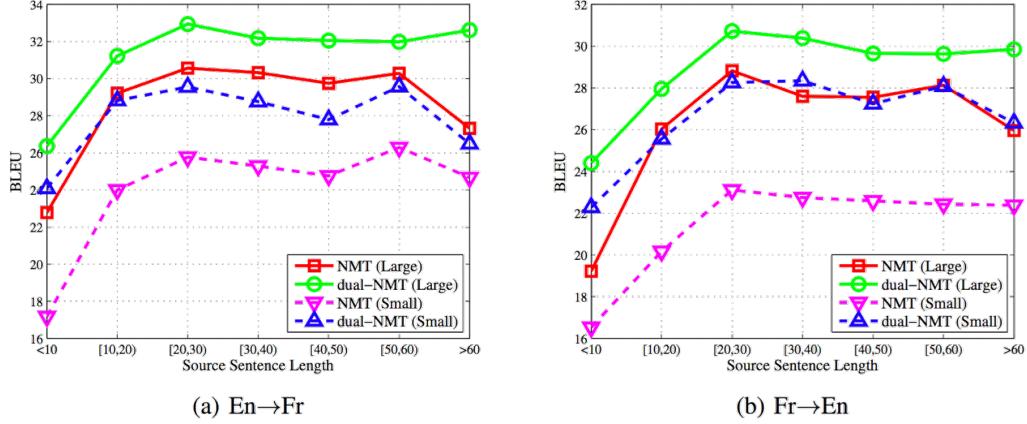


Figure 1: BLEU scores w.r.t lengths of source sentences

Figure 7:

## Dual Transfer Learning for NMT with Marginal Distribution Regularization

Dual Supervised Learning (DSL)은 베이즈 정리에 따른 수식을 제약조건으로 사용하였다면, 이 방법[Wang et al.2017]은 Marginal 분포(distribution)의 성질을 이용하여 제약조건을 만듭니다.

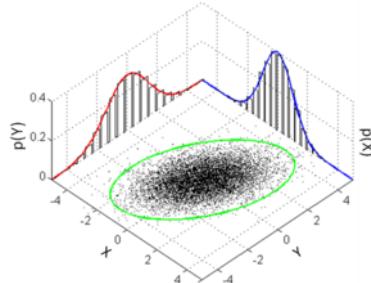


Figure 8: Marginal Distribution from Wikipedia

$$P(y) = \sum_{x \in \mathcal{X}} P(x, y) = \sum_{x \in \mathcal{X}} P(y|x)P(x)$$

Marginal 분포는 결합확률분포(joint distribution)를 어떤 한 variable에 대해서 합

또는 적분 한 것을 이릅니다. 이것을 조건부확률로 나타낼 수 있고, 여기서 한발 더 나아가 기대값 표현으로 바꿀 수 있습니다. 그리고 이를  $K$ 번 샘플링 하도록 하여 Monte Carlo로 근사 표현 할 수 있습니다.

$$\begin{aligned} P(y) &= \sum_{x \in \mathcal{X}} P(y|x; \theta) P(x) = \mathbb{E}_{x \sim P(x)} P(y|x; \theta) \\ &\approx \frac{1}{K} \sum_{i=1}^K P(y|x^i; \theta), \quad x^i \sim P(x) \end{aligned}$$

이제 위의 수식을 기계번역에 적용해 보도록 하겠습니다. 우리에게 아래와 같이  $N$ 개의 source 문장  $x$ , target 문장  $y$ 으로 이루어진 양방향 corpus  $\mathcal{B}$ ,  $S$ 개의 target 문장  $y$ 로만 이루어진 단방향 corpus  $\mathcal{M}$ 이 있다고 가정 해 보겠습니다.

$$\begin{aligned} \mathcal{B} &= \{(x^n, y^n)\}_{n=1}^N \\ \mathcal{M} &= \{y^s\}_{s=1}^S \end{aligned}$$

그럼 우리는 아래의 목적함수(objective function)을 최대화(maximize)하는 동시에 marginal 분포에 따른 제약조건 또한 만족시켜야 합니다.

$$\begin{aligned} \text{Objective} : & \sum_{n=1}^N \log P(y^n|x^n; \theta), \\ \text{s.t. } & P(y) = \mathbb{E}_{x \sim P(x)} P(y|x; \theta), \forall y \in \mathcal{M}. \end{aligned}$$

위의 수식을 DSL과 마찬가지로 Lagrange multipliers와 함께 기존의 손실함수(loss function)에 추가하여 주기 위하여  $S(\theta)$ 와 같이 표현합니다.

$$S(\theta) = [\log \hat{P}(y) - \log \mathbb{E}_{x \sim \hat{P}(x)} P(y|x; \theta)]^2$$

$$\mathcal{L}(\theta) = - \sum_{n=1}^N \log P(y^n|x^n; \theta) + \lambda \sum_{s=1}^S [\log \hat{P}(y) - \log \mathbb{E}_{x \sim \hat{P}(x)} P(y|x; \theta)]^2$$

이때, DSL과 유사하게  $\hat{P}(x)$ 와  $\hat{P}(y)$ 가 등장합니다.  $\hat{P}(y)$ 는 단방향(monolingual) corpus로 만든 언어모델(language model)을 통해 확률값을 구합니다. 위의 수식에

따르면  $\hat{P}(x)$ 를 통해 source 문장  $x$ 를 샘플링(sampling)하여 네트워크  $\theta$ 를 통과시켜  $P(y|x; \theta)$ 를 구해야겠지만, 아래와 같이 좀 더 다른 방법으로 접근합니다.

$$\begin{aligned}
P(y) &= \mathbb{E}_{x \sim \hat{P}(x)} P(y|x; \theta) = \sum_{x \in \mathcal{X}} P(y|x; \theta) \hat{P}(x) \\
&= \sum_{x \in \mathcal{X}} \frac{P(y|x; \theta) \hat{P}(x)}{P(x|y)} P(x|y) \\
&= \mathbb{E}_{x \sim P(x|y)} \frac{P(y|x; \theta) \hat{P}(x)}{P(x|y)} \\
&= \frac{1}{K} \sum_{i=1}^K \frac{P(y|x_i; \theta) \hat{P}(x_i)}{P(x_i|y)}, x_i \sim P(x|y)
\end{aligned}$$

위와 같이 target 문장  $y$ 를 반대 방향 번역기( $y \rightarrow x$ )에 넣어  $K$ 개의 source 문장  $x$ 를 샘플링(sampling)하여  $P(y)$ 를 구합니다. 이 과정을 다시 하나의 손실함수(loss function)으로 표현하면 아래와 같습니다.

$$\mathcal{L}(\theta) \approx - \sum_{n=1}^N \log P(y^n|x^n; \theta) + \lambda \sum_{s=1}^S [\log \hat{P}(y^s) - \log \frac{1}{K} \sum_{i=1}^K \frac{\hat{P}(x_i^s) P(y^s|x_i^s \theta)}{P(x_i^s|y^s)}]^2$$

Table 1: BLEU scores on En→Fr and De→En translation tasks.  $\Delta$  means the improvement over the basic NMT model, which only used bilingual data for training. The basic model for En→Fr is the RNNSearch model (Bahdanau, Cho, and Bengio 2015), and for De→En is a two-layer LSTM model. Note that all the methods for the same task share the same model structure.

