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 때문 입니다.

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 Classificaion)	명령, 질문, 잡담 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 방법 입니다. 의외로 기대 이상의 성능을 보여줄 때가 많습니다. 물론 단어를 여전히 discrete한 심볼로 다루기 때문에, 여전히 아쉬운 부분이 많습니다. 이번 섹션에서는 Naive Bayes를 통해서 텍스트를 분류 하는 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 X를 만족하며 확률을 최대로 하는 클래스 Y를 구할 수 있습니다. 이처럼 posterior 확률을 최대화(maximize)하는 y를 구하는 것을 Maximum A Posterior (MAP)라고 부릅니다. 그 수식은 아래와 같습니다.

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

다시한번 수식을 살펴보면, X(데이터)가 주어졌을 때, 가능한 클래스의 set $mathcalY$ 중에서 posterior를 최대로 하는 클래스 y를 선택하는 것 입니다.

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

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

MLE는 주어진 데이터X와 클래스 레이블(label) Y가 있을 때, parameter θ를 훈련하는 방법으로도 많이 사용 됩니다.

$$\hat{\theta} = \underset{\theta}{\text{argmax}}\ P(Y|X, \theta)$$

MLE vs MAP

경우에 따라 MAP는 MLE에 비해서 좀 더 정확할 수 있습니다. prior(사전)확률이 반영되어 있기 때문 입니다. 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 X는 5단위의 정수로 이루어져 있습니다. $X = \{\cdots, 145, 150, 155, 160, \cdots\}$

신발사이즈 155는 남자 신발사이즈 치곤 매우 작은 편 입니다. 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 Y를 예측 합니다. 먼저 우리가 알고자 하는 값인 posterior 확률은 아래와 같을 것 입니다.

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

이때, X가 다양한 feature(특징)들로 이루어진 데이터라면, 훈련 데이터에서 매우 희소(rare)할 것이므로 posterior 뿐만 아니라, likelihood $P(X = w_1, w_2, \cdots, w_n|Y = c)$를 구하기 어려울 것 입니다. 왜냐하면 보통 확률을 코퍼스의 출현빈도를 통해 추정할 수 있는데, feature가 복잡할 수록 likelihood 또는 posterioir를 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.

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

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

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

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 것 입니다. 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 방법 입니다. 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.

$$\mathcal{C} = \{pos, neg\}$$
$$\mathcal{D} = \{d_1, d_2, \cdots\}$$

위와 같이 긍정(pos)과 부정(neg)으로 클래스가 구성($\mathcal{C}$되어 있고, 문서 d로 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.

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

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.

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

마찬가지로 부정 감성에 대해 같은 작업을 반복 할 수 있습니다.

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

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

Add-one Smoothing

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

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

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

따라서 우리는 이런 경우를 위하여 각 출현횟수에 1을 더해주어 간단하게 문제를 완화 할 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.

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

"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_t는 이전 time-step에 등장한 단어들 $w_1, \cdots, w_{t_1}$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

Gx

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

Gy

Figure 2: Sobel Filters for vertial 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 convoltuion 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

