# Sequence to Sequence: Encoder + Decoder + Generator

텍스트, 도표, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

기본적인 sequence to sequence 구조이며, Encoder는 Many-to-one, Decoder는 one-to-many 방식을 가진다. 이것은 종합적으로 one-to-many구조로 취급한다.

Decoder 부분에서 Training할때와 inference할때는 Teacher forcing에 대한 차이점을 가진다. 나중에 구조로 확인하자.

인코더와 디코더를 통해 압축과 해제를 하기 때문에 auto encoder와 비슷하다.

\* Sequence to Sequence Equation

- Given dataset,

x~p(x): x는 p(x)에서 sampling, y~p(y|x): x가 주어졌을 때 y의 분포를 sampling

- Find parameter that maximize likelihood

- Minimize loss function by update parameter with gradient descent (Loss minimize)

* Encoder

Encoder는 source문장을 압축한 context vector를 decoder에 넘겨준다. (문장을 하나의 vector로 만들어준다.) Encoder는 train/test시에 항상 문장 전체를 받기 때문에, Encoder자체만 보면 non-auto regressive하기 때문에 bi-directional RNN이 사용가능 하다

\* Encoder Equation

- Given dataset,

- Can get hidden state of encoder

- If we use bi-directional

샘플링을 통해서 x와 y의 pair를 받는다. x는 m개, y는 n개

가 encoder의 embedding layer를 통과하면, =(bs, 1, |v|) -> (bs, 1, ws)

, ws is word embedding vector.

은 이전 time step의 encoder의 hidden state

* Decoder

Encoder의 마지막 hidden state를 decoder의 첫 hidden state로 받는다.

Decoder는 conditional language model(one-to-many)이다. 인코더로부터 문장을 압축한 context vector를 바탕으로 문장을 생성한다.

Auto-regressive task에 속하므로, uni-directional RNN을 사용한다.

인코더의 마지막 hidden state는 디코더의 첫 hidden state다. 하지만 인코더의 hidden state는 bi-directional RNN을 사용하기 때문에 2개, 디코더에선 uni-directional RNN을 사용하기 때문에 1개다. 이것은 attention 메커니즘으로 해결할 수 있다.

\* Decoder Equation

- Given dataset,

- We can get hidden state of decoder,

* Generator

Decoder로부터 hidden state를 받아서, Linear layer와 softmax를 거쳐서 해당 time step의 단어를 예측한다.

Generator는 디코더의 hidden state를 받아 현재 time-step의 출력 token에 대한 확률 분포(multinoulli distribution)반환한다.

단어를 선택하는 문제이므로 cross entropy loss를 통해 최적화가 가능하고, ground truth분포와 모델 분포 차이를 최소화하기 위함이다. 조건부 언어모델로 볼 수 있으므로 PPL로 치환 가능하다.

\* Generator Equation

- Given dataset,

- Hidden states from decoder can be calculated like as below.

- Generator returns a probability distribution of current output token.

\* Loss function

- We need to minimize negative log-likelihood,

- Log-likelihood can be calculated like as below:

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

디코더는 생성모델이기 때문에<EOS>, <BOS>가 존재한다.

# Attention 어텐션

RNN에 기반한 seq2seq 모델에는 크게 두 가지 문제가 있다.

첫째, 하나의 고정된 크기의 벡터에 모든 정보를 압축하려고 하니까 정보 손실이 발생하고,

둘째, RNN의 고질적인 문제인 기울기 소실(vanishing gradient) 문제가 존재한다

결국 이는 기계 번역 분야에서 입력 문장이 길면 번역 품질이 떨어진다. 이를 위한 대안으로 입력 시퀀스가 길어지면 출력 시퀀스의 정확도가 떨어지는 것을 보정해주기 위한 등장한 기법이 어텐션이다. 문장의 모든 length를 받을 수 없던 문제를 해결한다.

