Korean Text Generation with LSTM model

B611137 이강복

1. 프로그램의 목적

Numpy(Cupy) 라이브러리만 활용해 LSTM을 구축해 보고, 수식으로 만들어진 모델이 잘 적용되는지 Text_Generation을 통해 확인하고, 기존 Keras에 있는 RNN관련 모델들(SimpleRNN, LSTM)과 비교해 본다. 여기서 Text_Generation은, 불완전한 문장이 입력되었을 때 그다음에 나올 단어들을 예측하고 예측해서 문장의 끝이 나올 때까지 단어를 출력하는 방법이다.

2. 관련 연구

RNN은 시계열 데이터를 분석해서 주식가격 같은 것을 예측해서 언제 사고팔았는지 알려 줄 수 있고, 자율주행 시스템에서는 차의 이동 경로를 예측하고 사고를 피할 수 있도록 도울 수 있다. 또 RNN은 임의의 길이를 가진 Sequence, 즉 시계열 데이터를 다룰 수 있다. 예를 들면 문장, 문서, 오디오 샘플을 입력받을 수 있고, 지금 이 보고서에서 사용하는 Text Generation, 즉 자연어 처리(Natural Language Processing)에 매우 유용하게 사용할 수 있다.

RNN모델의 기본적인 원리는, 각 Time step마다 순환 뉴런(Recurrent neuron)이 현재 입력값 x_t와 이전 타임 스텝의 출력값 y_{t-1}을 입력으로 받아서 값을 예측하는 방식이다. 첫 번째 타임 스텝에서는 이전 출력이 없으므로 일반적으로 0으로 설정하고 진행한다.

RNN을 긴 Sequence로 훈련하려면, 많은 Time step에 걸쳐 훈련해야 하므로 훈련하는데 아주 오랜 시간이 걸리거나 훈련이 불안정할 수 있고 오랜 기간 훈련하게 되면 최종적으로 계산된 Gradient

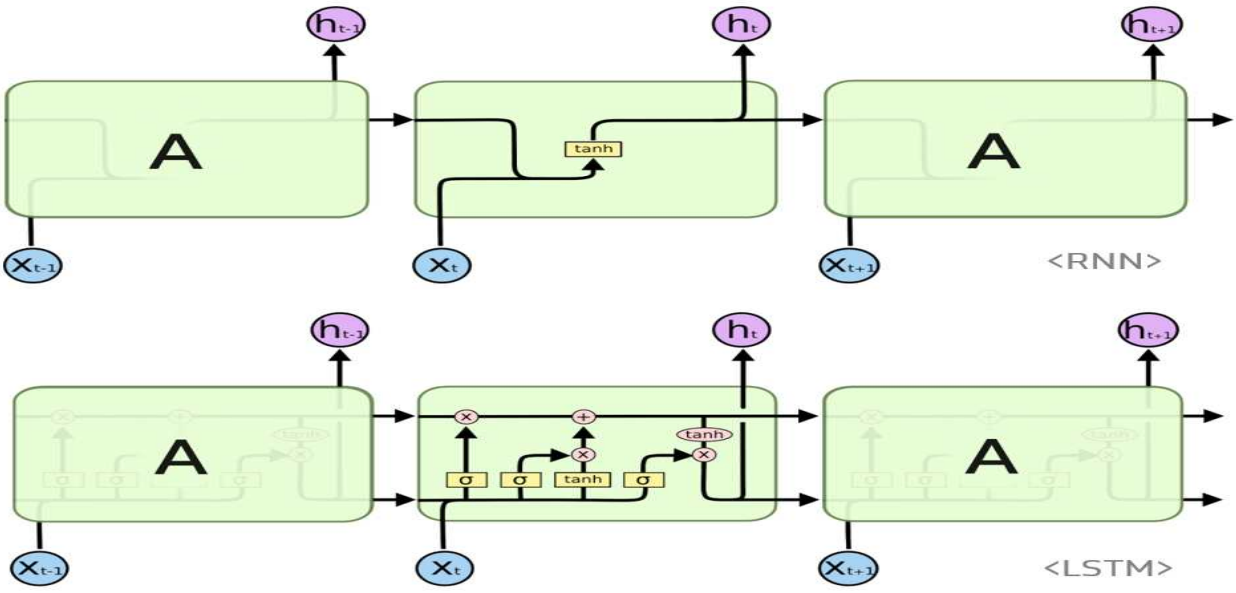
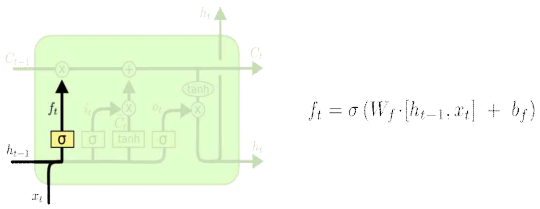