System	En→Fr	$\Delta$	De→En	$\Delta$
Basic model	29.92		30.99	
<i>Representative semi-supervised NMT systems</i>				
Shallow fusion-NMT (Gulcehre et al. 2015)	30.03	+0.11	31.08	+0.09
Pseudo-NMT (Sennrich, Haddow, and Birch 2016)	30.40	+0.48	31.76	+0.77
Dual-NMT (He et al. 2016a)	32.06	+2.14	32.05	+1.06
<i>Our dual transfer learning system</i>				
This work	<b>32.85</b>	<b>+2.93</b>	<b>32.35</b>	<b>+1.36</b>

Figure 9:

위의 테이블과 같이, 이 방법은 앞 챕터에서 소개한 기존의 단방향 corpus[Gulcehre et al.2015][Sennrich et al.2016]를 활용한 방식들과 비교하여 훨씬 더 나은 성능의 개선을 보여주었으며, 바로 앞서 소개한 Dual Learning[He et al.2016a]보다도 더 나은 성능을 보여줍니다. 마찬가지로, 불안정하고 비효율적인 강화학습을 사용하지 않고도 더 나은 성능을 보여준 것은 주목할 만한 성과라고 할 수 있습니다.

## Appendix: Importance Sampling

$$\begin{aligned}
\mathbb{E}_{X \sim p}[f(x)] &= \int_x f(x)p(x)dx \\
&= \int_x \left( f(x) \frac{p(x)}{q(x)} \right) q(x)dx \\
&= \mathbb{E}_{X \sim q} \left[ f(x) \frac{p(x)}{q(x)} \right],
\end{aligned}$$

$$\forall q \text{ (pdf) s.t. } q(x) = 0 \implies p(x) = 0$$

$$w(x) = \frac{p(x)}{q(x)}$$

$$\begin{aligned}
\mathbb{E}_{X \sim q} \left[ f(x) \frac{p(x)}{q(x)} \right] &\approx \frac{1}{K} \sum_{i=1}^K f(x_i) \frac{p(x_i)}{q(x_i)} \\
&= \frac{1}{K} \sum_{i=1}^K f(x_i) w(x_i)
\end{aligned}$$

where  $x_i \sim q$

# Productization of Neural Machine Translation System



Figure 1:

## Productization

이전까지의 챕터들에서 우리는 자연어처리에 대한 다양한 이론과 실습 예제들을 다루었습니다. 이제 이러한 자연어처리 알고리즘들이 어떻게 실제 필드에서 결합되어 적용되고 있는지 살펴보고자 합니다. Google, Microsoft와 같은 여러 회사들도 자신들의 기술력을 자랑하기 위하여 앞다투어 자신들이 적용한 자연어처리(기계번역) 시스템에 대한 논문을 아주 상세하게 논문에 공개 합니다. 많은 회사들이 자신들의 상용화 경험을 토대로 성과를 자랑하는 논문을 써 내고 있습니다. 우리는 자연어처리 분야의 가장 성공적인 상용화분야인 기계번역의 상용화 사례에 대해서 다루어 보도록 하겠습니다. 이제 기계번역은 2014년도의 큰

충격 이후로부터 수많은 응용 방법들이 쏟아져나왔고, 어느정도 표준적인 방법 A to Z가 정립되어 가고 있는 시점입니다.

사실, 요즈음과 같이 새로운 기술에 대한 논문 및 소스코드 공개가 당연시되는 현상 아래에서는 각 기계번역 시스템마다의 수준의 편차가 그렇게 크지 않습니다. 따라서 누가 최신 기술을 구현했느냐보다, 상용화에 대한 경험과 연륜, 데이터의 양과 품질(정제 이슈)에 따라 성능이 결정될 수 있습니다.

아래는 현재 대표적인 기계번역 시스템의 현 주소를 살펴볼 수 있는 간단한 샘플 문장입니다. (2018년 04월 기준)

차를 마시러 공원에 가던 차 안에서 나는 그녀에게 차였다. – Google: I was kicking her in the car that went to the park for tea. – Microsoft: I was a car to her, in the car I had a car and went to the park. – Naver: I got dumped by her on the way to the park for tea. – Kakao: I was in the car going to the park for tea and I was in her car. – SK: I got dumped by her in the car that was going to the park for a cup of tea.

위와 같이 어려운 문장(문장구조는 쉬우나 중의성에 대해서 높은 난이도)에 대해서 어찌보면 갈 길이 멀기도 합니다. 따라서, 위와 같이 누군가 저런 어려운 문제에 대한 한단계 더 높은 기술력을 가졌다기보단, 기술의 수준은 비슷하다고 볼 수 있습니다.

## Pipeline for Machine Translation

이번 섹션에서는 실제 기계번역 시스템을 구축하기 위한 절차와 시스템 Pipeline이 어떻게 구성되는지 살펴보도록 하겠습니다. 이 과정들은 대부분 기계번역 뿐만 아니라 기본적인 자연어처리 문제 전반에 적용 가능하다고 할 수 있습니다. 그리고 실제 Google 등에서 발표한 논문을 통해서 그들의 상용 번역 시스템의 실제 구성에 대해서 살펴보도록 하겠습니다.

통상적으로 번역시스템을 구축하면 아래와 같은 흐름을 가지게 됩니다.

## 준비과정

### 1. Corpus 수집, 정제

- 병렬 말뭉치(parallel corpus)를 다양한 소스에서 수집합니다. WMT등 번역 시스템 평가를 위해 학술적으로 공개 된 데이터 세트도 있을 뿐더러,

뉴스 기사, 드라마/영화 자막, 위키피디아 등을 수집하여 번역 시스템에 사용 할 수 있습니다.

- 수집된 데이터는 정제 과정을 거쳐야 합니다. 정제 과정에는 양 언어의 말뭉치에 대해서 문장 단위로 정렬을 시켜주는 작업부터, 특수문자 등의 noise를 제거해 주는 작업도 포함 됩니다.