어텐션의 기본 아이디어는 디코더에서 출력 단어를 예측하는 매 시점(time step)마다, Query를 보냄으로써, 인코더에서의 전체 입력 문장을 다시 한번 참고한다는 것이다. 단, 전체 입력 문장을 전부 다 동일한 비율로 참고하는 것이 아니라, 해당 시점에서 예측해야 할 단어와 연관이 있는 입력 단어 부분을 좀 더 집중(attention)해서 보게 된다.

- 기존의 Key-Value 함수와 달리, Query와 Key의 유사도에 따라 Value를 반환한다.

**- Decoder RNN(LSTM)의 hidden state의 한계로 인해 부족한 정보를 직접 encoder에 조회(query)하여 예측에 필요한 정보를 얻어오는 과정이며, 정보를 잘 얻기 위해서는 Query를 잘 만들어내는 과정을 학습한다.**

-Attention in seq2seq

Query: 현재 time-step의 decoder output (원하는 정보 표현)

Keys: 각 time-step 별 encoder output (쿼리와 연관된 정보)

Values: 각 time-step 별 encoder output (키에 해당하는 정보)

Query로부터 encoder 정보를 조회

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

디코더에서 인코더에게 query를 보내고, 이것을 적용해서 유사도를 계산한다.

\* BMM (Batch Matrix Multiplicatio)

우리는 어텐션을 수행하기 전에 Batch Matrix Multiplication(BMM)을 수행해야 한다.

스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

뒤에 두개만 연산. K가 같아야 한다

\*Attention Equation

- With entire encoder’s encoder’s hidden states and current decoder’s hidden states,

위 식을 전개하면,

- Re-define decoder’s hidden state, and feed into generator,

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

- Attention은 미분가능한 key-value- function이다. Attention 함수의 입력은 query, key, value다.

- 정보를 잘 얻기 위한 Query를 변환하는 방법을 배우는 과정이다. – query와 key의 곱에다가 softmax를 곱해주면 attention weight이 나온다. 즉 query와 key의 유사도 계산이다.

- Attention을 통해 RNN의 hidden state의 한계를 극복할 수 있다.

(LSTM을 쓰더라도 context vector에 모든 정보를 담기에는 어려운 한계가 있고, 더 긴 길이의 입력/출력에도 대처가 할 수 있게 된다.) 즉, 긴 length를 입력으로 받을 수 있다.

1. : 한 time step에 decoder의 hidden state, - decoder의 hidden state

2. : 모든 time step에 encoder의 hidden state, - encoder의 hidden state

3. (변형된 쿼리): query = self.linear(h\_t\_tgt), , h\_t\_tgt=

4. , , weight = torch.bmm(query, h\_src\_transpose(1,2))

5. =weight

6. : context\_vector = torch.bmm(weight,h\_src), w is weight, v is value

# Masking

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

우리는 항상 미니배치를 병렬적 구조를 가진다. 따라서 어텐션 가중치를 빈공간이 할당 되어야한다. 가장 긴문장 떄문에 짧은 문장들이 pad로 채워진다. 이 pad가 문제가 된다.

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

pad에도 attention가중치가 들어가는 문제가 발생한다. – garbage값.

대충 맞긴 하지만, 번역의 순서나 맞춤법들이 달라진다. 해결할 안전장치가 필요하다.

도표, 라인, 사각형이(가) 표시된 사진

자동 생성된 설명

pad부분을 -inf로 채우고 softmax값을 씌어주면 0이된다.

\* softmax이전의 In equations

- 미니배치 내의 문장 구성에 따라 <pad>가 동적으로 생성된다. Pad의 hidden state에는 attention weight가 할당되면 안됀다.

- 따라서 key, query의 dot product이후이(softmax이전에), masking을 통해 <pad>위치의 값을 임의 무한대로 변경한다.

- Transformer에서도 유용하게 쓰인다.

# Input feeding

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

동그라미 부분이 Input feeding이다. (; 는 concatenation을 의미)

는 softmax를 통과하기 이전의 가중치이며, 이것을 입력값의 embedding 값과 concatenation한다.