그림 1. RNN과 LSTM 모델

가 극도로 작아지게 되는 Vanishing Gradient Problem의 문제를 마주하게 된다. 그리고 Sequence가 긴 데이터를 처리할 때, 입력의 초반 부분을 조금씩 잊어버리게 되는 문제가 발생하게 된다. 이러한 문제를 해결하기 위해 나온 장기 메모리를 가진 모델이 바로 LSTM 모델이다.

LSTM은 Sepp Hochreiter와 Jurgen Schmidhuberd에 의해 1997년에 소개되었고, 다른 연구자들에 의해 수년간 점차 향상됐다. LSTM 모델을 기존 RNN 모델과 비슷하게 사용할 수 있지만, 훈련이 빠르게 수렴하고, 데이터에 있는 장기간의 의존성을 감지할 것이다.

LSTM의 핵심 아이디어는 네트워크가 장기 상태에 저장할 것, 버릴 것, 그리고 읽어 들일 것을 학습하는 것이다. 장기 기억 셀인 c_{t-1}은, 왼쪽에서 오른쪽으로 관통하면서 삭제 게이트(f_t)를 지나 일부 기억을 잃는다. 삭제 게이트는 activation function으로 시그모이드 함수를 사용하여 0에서 1 사이의 값이 나오는데, 0에 가까운 값이라면 이전 정보를 거의 잊어버리고, 1에 가깝다면 이전 정보를 그대로 보낸다고 생각하면 된다. 즉 삭제 게이트는 과거의 정보를 버릴지 말지 결정하는 게이트이다.

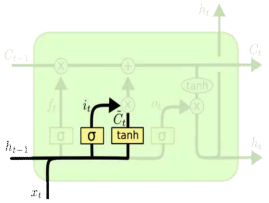


$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

그림 2. 삭제 게이트 f_t

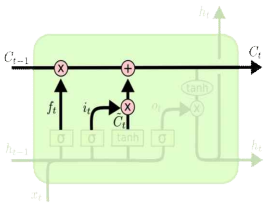
그런 다음 입력 게이트($i_t * \overline{c_t}$)에서 선택한 기억을 추가하기 위해 덧셈 연산을 해서 기억 일부를 추가한다. 입력 게이트는 현재 입력된 정보를 저장할지 말지를 결정하는 게이트이다.

이렇게 삭제 게이트와 입력 게이트가 설정되고, 이전 상태의 정보를 가지고 있는 c_{t-1}은, 삭제게이트와 입력게이트를 지나 새로운 정보 c_t가 만들어지게 된다.



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

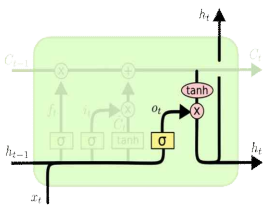
그림 3. 입력 게이트



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

그림 4. Update Cell

만들어진 c_t는 바로 출력으로 보내진다. 그래서 매 Time step 마다 일부 기억이 삭제되고 일부 기억이 추가가 된다. 또한 덧셈 연산 후 이 장기 상태(c_t)가 복사되어 tanh 함수로 전달되고, 이 결과는 출력 게이트(o_t)에 의해 걸러지게 된다. 걸러진 이 상태를 단기 상태(h_t)라고 하고, 이것은 이 Time step에서의 출력 y_t와 동일하고 다음 Cell의 Input으로 들어가게 된다.