## 2. Tokenization

- 각 언어 별 POS tagger 또는 segmenter를 사용하여 띠어쓰기를 정제(normalization) 합니다. 영어의 경우에는 대소문자에 대한 정제 이슈가 있을 수도 있습니다. 한국어의 경우에는 한국어의 특성 상, 인터넷에 공개 되어 있는 corpus 들은 띠어쓰기가 제멋대로일 수 있습니다.
- 한국어의 경우에는 Mecab과 같은 공개 되어 있는 파서(parser)들이 있습니다.
- 띠어쓰기가 정제 된 이후에는 Byte Pair Encoding(BPE by Subword or Wordpiece)를 통해 추가적인 tokenization을 수행하고 어휘 목록을 구성합니다.

## 3. Batchify

- 전처리 작업이 끝난 corpus에 대해서 훈련을 시작하기 위해서 mini-batch로 만드는 작업이 필요합니다.
- 여기서 중요한 점은 mini-batch 내의 문장들의 길이를 최대한 통일시켜 주는 것입니다. 이렇게 하면 문장 길이가 다름으로 인해서 발생하는 훈련 시간 낭비를 최소화 할 수 있습니다.
- 예를 들어 mini-batch 내에서 5단어 짜리 문장과 70단어 짜리 문장이 공존할 경우, 5단어 짜리 문장에 대해서는 불필요하게 65 time-step을 더 진행해야 하기 때문입니다. 따라서 5단어 짜리 문장끼리 모아서 mini-batch를 구성하면 해당 batch에 대해서는 훨씬 수행 시간을 줄일 수 있습니다.
- 실제 훈련 할 때에는 이렇게 구성된 mini-batch들의 순서를 임의로 섞어(shuffling) 훈련하게 됩니다.

## 4. Training

- 준비된 데이터셋을 사용하여 seq2seq 모델을 훈련 합니다.

## 5. Inference

- 성능 평가(evaluation)를 위한 추론(inference)을 수행 합니다. 이때에는 준비된 테스트셋(훈련 데이터셋에 포함되어 있지 않은)을 사용하여 추론을 수행합니다.

## 6. De-tokenization

- 추론 과정이 끝나더라도 tokenization이 되어 있어 아직 사람이 실제 사용하는 문장 구성과 형태가 다릅니다. 따라서 detokenization을

수행하면 실제 사용되는 문장의 형태로 반환 됩니다.

## 7. Evaluation

- 이제 이렇게 얻어진 문장에 대해서 정량평가를 수행 합니다. 기계번역을 위한 정량평가 방법으로는 BLEU가 있습니다. 비교대상의 BLEU점수와 비교하여 어느 모델이 더 우월한지 알 수 있습니다.
- 정량평가에서 기존 대비 또는 경쟁사 대비 더 우월한 성능을 갖추었음을 알게 되면, 정성평가를 수행 합니다. 번역의 경우에는 사람이 직접 비교 대상들의 결과들과 비교하여 어느 모델이 우월한지 평가 합니다.
- 정량평가와 정성평가 모두 성능이 개선되었음을 알게 되면 이제 서비스에 적용 가능 합니다.

# 서비스

## 1. API 호출 or 사용자로부터의 입력

- 대부분 서비스 별 API 서버를 만들어 실제 프론트엔드(front-end)로부터 API 호출을 받아 프로세스를 시작 합니다.
- 서비스의 스케일(scale)에 따라 load-balancer등을 두어 부하를 적절히 분배하기도 합니다.
- 서비스의 형태나 성격에 따라서 실제 모델 추론(inference)을 수행하지 않고, 정해진 응답을 반환하기도 합니다.

## 2. Tokenization

- 추론을 위해서 실제 모델에 훈련된 데이터셋과 동일한 형태로 tokenization을 수행 합니다.
- 언어에 따라서 띠어쓰기 정제를 위한 tokenizer를 사용합니다. (예: 한국어의 경우에는 Mecab)
- 띠어쓰기가 정제 된 이후에는 Byte Pair Encoding(BPE by Subword or Wordpiece)를 통해 subword 형태로 추가적인 tokenization을 해 줄 수 있습니다.

## 3. Inference

- 정량/정성 평가를 통해 성능이 입증된 모델을 통해 추론을 수행 합니다.

## 4. De-tokenization

- 다시 사람이 읽을 수 있는 형태로 detokenization을 수행 합니다.

## 5. API 결과 반환 or 사용자에게 결과 반환

- 최종 결과물을 API 서버에서 반환하여 서비스 프로세스를 종료 합니다. # Google Neural Machine Translation (GNMT) (Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation)

Google은 2016년 논문([Wo et al.2016])을 발표하여 그들의 번역시스템에 대해서 상세히 소개하였습니다. 실제 시스템에 적용된 모델 구조(architecture)부터 훈련 방법까지 상세히 기술하였기 때문에, 실제 번역 시스템을 구성하고자 할 때에 좋은 참고자료(reference)가 될 수 있습니다. 또한 다른 논문들에서 실험 결과에 대해 설명할 때, GNMT를 baseline으로 참조하기도 합니다. 아래의 내용들은 그들의 논문에서 소개한 내용을 다루도록 하겠습니다.

## Model Architecture

Google도 seq2seq 기반의 모델을 구성하였습니다. 다만, 구글은 훨씬 방대한 데이터셋을 가지고 있기 때문에 그에 맞는 깊은 모델을 구성하였습니다. 따라서 아래에 소개될 방법들이 깊은 모델들을 효율적으로 훈련 할 수 있도록 사용되었습니다.

### Residual Connection

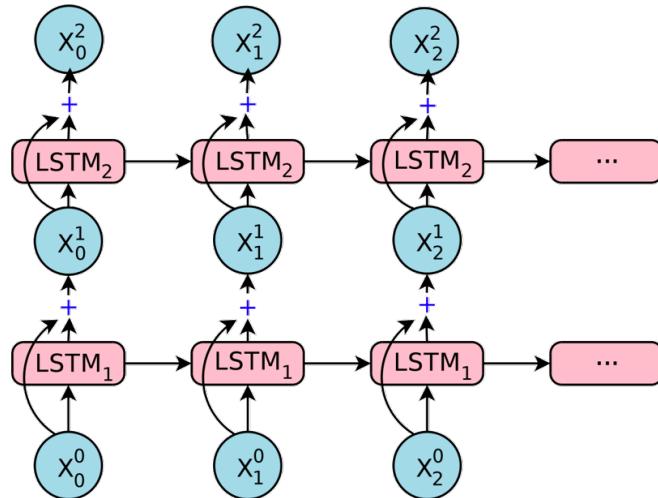


Figure 2:

보통 LSTM layer를 4개 이상 쌓기 시작하면 모델이 더욱 깊어(deeper)짐에 따라서 성능 효율이 저하되기 시작합니다. 따라서 Google은 깊은 모델은 효율적으로 훈련시키기 위하여 residual connection을 적용하였습니다.