1. 를 Embedding과 concat해서 이전정보를 유지한다.

2. 이전 time step의 출력을 보완하는 방법이다. (이전 time step의 가중치를 유지함으로써)

이 방법은 sampling과정에서 손실되는 정보를 최소화한다.

또한 이전 예측값을 무시하고 정답을 입력으로 받는 Teacher Forcing으로 인한 학습/추론 사이의 괴리를 최소화할 수 있다.

- Encoder

문장을 받아 context vector로 압축하고, bi-directional RNN을 통해 구현된다. (non-autoregressive속성)

- Decoder & generator

encoder로부터 문장을 받아 문장을 생성한다. Cross Entropy(PPL)를 통해 최적화하며, auto-regressive한 속성을 가진다.

- Input Feeding

Sampling 과정에서 손실된 정보를 word embedding에 concatenation해준다

- Attention

Key-value함수로, 디코더의 hidden state를 인코더의 각각 hidden state에 유사도를 비교하고, 좋은 query를 만들어내는 과정을 학습한다. 문장의 모든 length를 받을 수 없던 문제를 해결한다.

Query: 현재 time step의 decoder

Key, Value: 각 time step별 encoder

# Full equation

- Encoder

- Decoder

- Weight

- Decoder의

- Loss

- Log-likelihood can be calculated like as below:

# Encoder to Decoder

스케치, 도표, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명

Decoder의 시작 hidden state는 1개: Uni-directional LSTM

Decoder의 시작 hidden state는 2개: Bi-directional LSTM

hs의 호환이 필요하다. 정해진 hidden size의 반큼만 인코더의 LSTM hs로 삼아준다. (hs/2)

# Teacher forcing

원래의 RNN에서는 이전 time step의 결과를 넣지만, 그냥 실제 정답을 넣어준다.

MLE수식상 Teacher forcing을 써야 할 수밖에 없다(정답을 RNN의 입력에 넣어줘야 한다.)

- Inference

- Auto-regressive: 과거 자신의 상태를 참조하여 현재 자신의 상태를 업데이트한다.

- Training 과정의 Decoder(Teacher forcing) / Inference 과정의 Decoder

도표, 평면도, 기술 도면, 텍스트이(가) 표시된 사진

자동 생성된 설명도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

Training 과정의 Decoder는 실제 정답을 받아오고,

Inference과정의 Decoder는 이전 time-step의 정답을 받아온다.

-Auto-regressive task를 feed-forward할때는 보통 이전 time-step의 이전 출력이 현재 time-step의 입력이 된다.

-MLE수식상 실제정답을 넣어주는 Teacher forcing을 통해 Auto-regressive task에 대한 sequential modeling을 할 수 있다 하지만 training model과 inference model의 괴리(discrepancy)가 생긴다. 훈련과 추론 방식이 달라짐으로써, 함수가 2개가 되고 코드 또한 2개로 짜야 한다

-강화학습(RL)을 통해 이러한 괴리를 없애고 성능을 높일 수 있다. 이 외에도 다양한 방법(professor forcing등)들이 제안됐다.

# Gradient Accumulation

\* Batch size

- 큰 배치사이즈는 epoch내의 forward & backward횟수를 줄여주어 학습의 속도를 높여준다.

- batch size에 따라 성능이 바뀔수도 있다. 작은 데이터셋에서는 local minima를 탈출할 수 있다고 알려져 있으나, 큰 데이터 셋에서는 큰 배치사이즈가 오히려 성능이 높아지기도 한다. -> 데이터가 많을 경우, 미니배치 사이즈가 작으면 gradient방향은 noise와 편향된 정보로 ground truth방향과 다를것이다. 큰 미니배치 사이즈는 ground truth 방향과 비슷할 것이다.

- 따라서 모델의 성능이 떨어지지 않는 한도 내에서 배치사이즈를 최대로 하여 학습을 빠르게 진행할 수 있다. 기본적으로 SGD를 사용할 경우 LR와 배치사이즈는 비례관계를 갖지만, Adam을 사용할경우 LR에 크게 신경쓰지 않는다.