$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

그림 5. 출력 게이트

3. 사용 플랫폼

LSTM 모델 구축을 위해 Numpy 라이브러리를 사용하나, GPU 기반 환경을 위해 Numpy처럼 사용할 수 있는 Cupy를 사용하였고, 기존 모델의 학습과 단어의 토큰화를 위해 Keras를 사용하였고, 한국어 문장을 형태소 단위로 분석하기 위해

koNLPy 라이브러리의 Okt를 사용하였다.

4. 데이터

4.1 데이터 출처

데이터는 국립국어원 모두의 말뭉치 사이트(https://corpus.korean.go.kr/main.do) 에 있는 국립국어원 신문 말뭉치(.ver 1)를 사용하였고, 그 중 첫 번째 파일인 NIRW1900000001.json 파일을 사용하였다.

그림 6. NIRW1900000001.json

4.2 데이터 형식

데이터 파일은 JSON 파일이고, 파일 형식은 그림 1과 같다. 실제 데이터가 담겨있는 'paragraph' 는 총 19354번째 paragraph까지 존재하고, 각 paragraph에 존재하는 문장들은 'form'에 담겨있고, 이 'form'의 개수는 paragraph마다 각각 다르다. 또 각 paragraph 안에 form에 해당하는 내용에 꼭 한 개의 문장이 들어있는 것이 아닌 여러 개의 문장이 담겨있는 일도 있다.

4.3 데이터 전처리

우선 python의 JSON 라이브러리를 통해 JSON 파일을 불러오고, paragraph의 form에 접근하여 문장을 추출한다. 첫 번째로 문장을 추출할 때. 각 paragraph의 첫 문장은 기사의 제목이기 때문에 문장이랑은 살짝 거리가 있는 것 같아서 제외해 주었다. 그리고 나머지 부분들을 문장으로 추출하는데 이때 괄호 안의 내용(ex) 이강복(24세)의 (24세))은 문장에 큰 영향을 안 끼친다고 판단하여 정규표현 식을 사용하여 제거하였고, 한 form에 여러 가지 문장이 있는 일도 있으므로 문장의 끝을 나타내는'.'을 기준으로 text를 split 하였다. 이때, 53.7%같이 소수점 숫자가 담겨있는 경우, 53과 7%로 이상하게 나뉘는데, 뒤에 소수점을 없애도 문장의 의미엔 크게 영향을 끼치지 않아서 split 하기 이전에 소수점 이하를 제거해 주었다. 이후엔 다시 정규표현 식을 사용하여 텍스트 안에 있는 특수문자들(", ', -, ... etc)을 제거해 준다. 이렇게 해서 문장이 완성되게 되면, 이전에 '.'을 기준으로 split 하여 '.'이 사라지게 되어서 문장의 끝을 알 수 없게 되기 때문에 문장의 끝에 eos(End of Sentence) 라는 단어를 붙여 문장이 마지막이라는 것을 뜻하는 토큰으로 사용하였다.

4.4 데이터 토큰화

이렇게 전처리한 문장들을 정수로 토큰화하는 작업은, Keras에 있는 Tokenizer를 사용해 토큰화하였다. 이전 과정에서 전처리한 문장들을 Tokenizer에 fit 시키면, 문장에 존재하는 단어들은 단어마다 정수 Index가 부여된다. 그리고 인코딩된 단어들을 이용하여, 각 문장을 여러 줄로 분해하여 훈련 데이터를 구성한다. 예를 들어 '나는 밥을 먹는다' 라는 문장이 있다고 하면, 이 문장을 '나', '나는', '나는 밥', '나는 밥을', '나는 밥을 먹는다'와 같이 분해하여 데이터를 만들었다,

4.5 데이터 패딩 및 훈련 데이터 설정

문장마다 구성된 단어의 개수가 다르므로 위에서 만든 데이터는 각 문장의 길이에 따라 길이가 다르므로 데이터의 길이를 맞춰 주는 패딩 작업을 해야 한다. 패딩 작업을 하기 전에, 가장 긴 샘플의 길

이를 찾아 주고, 나머지 데이터들을 그 데이터의 길이로 맞춰 패딩을 해 준다. 패딩을 할 때 최대 샘플의 길이보다 짧은 경우에는 앞에 0으로 패딩하였다. 이렇게 데이터 패딩까지 완료를 하면, 최종 데이터의 맨 마지막을 y로, 나머지 데이터를 x로 하여 x가 입력되었을 때 출력이 y가 나올 수 있도록 데이터 세트를 만들어 주었다. 그리고 데이터 y에 대해서는 One-Hot Encoding을 수행하였다.