## Bi-directional Encoder for First Layer

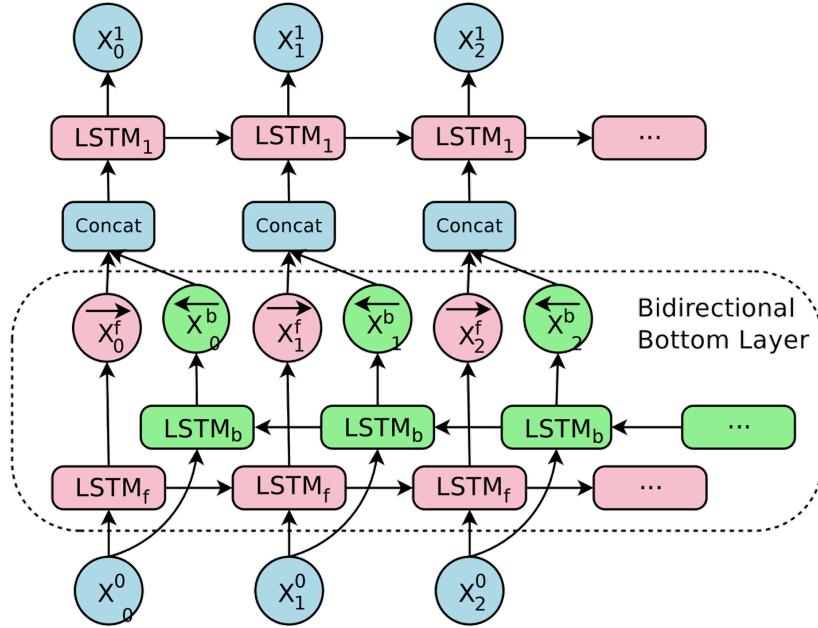


Figure 3:

또한, 모든 LSTM stack에 대해서 bi-directional LSTM을 적용하는 대신에, 첫번째 층에 대해서만 bi-directional LSTM을 적용하였습니다. 따라서 훈련(training) 및 추론(inference) 속도에 개선이 있었습니다.

## Segmentation Approaches

### Wordpiece Model

구글도 마찬가지로 BPE 모델을 사용하여 tokenization을 수행하였습니다. 그리고 그들은 그들의 tokenizer를 오픈소스로 공개하였습니다. – SentencePiece: <https://github.com/google/sentencepiece> 마찬가지로 아래와 같이 띄어쓰기는 underscore로 치환하고, 단어를 subword별로 통계에 따라 segmentation 합니다.

- original: Jet makers feud over seat width with big orders at stake
- wordpieces: \_J et \_makers \_fe ud \_over \_seat \_width \_with \_big \_orders \_at \_stake

## Training Criteria

Table 6: Single model test BLEU scores, averaged over 8 runs, on WMT En→Fr and En→De

Dataset	Trained with log-likelihood	Refined with RL
En→Fr	38.95	39.92
En→De	24.67	24.60

Figure 4:

Google은 강화학습을 다른 챕터에서 설명한 강화학습 기법을 사용하여 Maximum Likelihood Estimation (MLE)방식의 훈련된 모델에 fine-tuning을 수행하였습니다. 따라서 위의 테이블과 같은 추가적이 성능 개선을 얻어낼 수 있었습니다.

기존 MLE 방식의 목적함수(objective)를 아래와 같이 구성합니다.  $Y^{*(i)}$ 은 최적(optimal)의 정답 데이터를 의미합니다.

$$\mathcal{O}_{ML}(\theta) = \sum_{i=1}^N \log P_\theta(Y^{*(i)}|X^{(i)})$$

여기에 추가로 RL방식의 목적함수(objective)를 추가하였는데 이 방식이 policy gradient 방식과 같습니다.

$$\mathcal{O}_{RL}(\theta) = \sum_{i=1}^N \sum_{Y \in \mathcal{Y}} P_\theta(Y|X^{(i)}) r(Y, Y^{*(i)})$$

위의 수식도 Minimum Risk Training (MRT) 방식과 비슷합니다.  $r(Y, Y^{*(i)})$  또한 정답과 sampling 데이터 사이의 유사도(점수)를 의미합니다. 가장 큰 차이점은 기존에는 risk로 취급하여 최소화(minimize)하는 방향으로 훈련하였지만, 이번에는 reward로 취급하여 최대화(maximize)하는 방향으로 훈련하게 된다는 것입니다.

이렇게 새롭게 추가된 목적함수(objective)를 아래와 같이 기존의 MLE방식의 목적함수와 선형 결합(linear combination)을 취하여 최종적인 목적함수가 완성됩니다.

$$\mathcal{O}_{Mixed}(\theta) = \alpha * \mathcal{O}_{ML}(\theta) + \mathcal{O}_{RL}(\theta)$$

이때에  $\alpha$ 값은 주로 0.017로 설정하였습니다. 위와 같은 방법의 성능을 실험한 결과는 다음과 같습니다.

Table 6: Single model test BLEU scores, averaged over 8 runs, on WMT En→Fr and En→De

Dataset	Trained with log-likelihood	Refined with RL
En→Fr	38.95	39.92
En→De	24.67	24.60

Figure 5:

*En → De*의 경우에는 성능이 약간 하락함을 보였습니다. 하지만 이는 decoder의 length penalty, coverage penalty와 결합되었기 때문이고, 이 페널티(penalty)들이 없을 때에는 훨씬 큰 성능 향상이 있었다고 합니다.

## Quantization

실제 인공신경망을 사용한 제품을 개발할 때에는 여러가지 어려움에 부딪히게 됩니다. 이때, Quantization을 도입함으로써 아래와 같은 여러가지 이점을 얻을 수 있습니다.

- 계산량을 줄여 자원의 효율적 사용과 응답시간의 감소를 얻을 수 있다.
- 모델의 실제 저장되는 크기를 줄여 deploy를 효율적으로 할 수 있다.
- 부가적으로 regularization의 효과를 볼 수 있다.