Transformer에서는 batch size를 1000 이상으로 가져가기도 한다.

하지만 GPU가 허락하지 않는다. 따라서 우리는 gradient accumulate를 사용한다

\* Gradient Accumulation: 속도는 둘째치고 성능을 위해서라도 batch\_size를 크게 설정하고싶다.

-Forward & backward를 할 때마다 파라미터 업데이트(optimizer.step() 호출)를 하는 대신, gradient를 누적해서 누적해서 나중에 한번에 업데이트하는 방법이다.

- 마치 누적 횟수만큼의 배치사이즈가 증가된 효과이다. k번 accumulation = k \* batch size

- 속도 상의 이점은 없다. sample마다 미분해주고 더한 후(누적해서) 업데이트

optimizer.step() 호출 횟수만 다르고, feed forward, backward횟수는 같다

- Seq2seq에선 Adam optimizer기준, batch size는 256이 가장 좋은 성능이다.

--iteration\_per\_update 파라미터로 조절. Batch size는 64, iteration\_per\_update를 4로 조절하면 256이 된다. Feed forward가 4번 호출되어야, optimizer step을 호출한다

# Automatic Mixed Precision (AMP)

GPU의 한계로 인한 학습의 비효율성 해소할 수 있으면 좋을것이다.

- 메모리는 더 큰 모델로 더 큰 배치사이즈로 학습하고, 연산속도는 Float Point 16(half-precision)으로 연산할 경우 속도 및 메모리에 유리하기 때문에 연산속도가 증가한다.

- FP16의 경우 표현 범위의 한계가 있어, 너무 작은 값의 경우 0으로 표현되어 너무 작은 값으로 나눌경우 NAN으로, 너무 클 경우 inf로 표현된다.

- scaling을 통해 FP16에 표현 가능한 범위 내에서 연산을 수행하면 된다.

AMP를 통해 scaling을 하고 FP16으로 연산을 수행하자. 그렇다면 속도 증가와 메모리 사용량 감소를 기대할 수 있다. 속도는 더 빠르고, 성능은 비슷하거나 뛰어나다.

CPU에서는 사용 불가능 하다.

sclaer = GradScaler() #scaling

with autocast(): #autoc casting

output = model()

loss = loss\_fn(output, target)

# Greedy Search

- NLG는 가장 확률 높은 문장을 만들어내는 과정인 search problem에 속한다.

- Node로는 주어진 문장x, 이제까지 출현한 단어들 edge는 이다.

- NLG에서 Edge들의 합이 최대가 되는 경로를 찾는데, 경로의 시작은 <BOS>, 경로의 끝은<EOS>가 된다.

- Backtracking(Exhaustive search): 모든 경로를 다 가볼 수 있다면, 가장 정확한 정답을 맞출 수 있을 것이다 -> 현실적으로 불가능하다.

- Greedy Search: 모두 다 search 할 수 없으니, 눈앞의 최선의 확률을 찾는 것은 성능과 속도간 Sweet spot이라고 할 수 있다. (trade off) 최선은 아니지만 차선책이다. 이것은 NLG는 지금의 선택은 다음 time step의 결과가 바뀌는 auto-regressive task이기 때문이다. 이러한 차선을 여러 개 따지는 것을(top-k) 나중에 배우는 Beam Search라고 한다.

# PPL, BLEU Evaluation

번역을 평가하는 score metric인 BLEU

테스트 문장에 대해서 확률을 높게 반환할수록, PPL이 작을수록 좋은 모델이다. 번역에서도 PPL은 유효하긴 하지만 무조건적이지 않다.

\* PPL(perplexity): 역수를 취한 후 n제곱근을 해준다.

Time step별 branch수를 의미하며, PPL이 낮을수록 확률분포가 sharp하고, PPL이 클수록 확률분포가 Flat하다. 이것은 Entropy관점에서 볼 수 있다.