5. 시스템 모델

입력 계층(Input Layer)과 출력 계층(Output Layer)의 크기는 단어를 토큰화하였을 때 나오는 전체 단어 사전의 크기(Vocabulary Size)로 설정하고, 이렇게 시퀀스로 입력되는 값들은 임베딩 계층(Embedding Layer)을 거쳐서 원-핫 벡터로 표현되어있는 단어들의 차원 수를 줄여주게 된다. 이렇게 차원이 줄어들게 된 단어 벡터들은 각 모델의 은닉 계층(Hidden Layer)으로 들어가게 되고, 출력 계층으로 다시 나오게 된다. 이 보고서에서 사용하는 시스템 모델의 은닉 계층은 하나의 계층으로 이루어져 있다. 그 그림은 아래 그림 7과 같다.

이렇게 학습된 모델을 통해 Text_Generation을 실시하는데, 예를 들어 '나는 밥'이라는 불완전한 문장이 입력으로 주어지게 되면, '나는 밥'이라는 입력에 대한 다음 출력을 predict 하여 그중 확률이 가장 높은 단어를 이 불완전한 문장에 붙인다. 저 predict의 결과로'을' 이 나왔다고 하면, 이제 다음 입력으로는 '나는 밥을' 이 되고, 이어서 계속 진행을 하여, eos 가 나올 때까지 계속 반복을 진행한다.

6. 실험 준비

실험은 총 두 가지로 실시한다. 먼저 기존 Keras에 있는 RNN과 LSTM 모델을 먼저 비교하는 동시에 형태소 단위 분석과 단어 단위 분석의 차이를 보는 실험을 할 것이고, 그 다음에 Numpy(Cupy)로 만든 LSTM모델과 이 모델들을 비교하는 실험을 할 예정이다.

먼저, 첫 번째 실험에서는, 학습하는 데이터의 총 문장 개수는 1117개이고, 그에 따른 단어 사전의 크기(Vocabulary Size)는 형태소 단위와 단어 단위가 각각 5478개, 7373개이고, 임베딩 계층의 크기는 10, 모델의 은닉 뉴런 수는 128, 손실 함수(Loss Function)은 Cross_Entropy, Optimizer는 'adam'을 사용하고 epochs는 150, batch_size는