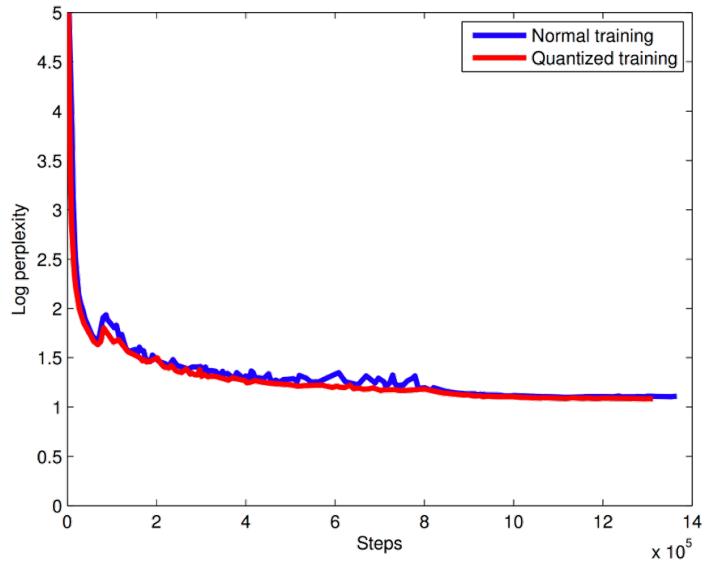


Figure 6:

위의 그래프를 보면 전체적으로 Quantized verion이 더 낮은 loss를 보여주는 것을 확인할 수 있습니다.

## Search

### Length Penalty and Coverage Penalty

Google은 기존에 소개한 Length Penalty에 추가로 Coverage Penalty를 사용하여 좀 더 성능을 끌어올렸습니다. Coverage penalty는 attention weight(probability)의 값의 분포에 따라서 매겨집니다. 이 페널티는 좀 더 attention이 고루 잘 퍼지게 하기 위함입니다.

$$s(Y, X) = \log P(Y|X)/lp(Y) + cp(X; Y)$$

$$lp(Y) = \frac{(5 + |Y|)^\alpha}{(5 + 1)^\alpha}$$

$$cp(X; Y) = \beta * \sum_{i=1}^{|X|} \log (\min (\sum_{j=1}^{|Y|} p_{i,j}, 1.0))$$

where  $p_{i,j}$  is the attention weight of the  $j$ -th target word  $y_j$  on the  $i$ -th source word  $x_i$ .

Coverage penalty의 수식을 들여다보면, 각 source word  $x_i$ 별로 attention weight의 합을 구하고, 그것의 평균(=합)을 내는 것을 볼 수 있습니다. 로그(log)를 취했기 때문에 그 중에 attention weight가 편중되어 있다면, 편중되지 않은 source word는 매우 작은 음수 값을 가질 것이기 때문에 좋은 점수를 받을 수 없을 겁니다.

실험에 의하면  $\alpha$ 와  $\beta$ 는 각각 0.6, 0.2 정도가 좋은것으로 밝혀졌습니다. 하지만, 상기한 강화학습 방식을 training criteria에 함께 이용하면 그다지 그 값은 중요하지 않다고 하였습니다.

## Training Procedure

Google은 stochastic gradient descent (SGD)를 써서 훈련 시키는 것 보다, Adam과 섞어 사용하면 (epoch 1까지 Adam) 더 좋은 성능을 발휘하는 것을 확인하였습니다.

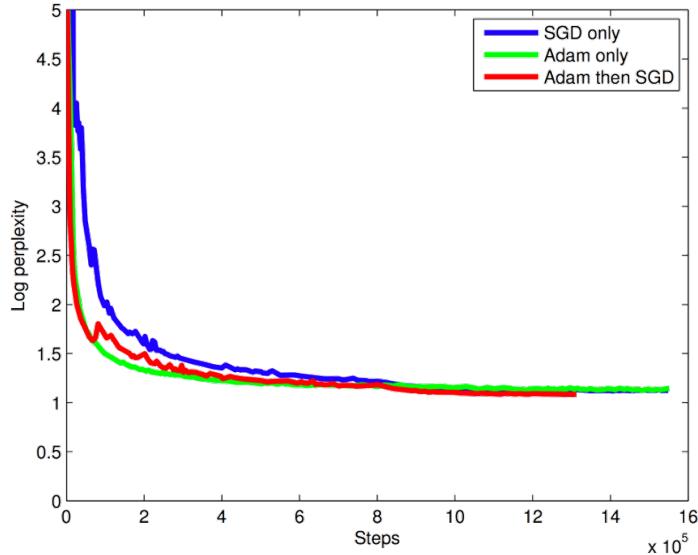


Figure 7:

## Evaluation

Table 10: Mean of side-by-side scores on production data

	PBMT	GNMT	Human	Relative Improvement
English → Spanish	4.885	5.428	5.504	87%
English → French	4.932	5.295	5.496	64%
English → Chinese	4.035	4.594	4.987	58%
Spanish → English	4.872	5.187	5.372	63%
French → English	5.046	5.343	5.404	83%
Chinese → English	3.694	4.263	4.636	60%

Figure 8:

실제 번역 품질을 측정하기 위하여 BLEU 이외에도 정성평가(implicit human evaluation)을 통하여 GNMT의 성능 개선의 정도를 측정하였습니다. 0(Poor)에서 6(Perfect)점 사이로 점수를 매겨 사람의 번역 결과 점수를 최대치로 가정하고 성능의 개선폭을 계산하였습니다. 실제 SMT 방식 대비 엄청난 천지개벽 수준의 성능 개선이 이루어진 것을 알 수 있고, 일부 언어쌍에 대해서는 거의 사람의 수준에 필적하는 성능을 보여주는 것을 알 수 있습니다.

## The University of Edinburgh's Neural MT Systems for WMT17

사실 Google의 논문은 훌륭하지만 매우 스케일이 매우 큽니다. 저는 그래서 작은 스케일의 기계번역 시스템에 관한 논문은 이 논문[Sennrich et al.2017]을 높게 평가합니다. 이 논문도 기계번역 시스템을 구성할 때에 훌륭한 baseline이 될 수 있습니다. Edinburgh 대학의 Sennrich교수는 매년 열리는 WMT 대회에 참가하고 있고, 해당 대회에 참가하는 기계번역 시스템들은 이처럼 매년 자신들의 기술에 대한 논문을 제출합니다. 좋은 참고자료로 삼을 수 있습니다.