- Entropy and Perplexity

PPL은 어순의 변화에 취약하기 때문에 정확한 번역의 품질을 반영하지 못한다. 교착어에서 번역의 어순은 크게 상관없지만, 고립어의 경우 어순은 번역에 크게 영향을 미친다.

\* BLEU

: 각 n-gram별 precision의 가중 평균

BLEU Example: 2-gram Precision(Bi – gram 2개씩 단어씩)

|  |  |  |
| --- | --- | --- |
| Hyp1 | | |
| 2-gram | count | hit |
| <BOS> 재호는 | 1 | 1 |
| 재호는 진원이를 | 1 | 1 |
| 진원이를 많이 | 1 | 0 |
| 많이 사랑한다 | 1 | 1 |
| 사랑한다 . | 1 | 1 |
| . <EOS> | 1 | 1 |
| 합계 | 6 | 5 |
| Precision = | | |

|  |  |  |
| --- | --- | --- |
| Hyp2 | | |
| 2-gram | count | hit |
| <BOS> 재호는 | 1 | 1 |
| 재호는 진원이와 | 1 | 1 |
| 진원이와 제주도를 | 1 | 0 |
| 제주도를 여행한다 | 1 | 1 |
| 여행한다 . | 1 | 0 |
| . <EOS> | 1 | 1 |
| 합계 | 6 | 4 |
| Precision = | | |

보통 uni-gram, bi-gram, tri-gram, four-gram에 가중평균을 적용한다.

- perplexity(Cross Entropy) 은 낮은 값이 좋다. 어순의 변화에 취약하다

- BLEU는 높은 값이 좋다. (BLEU는 확률이니까) 어순의 변화에 ROBUST하다

- 하지만 BELU도 유의어/동의어 등에 대처는 떨어진다. 이때는 정성평가가 필요하다(Intrinsic Evaluation)

PPL은 left-to-right은 left-to-right관점에 다음 time-step의 단어에 대한 확률 분포를 수치화한다. 거시적 관점에서는 낮은 PPL에 좋은 성능을 가리키지만, 미시적인 관점에서는 작은 PPL의 차이는 모델의 우열을 가리기 힘들다.

그렇다면 BLEU를 통해 maximize하도록 하는방법은? 미분 불가능한 함수이므로 MLE에 적용하기 어렵다. 그래도 하지만 값을 구할 수는 있다! (구현된 오픈소스를 이용하자)

테스트셋 선정은 매우 중요한 문제다 난이도와 적합성 문제등에서 그렇다.

정량 평가(extrinsic evaluation)를 통해 최종 후보 모델을 선정하고,

정상 평가(intrinsic evaluation)를 통해 최종 배포를 결정한다.

정성 평가 시에는 일관성과 객관성(blind test등) 확보가 가장 중요하다

\* BLEU Preprocessing

En: moses tokenizer + bpe

Ko: mecab + bpe

모든 문장을 하기 어렵기 때문에 1000개만 진행하더라 직접 추론할떄 시간적으로 오래 걸리기때문인거같다

# Beam Search

Generation == Search Problem

- 가장 확률이 높은 문장을 만들어내는 것은 최단 경로 찾기와 만든 문제

- Greedy Search: 지금의 최선이 나중에는 나쁜 선택이 될 수 있다. – auto regressive하기 때문

- Beam Search: Top-k를 tracking하여 greedy search를 조금 더 안전하게 수행한다.

최선은 아니지만 차선의 답을 찾을 수 있다. – 엔지니어링적인 문제

Greedy Search는 Beam Search에서 beam size가 1인 경우다.

보통 beam size는 3~10으로 설정한다. 3~5

다음 그림은 k=3 (beam size가 3)인 경우다

도표, 텍스트, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

3개의 결과를 합쳐서 누적으로 top-k하는 방식이다. <EOS>가 K개 나올때까지 진행한다. K가 클수록 잘못된 값을 만회할수 있고, 성능도 증가하고, 메모리도 증가한다.