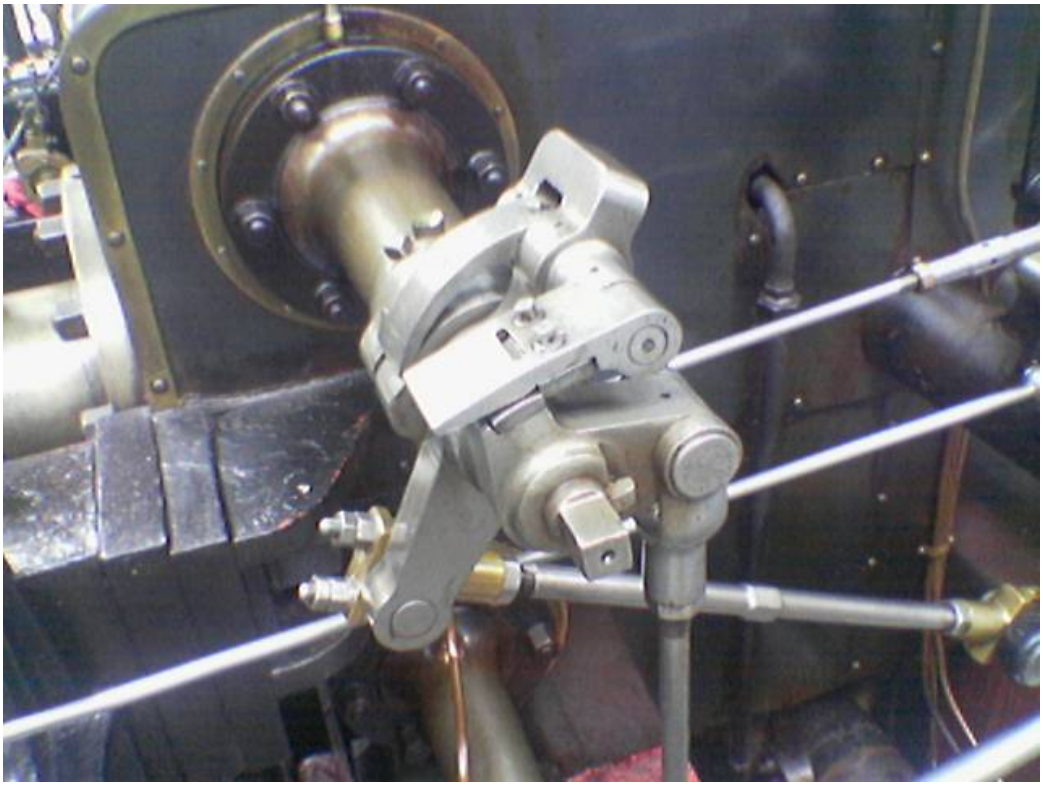

Figure 3: An image before Sobel filter (from Wikipedia)

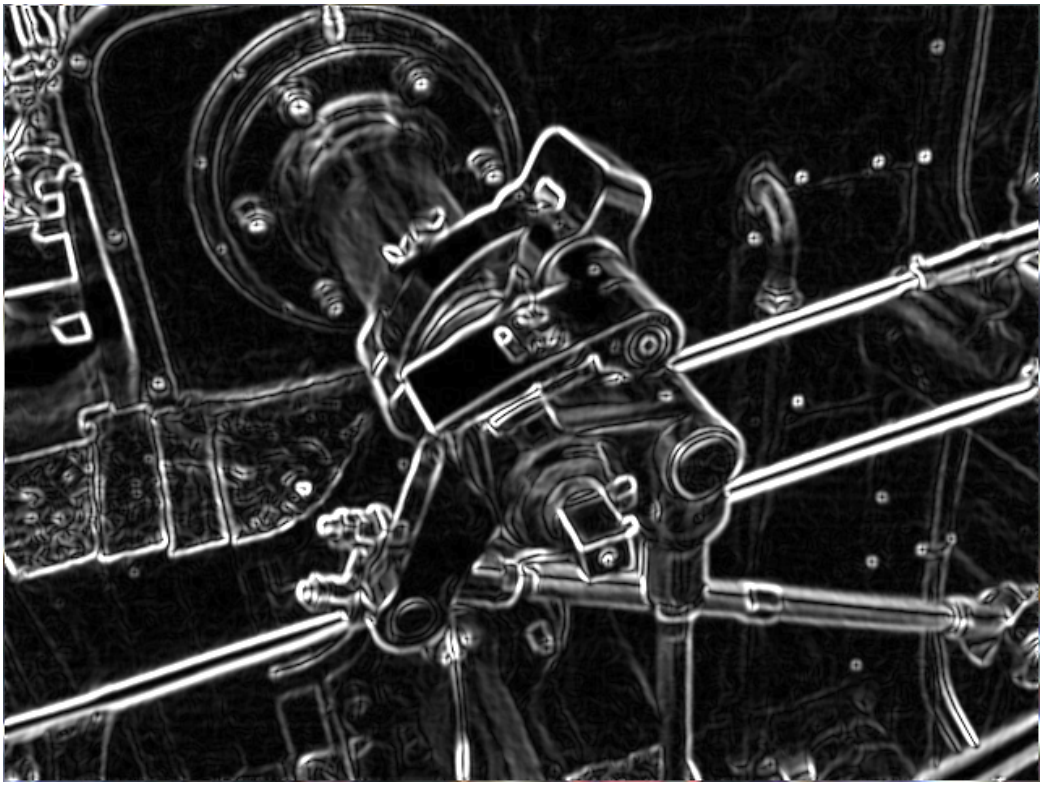

Figure 4: Image after applying Sobel filter (from Wikipedia)