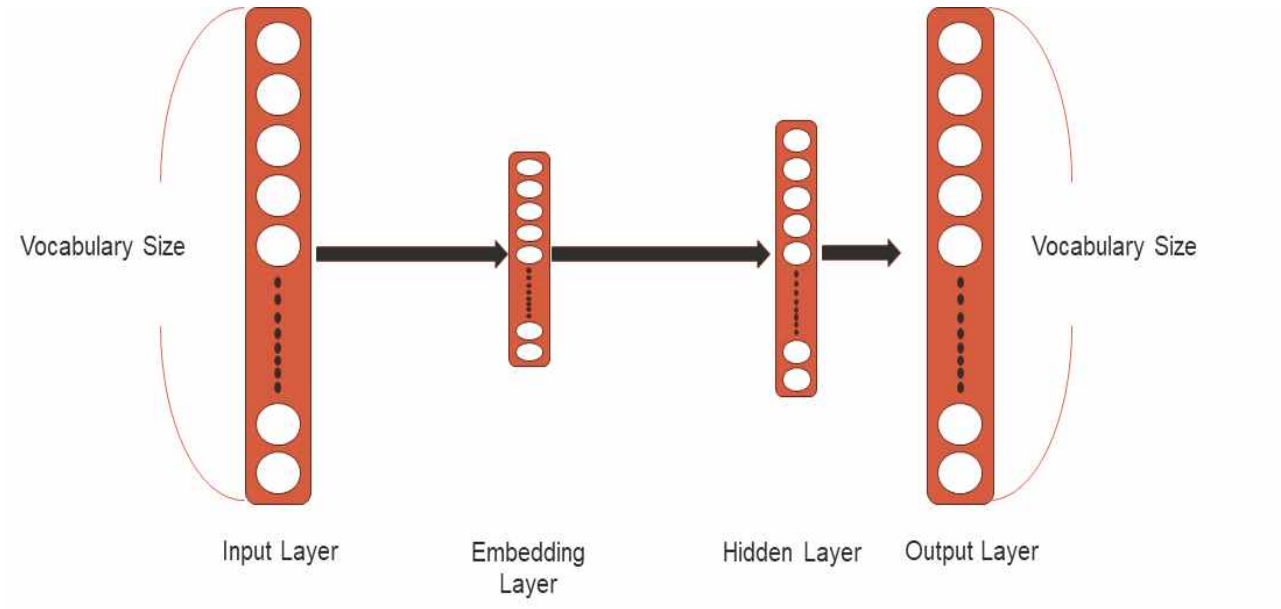


그림 7. Model Architecture

50으로 SimpleRNN 모델과 LSTM모델 모두 동일하게 설정을 하였고, model 구현 부분에서 두 은닉층의 모델이 다른 것 외에는 전부 똑같이 설정을 하고 실험을 진행 하였다. 정리하면 아래의 표 1과 같다.

Model	SimpleRNN	LSTM	SimpleRNN	LSTM
Preprocess	Morphs		Word	
Data	NIRW1900000001.json			
Sentences	1117			
Input Data / Max_len	18744 / 46		11939 / 29	
Vocabulary size	5478		7373	
Embedding Size	10			
Hidden Size	128			
Optimizer	Adam			
Loss Function	Cross_Entropy			
epochs	100			
batch	50			

표 1. Test 1 Info.

두 번째 실험에서는, my_LSTM 모델을 추가해서 비교하는 실험이다. 학습하는 데이터의 총 문장 개수는 1117개이고, 그에 따른 단어 사전의 크기는 7373개, 임베딩 계층의 크기는 10, 은닉층의 뉴런 개수는 128, 손실함수는 Cross_Entropy를 사용하고 Optimizer는 Adam, epochs는 50, batch_size는 50으로 세가지 모델 모두 동일하게 설정 하였다. 정리하면 아래의 표 2와 같다.

Model	SimpleRNN	LSTM	my_LSTM
Preprocess	Word		
Data	NIRW1900000001.json		
Sentences	1117		
Input Data	12331		
Vocabulary Size	7373		
Embedding Size	10		
Hidden Size	128		
Optimizer	Adam		
Loss Function	Cross_Entropy		
epochs	50		
batch_size	50		

표 2. Test 2 Info.

하드웨어는 Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz, RAM 16.0GB, Windows 10-64bit, NVIDIA GeFore GTX 1060 3GB 환경에서 실행하였다.

7. 실험 결과 및 분석

7.1 Test 1 결과

아래 표 3와 표 4에는 각각 형태소 분석을 사용한 Text_Generation과 단어 분석을 사용한 Text_Generation의 결과가 각각 나와있다. 표 3는 입력 값으로 '어린 아이'를 입력으로 한 예시이고, 표 4는 입력 값으로 '그리고 그는'을 입력으로 한 Text_Generation 이다. Perplexity는 test 데이터 10개를 실험하였을 때 나온 Perplexity의 평균으로 계산을 하였다. Test 1을 통해, LSTM 모델은 단어 분석 보다 형태소 분석에서 우위를 보였고, SimpleRNN 모델은 예상과는 다르게 단어 분석을 사용한 Text_Generation에서 LSTM보다 낮은 Perplexity를 보여 주었다. 물론 데이터의 크기와 epoch의 수가 알맞지 않아 발생한 일일 수도 있지만, 시퀀스의 길이를 결정하는 max_len 의 차

이로 인해 이러한 성능 차이가 발생한 것이 아닐까 라고 생각한다. 장기 의존성의 장점을 가진 LSTM 이 좀더 긴 Sequence인 형태소 분석에서 우위를 보였다고 생각한다.