## Subword Segmentation

---

### Algorithm 1 Learn BPE operations

---

```
import re, collections

def get_stats(vocab):
    pairs = collections.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i],symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
    bigram = re.escape(' '.join(pair))
    p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
    for word in v_in:
        w_out = p.sub(''.join(pair), word)
        v_out[w_out] = v_in[word]
    return v_out

vocab = {'l o w </w>' : 5, 'l o w e r </w>' : 2,
         'n e w e s t </w>':6, 'w i d e s t </w>':3}
num_merges = 10
for i in range(num_merges):
    pairs = get_stats(vocab)
    best = max(pairs, key=pairs.get)
    vocab = merge_vocab(best, vocab)
    print(best)
```

---

$$\begin{array}{ccc} r \cdot & \rightarrow & r \cdot \\ l o & \rightarrow & lo \\ l o w & \rightarrow & low \\ e r \cdot & \rightarrow & er \cdot \end{array}$$

Figure 1: BPE merge operations learned from dictionary {‘low’, ‘lowest’, ‘newer’, ‘wider’}.

[Sennrich et al. 2016]

이 논문 또한 (그들이 처음으로 제안한 방식이기에) BPE 방식을 사용하여 tokenization을 수행하였습니다. 이제 우리는 subword 기반의 tokenization 방식이 하나의 정석이 되었음을 알 수 있습니다. 위의 code는 BPE algorithm에 대해서 간략하게 소개한 code입니다. 전처리 챕터에서 소개했지만, subword 방식은 위와 같이 가장 많이 등장한 문자열(character sequence)에 대해서 합쳐주며 iteration을 반복하고, 원하는 어휘(vocabulary) 숫자가 채워질때까지 해당 iteration을 반복합니다.

## Architecture

이 논문에서는 seq2seq를 기반으로 모델 구조(architecture)를 만들었는데, 다만 LSTM이 아닌 GRU를 사용하여 RNN stack을 구성하였습니다. Google과 마찬가지로 residual connection을 사용하여 stack을 구성하였고, encoder의 경우에는 4개층, decoder의 경우에는 8개 층을 쌓아 모델을 구성하였습니다. 실험 시에는 *hidden size = 1024*, *word vector dimension = 512*를 사용하였습니다. 또한, Google과는 다르게 순수하게 Adam만을 optimizer로 사용하여 훈련을 하였습니다.

## Synthetic Data using Monolingual Data

이전 섹션에서 소개한 그들이 제안한 논문[Sennrich et al.,2015]의 방식대로 back translation과 copied translation 방식을 사용하여 합성 병렬(pseudo parallel) corpus를 구성하여 훈련 데이터셋에 추가하였습니다. 이때에 비율은 실험 결과에 따라서 *parallel : copied : back = 1 : 1 ~ 2 : 1 ~ 2*로 조절하여 사용하였습니다.

## Ensemble

이 논문에서 그들은 2가지 양상을(ensemble) 기법을 모두 사용하였습니다.

- checkpoint ensemble
  - 특정 epoch에서부터 다른 모델로 다시 훈련하여 ensemble을 구성합니다. 훈련 중간부터 다시 훈련하기 때문에 시간적으로 굉장히 효율적입니다.
- independent ensemble
  - 처음부터 다른 모델로 훈련하여 ensemble로 구성합니다. 처음부터 다시 훈련하므로 checkpoint 방식에 비해서 비효율적이지만, 다양성(diversity) 관점에서 낫습니다.
- Machine translation that makes sense: the Booking.com use case
- Machine Translation at Booking.com: Journey and Lessons Learned [Levin et al.2017] # Microsoft Machine Translation (Achieving Human Parity on Automatic Chinese to English News Translation)

2018년 3월에 나온 Microsoft의 기계번역 시스템에 대한 논문([Hassan et al.,2018])입니다. 2016년에 발표한 Google의 논문은 기계번역 자체의 기본 성능을 끌어올리는 모델 아키텍쳐와 훈련 방법 등의 내부 구조에 대해서 많은 설명을

할애하였던 것과 달리, Microsoft의 논문은 기술을 이미 끌어올려진 기술의 기반 위에서 더욱 그 성능을 견고히하는 방법에 대한 설명에 분량을 더 할애하였습니다.

이 논문은 중국어와 영어 간 기계번역 시스템을 다루고 있고, 뉴스 도메인(domain) 번역에 있어서 사람의 번역과 비슷한 성능에 도달하였다고 선언하고 있습니다. 다만, 제안한 방법에 의해 구성된 기계번역 시스템이 모든 언어쌍에 대해서, 모든 분야의 도메인에 대해서 같은 사람 번역 수준에 도달하지는 못할 수도 있다고 설명하고 있습니다.

Microsoft는 전통적인(?) RNN방식의 seq2seq 대신, Google의 Transformer 구조를 사용하여 seq2seq를 구현하였습니다. 이 논문에서 소개한 중점 기술은 아래와 같습니다.

- Back-translation과 Dual learning(Unsupervised and Supervised, both)을 통한 단방향(monolingual) corpora의 활용 극대화
- Auto-regressive 속성(이전 time-step의 예측(prediction)이 다음 time-step의 예측에 영향을 주는 것)의 단점을 보완하기 위한 Deliberation Networks([Xia et al.,2017])과 Kullback–Leibler (KL) divergence를 이용한 regularization
- NMT 성능을 극대화 하기 위한 훈련 데이터 선택(selection)과 필터링(filtering)

위의 기술들에 대해서 한 항목씩 차례로 살펴보도록 하겠습니다.

## Exploiting the Dual Nature of Translation

### Dual Learning for NMT

이 논문에서는, 앞 챕터에서 설명한, duality를 활용하여 양방향(bilingual) corpus를 활용한 Dual Supervised Learning (DSL) [Xia et al.2017] 방식과, 단방향(monolingual) corpus를 marginal 분포(distribution)에 적용하여 활용한 Dual Unsupervised Learning (DUL) [Wang et al.2017] 방식을 모두 사용하였습니다.

### Dual Unsupervised Learning (DUL)

$$\begin{aligned}\mathcal{L}(x; \theta_{x \rightarrow y}) &= E_{y \sim P(\cdot|x; \theta_{x \rightarrow y})} \{\log P(x|y; \theta_{y \rightarrow x})\} \\ &= \sum_y P(y|x; \theta_{x \rightarrow y}) \log P(x|y; \theta_{y \rightarrow x})\end{aligned}$$

$$\frac{\partial \mathcal{L}(x; \theta_{x \rightarrow y})}{\partial \theta_{x \rightarrow y}} = \sum_y \frac{\partial P(y|x; \theta_{x \rightarrow y})}{\partial \theta_{x \rightarrow y}} \log P(x|y; \theta_{y \rightarrow x})$$

## Dual Supervised Learning (DSL)

### Joint Training of Src2Tgt and Tgt2Src Models

### Beyond the Left-to-Right Bias

#### Deliberation Networks

Microsoft는 [Xia et al.2017]에서 소개한 방식을 통해 번역 성능을 더욱 높이려 하였습니다. 이 방법은 사람이 번역을 하는 방법에서 영감을 얻은 것입니다. 사람은 번역을 할 때 source 문장에서 초안(draft) target 문장을 얻고, 그 초안으로부터 최종적인 target 문장을 번역 해 내곤 합니다. 따라서 신경망을 통해 같은 방법을 구현하고자 하였습니다.