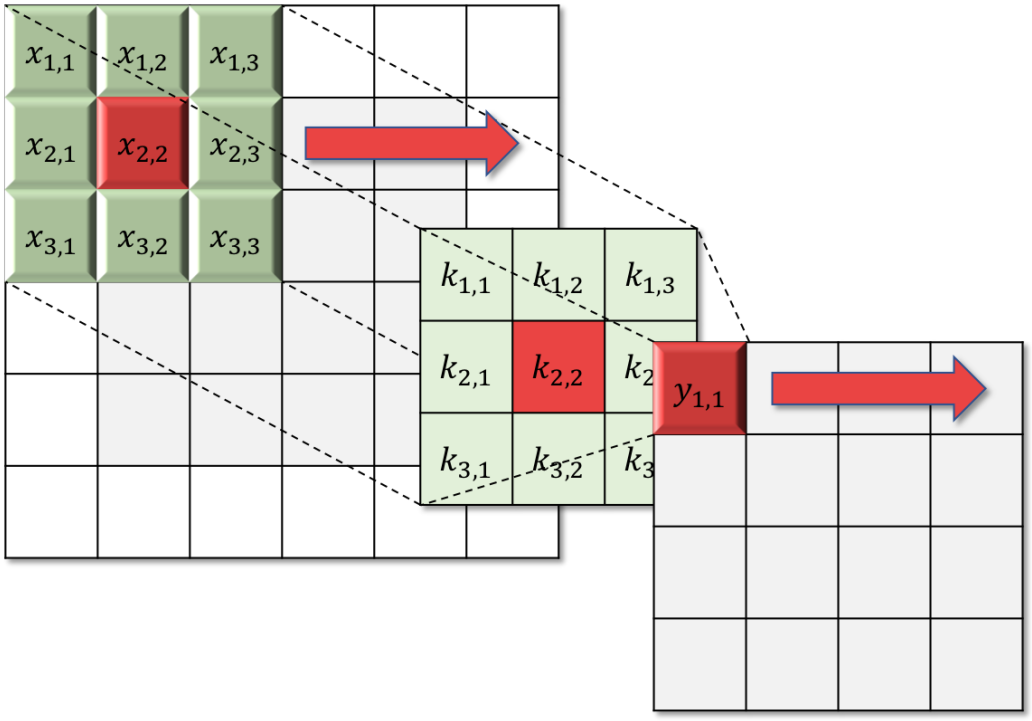

Figure 5: Convolution 연산을 적용하는 과정

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

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

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 3×3 이므로, 6×6 입력에 적용하면 4×4 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 데이터에 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 생각 해 볼 수 있습니다. 그럼 문장은 여러 단어로 이루어져 있고, 각각의 단어는