Model	Perplexity	Text_generation Example
Simple RNN	3374	**어린 아이** 는 없이 이어졌습니다.
LSTM	2627	**어린 아이** 의 의견을 존중했을 뿐입니다.

표 3. 형태소 분석을 사용한 Text_Generation

Model	Perplexity	Text_generation Example
Simple RNN	3163	**그리고 그는** 이어 예전에 1월1일 하루만 쉬었는데 이번엔 시장 전체가 오늘부터 4일간 쉰다고 덧붙였다.
LSTM	3347	**그리고 그는** 농어가 아닌 죽은지 얼마 안 된 선어다.

표 4. 단어 분석을 사용한 Text_Generation

7.2 Test 2 결과

Test2 에서는 예상과는 다른 결과가 나왔다. Test1에서도 Perplexity가 높게 나와서 이상하다고 생각했는데 Test2 에선 더 높게 나왔다. 그런데 my_LSTM 모델에서는 오히려 Perplexity가 낮게 나왔지만 문장의 흐름이 어색한 것을 느낄 수 있다. 세 모델 다 입력 값으로는 '텔레비전 광고'를 넣었다.

Model	Perplexity	Text_generation Example
Simple RNN	5408	**텔레비전 광고** 이 남았다 생활를 한 로 결혼을 많이 했습니다.
LSTM	5705	**텔레비전 광고** 는 에 따라서 쓰고를 하고 하고를 오래되었는데 합니다.
My_LSTM	18	**텔레비전 광고** 세워져 2월 내리면 이 이 이을 지키기을 선뜻 것 이을 이을을 원하자 것을 이 것을 이을 내어 이을 맞추어.

표 5. Test2 실험 결과

그리고 my_LSTM에 혹시 문제가 있는 것은 아닐 까 해서 Loss History를 그래프로 나타내 보았을 때 점점 감소하고 있는 모습을 띄고 있어서 epoch가 더 컸다면 제대로 학습이 되지 않았을까 하는 생각도 있다.

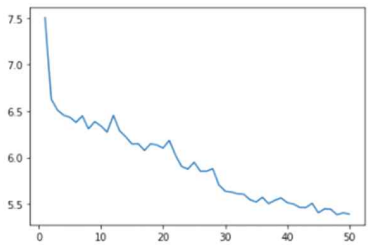


그림 8. Loss_History

8. 결론

전체적으로 기존 모델의 Perplexity가 너무 높게 나와서 당황을 했는데, 이는 훈련 데이터 Size, 훈련 epochs 설정 등 충분한 훈련을 거치지 못해 발생한 문제 인 것 같다. 그리고 직접 만든 LSTM 모델에도 학습 방법이나 가중치 Update 간에 문제가 있어서 과적합이 일어난 것 같다.

그래도 Test들을 통해 얻어낼 점들도 분명히 있었다. Test1을 통하여 Sequence(max_len)의 길이에 따라 RNN모델도 LSTM만큼 좋은 성능을 낼 수 있다고 생각하였다. 또 Test2에서 직접적인 성능 비교를 할 수는 없었으나, 직접 만든 LSTM 모델도 점점 Loss값이 작아지는 것을 확인하여 학습이 진행되고 있다 라는 것을 알 수 있었고, Test1의 단어 분석을 사용한 결과와 Test2의 기존 모델의 성능 차이를 통해, 훈련 epoch를 어떻게 설정하냐가 성능에 상당히 영향을 미칠 것이다 라는 것을 알게 되었다.

9. Contribution

9.1 데이터 전처리

json 파일에서 데이터를 가져온 후, 정규표현식을 통해 문장을 정제하는 과정은 직접 실행하였다.

이후에 데이터를 정수 인코딩을 통해 토큰화 하고 패딩을 거쳐서 훈련 데이터 집합을 만드는 과정은 모두 (https://wikidocs.net/45101) 에서 참고를 하였고, 훈련 데이터를 생성하는 코드 중간에 Okt 라이브러리를 사용하여 한글 문장을 형태소 단위로 나눠주는 코드를 추가 해 주었다.