예를 들어 기존의 번역의 수식은 deliberation networks를 통해 아래와 같이 바뀔 수 있습니다.

$$P(Y|X) = \prod P(y_i|X, y_{<i}) \longrightarrow P(Y|X) = \prod P(y_i|X, Y_{mid}, y_{<i})$$

### Agreement Regularization of Left-to-Right and Right-to-Left Models

#### Data Selection and Filtering

#### Evaluation

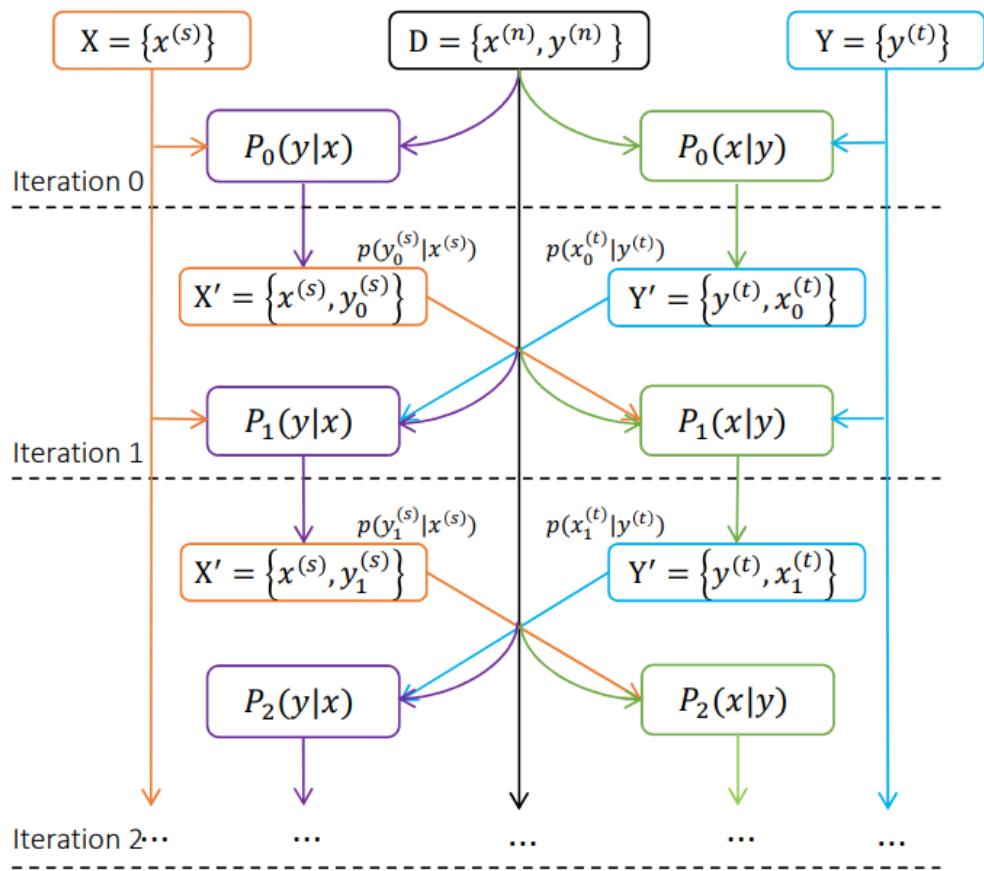


Figure 1: Illustration of joint training: S2T  $p(\mathbf{y}|\mathbf{x})$  and T2S  $p(\mathbf{x}|\mathbf{y})$

Figure 9:

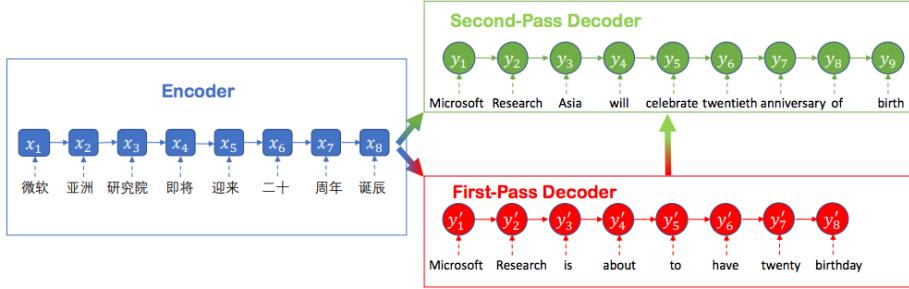


Figure 3: An example showing the decoding process of deliberation network.

Figure 10:

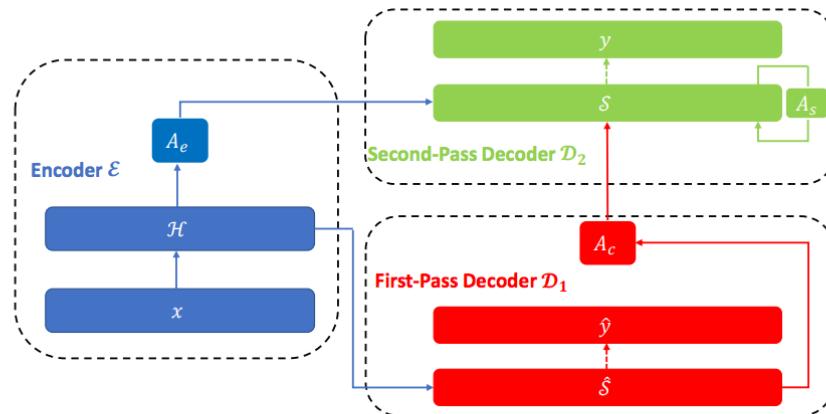


Figure 4: Deliberation network: Blue, red and green parts indicate encoder  $\mathcal{E}$ , first-pass decoder  $\mathcal{D}_1$  and second-pass decoder  $\mathcal{D}_2$  respectively. Solid lines represent the information flow via attention model. The self attention model within  $\mathcal{E}$  and the  $\mathcal{E}$ -to- $\mathcal{D}_1$  attention model are omitted for readability.