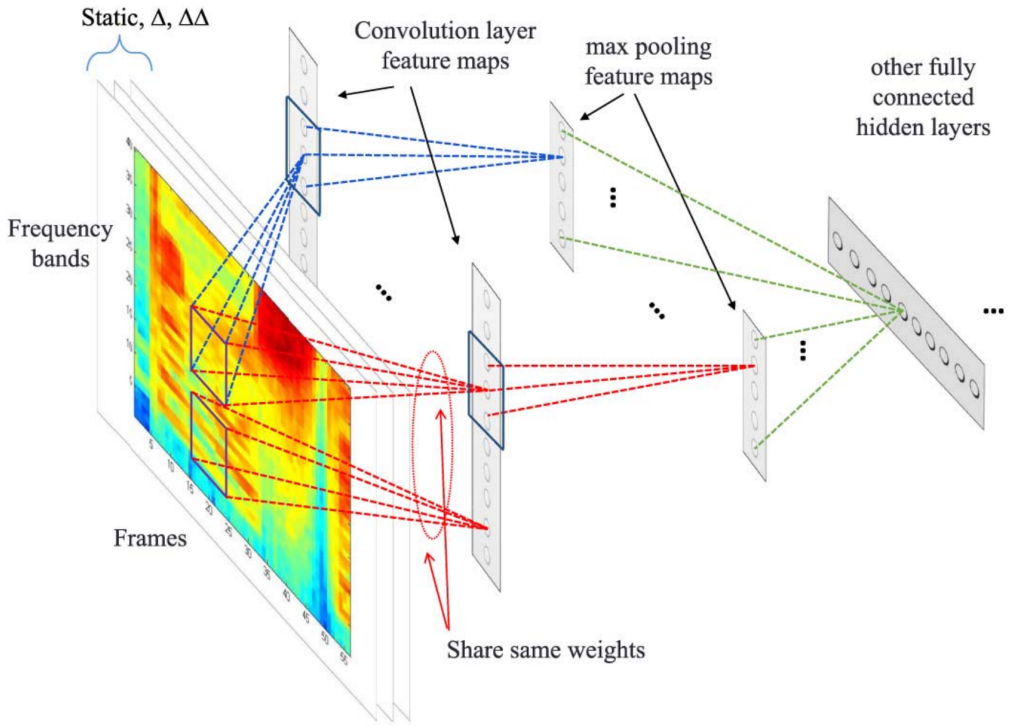

Figure 6: Example of convolutional neural network for speech recognition Abdel-Hamid et al.2014

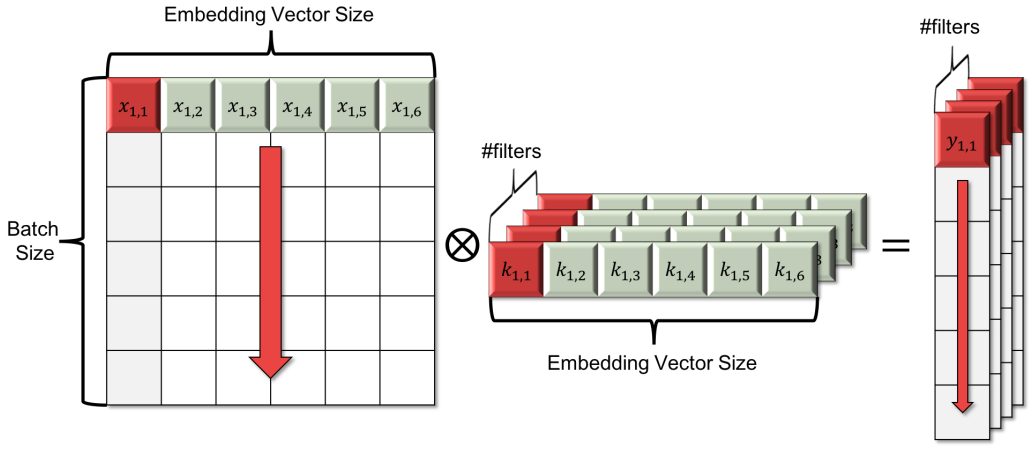

Figure 7: 1D Convolutional neural network

embedding layer를 통해 embedding vector로 변환 된 상태 입니다. 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 at el.2014]에서는 이를 이용하여 CNN 레이어만을 사용한 훌륭한 성능의 텍스트 분류 방법을 제시하였습니다.

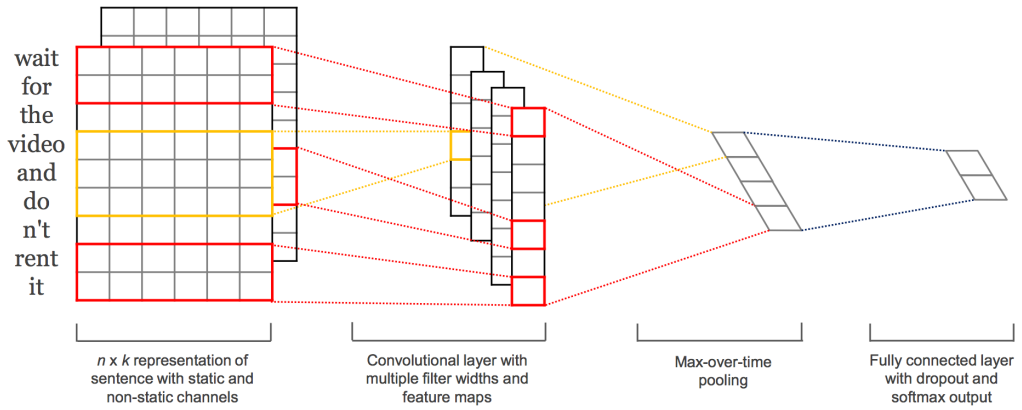

Figure 8: CNN for text classification arthictecture [Kim at el.2014]

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 것 입니다. 이때, max pooling layer를 적용하여 가변 길이의 변수를 제거할 수 있습니다. Max pooling 결과의 크기는 필터의 갯수와 같을 것 입니다. 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 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
```

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) 데이터 입니다. 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,
```