9.2 모델 학습

기존 keras에 있는 모델을 이용하여 모델을 설계하는 코드는 (https://wikidocs.net/45101)에서 참고를 하여 모델을 만들었고, Numpy(Cupy)라이브러리를 이용한 LSTM 코드는 Numpy 라이브러리를 통해 min-char-RNN을 구현 한 코드인 https://gist.github.com/karpathy/d4dee566867f8291f086 를 참고하여 LSTM 코드를 만들었다. 훈련에 사용되는 Adam Optimizer은 다음 링크 https://arxiv.org/pdf/1412.6980.pdf 에 있는 알고리즘 내용을 참고를 하여 직접 구현하였다.

9.3 성능평가 및 Text_Generation

마지막으로 학습한 모델을 바탕으로 Text_Generation을 발생시키는 함수는 (https://wikidocs.net/45101) 에서 참고를 하여 함수를 만들었으며, Perplexity를 계산하기 위해 predict_classes가 아닌 predict를 사용 후 softmax를 통해 확률을 계산하여 Perplexity를 구하는데 사용하였고, 기존 함수와 다르게 eos 토큰이 나올 때 까지 반복문을 돌려 문장이 끝이 나올 때 함수를 끝내도록 수정을 하였다.

10. 재현 방법

10.0 환경 설정

Anaconda를 통해 가상환경을 먼저 설정을 해야 한다.

https://www.anaconda.com/products/individual 에서 다운로드 후 아나콘다를 실행하고 가상환경을 만들어 주어야 한다. cd 명령어를 통해 venv.yml 파일이 있는 곳 까지 이동 한후 다음과 같은 명령어를 입력해 준다.

conda env create -n venv —file venv.yml

명령어를 입력하고 나면 venv에 필요한 라이브러리들이 설치가 된다.

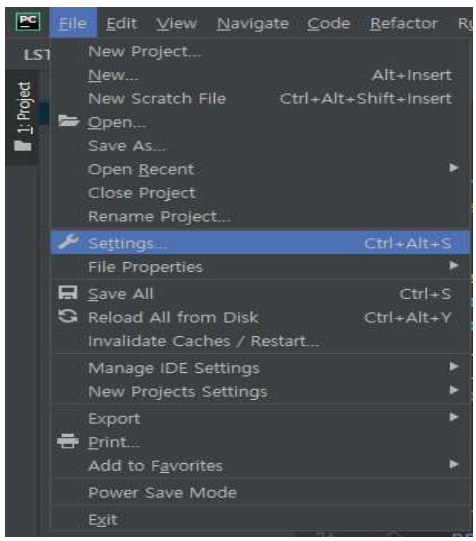
그 다음 절차로는 Pycharm 이 설치가 되어있어야 한다.

https://www.jetbrains.com/ko-kr/pycharm/download/#section=windows사이트에서 community 버전을 받으면 된다.

그리고 GPU 기반 환경을 위해 GTX 1050 이상의 그래픽 카드가 필요하다 (CUDA 파스칼 이상)

Pycharm 실행 후에는

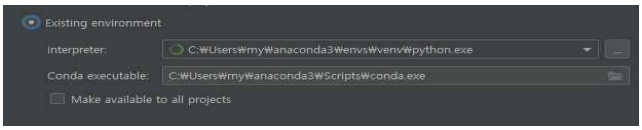
1. 왼쪽 위의 File의 Settings을 들어간다.



2. 왼쪽에 있는 Python Interpreter을 클릭한다.

3. 다음 그림의 오른쪽에있는 톱니바퀴를 누른다. 이후 Add를 누른다.

4. 이후에 왼쪽에서 Conda Environment를 클릭 한 다음 Existing environment에서 방금 전에 설치 하였던 venv 관련 파일이 자동으로 등록이 되어있을 것이다. 그러면 ok를 누른다.



5. 오른쪽 아래의 Apply를 누르면 가상환경이 등록이 된다.