Figure 11:

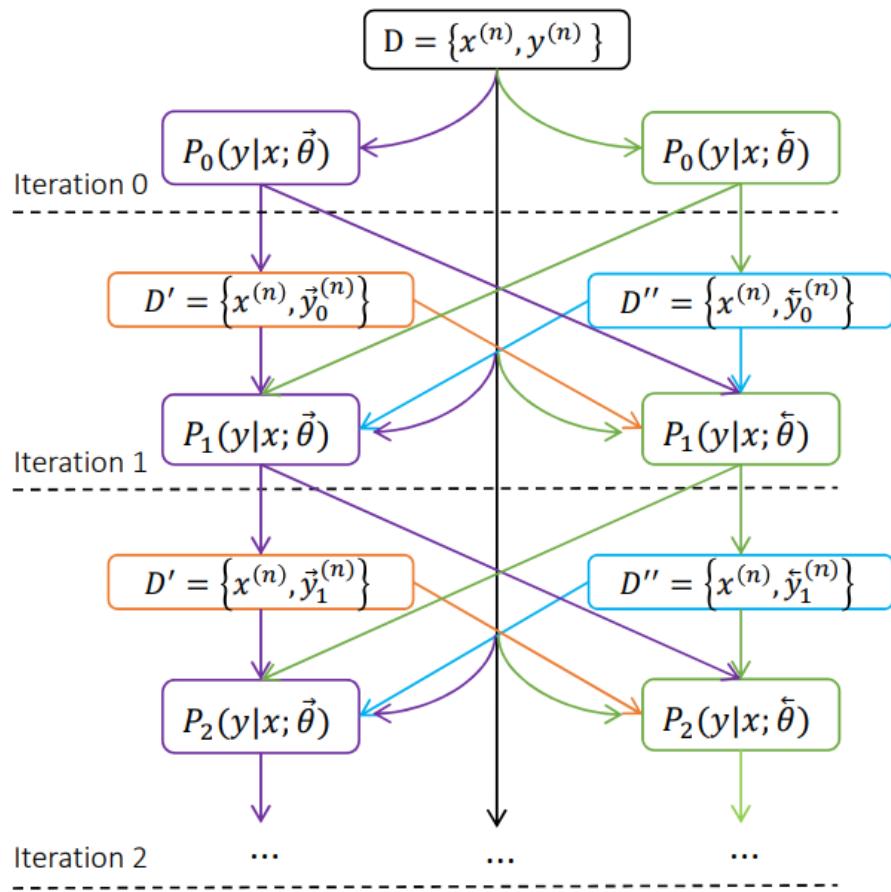


Figure 2: Illustration of agreement regularization: L2R  $p(\mathbf{y}|\mathbf{x}; \vec{\theta})$  and R2L  $p(\mathbf{y}|\mathbf{x}; \overleftarrow{\theta})$

Figure 12:

SystemID	Settings	BLEU
Sogou	WMT 2017 best result [42]	26.40
Base	Transformer Baseline	24.2
BT	+Back Translation	25.57
DL	BT + Dual Learning	26.51
DLDN	BT + Dual Learning + Deliberation Nets	27.40
DLDN2	DLDN without first decoder reranking	27.20
DLDN3	BT+ Dual Learning + R2L sampling	26.88
DLDN4	BT+ Dual Learning + Bi-NMT	27.16
AR	BT + Agreement Regularization	26.91
ARJT	BT + Agreement Regularization + Joint Training	27.38
ARJT2	ARJT + dropout=0.1	27.19
ARJT3	ARJT + dropout=0.05	27.07
ARJT4	ARJT + dropout=0.01	26.98

Table 1: Automatic (BLEU) evaluation results on the WMT 2017 Chinese-English test set

Figure 13:

## References

### Websites:

- <http://cs224d.stanford.edu/syllabus.html>
- <http://web.stanford.edu/class/cs224n/syllabus.html>
- <http://web.stanford.edu/class/cs124/>
- <http://cs231n.stanford.edu/syllabus.html>
- <https://web.stanford.edu/~jurafsky/slp3/>
- [https://github.com/nyu-dl/NLP\\_DL\\_Lecture\\_Note/blob/master/lecture\\_note.pdf](https://github.com/nyu-dl/NLP_DL_Lecture_Note/blob/master/lecture_note.pdf)
- <https://sites.google.com/view/seq2seq-icml17>
- <https://khan.github.io/KaTeX/function-support.html>
- <https://github.com/OpenNMT/OpenNMT-py/>
- <https://machinelearningmastery.com/applications-of-deep-learning-for-natural-language-processing/>    <http://ruder.io/deep-learning-nlp-best-practices/>
- Oxford NLP lecture notes

### Youtube:

- RL by David Silver
- NLP by Prof. Jurafsky
- NLP by Prof. Michael Collins
- NLP at Oxford Univ. w/ Deepmind
- NLP at Stanford by Prof. Manning
- Machine Learning by Mathematicalmonk

### Depends on Chapters:

- NLP w/ Deeplearning
- Hello PyTorch
- WSD
- Preprocesing
- Word Embedding Vector
- Sequence Modeling

- Text Classification
- Deep Learning for Sentiment Analysis: A Survey
- Learning to Generate Reviews and Discovering Sentiment
- <https://www.toptal.com/machine-learning/nlp-tutorial-text-classification>
- article from WildML
- Language Modeling
- Machine Translation
- NON-AUTOREGRESSIVE NEURAL MACHINE TRANSLATION
- WORD TRANSLATION WITHOUT PARALLEL DATA
- DiSAN: Directional Self-Attention Network for RNN/CNN-Free Language Understanding
- BI-DIRECTIONAL BLOCK SELF-ATTENTION FOR FAST AND MEMORY-EFFICIENT SEQUENCE MODELING
- RL
- SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient
- BATCH POLICY GRADIENT METHODS FOR IMPROVING NEURAL CONVERSATION MODELS
- AN ACTOR-CRITIC ALGORITHM FOR SEQUENCE PREDICTION
- REINFORCEMENT LEARNING NEURAL TURING MACHINES
- Reinforcement Learning for Bandit Neural Machine Translation with Simulated Human Feedback
- On Monte Carlo Tree Search and Reinforcement Learning
- DEEP REINFORCEMENT LEARNING: AN OVERVIEW
- Asynchronous Methods for Deep Reinforcement Learning
- Evolution Strategies as a Scalable Alternative to Reinforcement Learning
- A3C Slides
- Asynchronous Methods for Deep Reinforcement Learning
- etc
- Text Summarization Techniques: A Brief Survey
- Deep Learning for Speech Recognition @ Cambridge
- STATE-OF-THE-ART SPEECH RECOGNITION WITH SEQUENCE-TO-SEQUENCE MODELS
- DeepMind Seminar @ Youtube
- Stanford Lecture @ Youtube
- Deep Speech Recognition @ MS
- A Deep Reinforcement Learning Chatbot (Short Version)