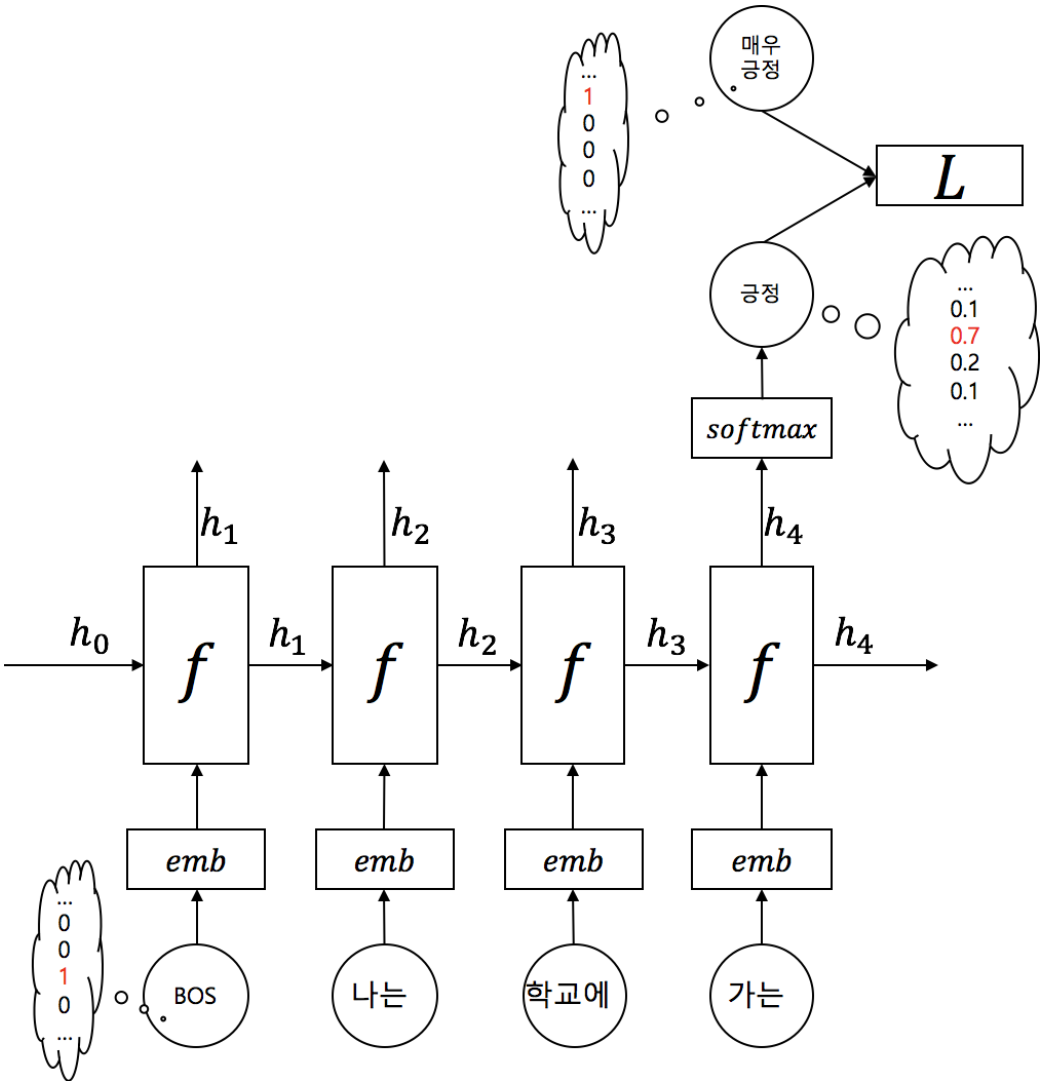

Figure 9: RNN의 마지막 time-step의 출력을 사용 하는 경우

```
                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 소스코드 입니다. 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

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 fot 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, vali
                                                                       batch_size=b
                                                                       device='cuda
                                                                       shuffle=shuf
                                                                       sort_key=lam
                                                                       sort_within_
                                                                       )

        # 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())

            # Caluclation 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=%.4




            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
                                                                         optim
                                                                         batch
                                                                         verbo
                                                                         )

        _, 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]
                 ):
        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 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=No


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 con
                                    )

    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]] fo

    if __name__ == '__main__':
        config = define_argparser()
        main(config)
```