6. 그리고 아래에 나와있는 함수들을 호출해 main함수를 실행하면 된다.

10.1 함수 설명

- data_load(num, morph) : json파일에서 num 개의 document에서 문장을 추출하는 함수이다. morph = True로 설정시 형태소 단위로 문장을 나누어 주고, 그렇지 않으면 단어 단위로 문장을 나누게 된다.

- my_model_train(e, b_size) : Cupy로 만든 LSTM 모델의 훈련을 한다. hidden : 128, Embedding : 10, output : vocab_siz로 설정 되어 있고, epoch = e, batch : b_size으로 설정되어 있다. 훈련된 모델은 같은 파일 디렉토리에 .npy형식으로 가중치와 Loss_History가 저장이 된다.

- model_train(e, b_size) : Keras에 존재하는 RNN모델과 LSTM 모델을 훈련한다. 은닉층은 하나씩이며 뉴런 수는 128, Embedding : 10, activation : softmax, loss : cross_entropy, optimizer : adam으로 설정되어있고 epoch : e, batch_size = b_size 이다. 모델 훈련이 끝나게 되면 같은 파일 디렉토리에 .h5파일로 저장이 된다.

- get_model_test() : test2를 실시하는 함수이다. 이미 학습된 모델을 가져와 테스트 하는 것이므로, 이 함수를 실행시키려면 초반의 data_load부분에서 data_load(30, False)를 입력해야 한다.

- get_model_morphs() : 형태소 단위로 토큰화가 된 rnn과 lstm모델을 test하고 결과를 보여준다. 이 때 앞부분 데이터 로드 시 data_load(30, True)를 입력해야 실행이 된다.

- get_model_words() : 단어 단위로 토큰화가 된 RNN과 LSTM 모델을 test하고 결과를 보여준다. 이때 앞부분 데이터 로드시 data_load(30, False)를 입력해야 실행이 된다.

- text_generation(model, word, npy) : 모델을 이용해 문장을 만들어 주는 함수이다. 불완전한 문장인 word 입력시 문장을 만들어 주게 된다. 리턴값으로는 그 문장의 perplexity를 리턴한다. 그리고 npy의 경우는 Keras 모델로 사용할 시에는 Numpy, 내가 만든 모델의 경우에는 Cupy를 사용해야 하기 때문에 구별을 위해 넣었고, npy = True일 때 numpy를 사용한다. 그리고 max_len 보다 길이가 길어지게 된다면 함수가 리턴된다.

- main : 메인 함수에는 data_load 함수 후에 문장을 토큰화 하고 훈련 Sequence를 만들고 데이터 패딩까지 실시하는 코드가 존재한다. 이곳은 건드리지 않아도 된다.

10.2 Test 재현 방법

10.2.1. Test 1 재현하기

word 단위로 나뉘어진 결과를 보고 싶다면 main 함수 첫 부분에서

sentences = data_load(30,False)

를 넣어 준 후 이후 main함수 맨 마지막에

get_model_word()

를 호출하면 된다.

형태소 단위로 나뉘어진 결과를 보고 싶다면 main함수 첫 부분에서

sentences = data_load(30, True)

로 설정 한 다음 main함수 맨 마지막에

get_model_morphs()

를 호출하면 된다.

10.2.2 Test 2 재현하기

우선 메인함수 처음에

sentences = data_load(30, False)

를 해 준후 main함수 마지막 부분에

get_model_test()

를 호출하면 결과를 확인할 수 있다.

출처 및 참고문헌

https://wegonnamakeit.tistory.com/7 LSTM그림
https://wikidocs.net/45101/ 토큰화
https://arxiv.org/pdf/1412.6980.pdf ADAM
https://gist.github.com/karpathy/d4dee566867f8291f086 min-char-rnn numpy
Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn. Keras &TensorFlow* 2판 (n.p.: 한빛미디어, n.d.), 597-619.

Kim, Y.-h., Hwang, Y.-k., Kang, T.-g., and Jung, K.-m. "LSTM Language Model Based Korean Sentence Generation." The Journal of Korean Institute of Communications and Information Sciences, Vol. 41, No. 5 (2016), 592~601.