병렬로 처리하면, 속도를 적게 떨어뜨리고 성능이 올라간다. 미니배치가 K배 늘어난것처럼! 동시에 처리하면 softmax 가 동시에 나온다. 예를들면 bs가 256, k=3이면 는 256\*3의 사이즈다.

- auto regressive특성으로 인해 greedy search의 한계가 발생하는데 beam search로 완화한다.

- k(beam size)번의 반복 inference를 통해 조금 더 나은 성능의 추론 수행 가능한다. 하지만 sequential한 연산은 속도저하를 야기하기 때문에 병렬적으로 처리해야 한다.

- Mini-batch를 처리하는 것처럼 k번을 동시에 처리해야 한다.

- 따라서 Beam Search는 속도저하가 거이 없이, 모델 개선과 training corpus 데이터 수 증가가 없이도 성능이 올라간다.

\* Parallelized beam search

- 한 샘플에 대한 beam search를 수행할 때, k번의 inference가 수행된다. 마치 k개의 샘플에 대한 inference로 볼 수 있다.

- n샘플에 대한 beam search를 수행할 때, n\*k번의 inference가 수행된다. 마치 n\*k개의 샘플에 대한 inference로 볼 수 있다.

- 따라서 n\*k개 샘플의 mini batch에 대한 inference를 수행하면 된다.

도표, 텍스트, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

도표, 스케치, 기술 도면, 평면도이(가) 표시된 사진

자동 생성된 설명

- Mini batch parallelized beam search를 통해 성능과 속도 모두 잡을 수 있다. 하지만 무조건 속도가 빨라지는 것은 아니다.

- 실제 deploy환경에서는 beam 개수 조절을 통해 속도와 성능 사이의 trade-off를 조절한다

실제 배포를 위해서는 code profiling을 통해 속도 저하를 발생시키는 코드를 제거해야 한다.

매 time step마다 각각의 beam search class로부터 한 sample에 대한 가짜 배치를 받아서(k=3이라 가정) 합치고, 가짜 미니배치를 만들고, decoder에 통과시켜 다시 분할시킨다.

\* Parallelization for single sample beam search:

-k(beam-size)만큼 inference가 반복되고, 병렬 처리를 위해서 k개의 샘플인 것처럼 추론을 수행한다.

\* Mini-batch parallelization for multi sample beam search:

- k\*batch\_size만큼 inference가 반복된다.

- 병렬 처리를 위해서 k\*batch size개의 샘플인 것처럼 추론을 수행한다.

From Board:

Input:

Last hidden state: - LSTM

Last cell state: - LSTM

Last H-tilde: – input feeding

Expand to fake bath size: fake minibatch beam size만큼

Entire output from encoder: , mask

Hidden & cell state와 의 tensor shape의 구성이 다르다

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 도표, 디자인이(가) 표시된 사진

자동 생성된 설명

Beam Search는 위 그림처럼 따로 구현해준다. 이것은 딥러닝적인 구현보다, 프로그래밍 엔지니어링적으로 구현하는 것이다.

- Encoder: Sequence를 하나의 context vector로 압축

- Decoder: Context vector를 condition으로 받는 조건부 언어 모델

- Generator: Decoder의 hidden state를 softmax를 통해 multinoulli분포료 변환

- Attention:

RNN(or LSTM)의 hidden state가 모든 정보를 담기는 어렵다. 따라서 attention을 통해 decoder에서 encoder에 직접 query를 날려 필요한 정보를 access한다. 이때 필요한 정보를 얻기 위한 query를 만들어내는 과정을 배운다.

- Input Feeding:

Auto-regressive특성으로 인한 정보 손실을 보완한다. 는 softmax를 통과하기 이전의 가중치이며, 이것을 입력값의 embedding 값과 concatenation한다. Teacher forcing과 inference 사이의 괴리를 완화한다

- Teacher Forcing:

Auto-regressive task를 학습하기 위함이며, 입력을 지난 time-step의 출력을 받는 것이 아닌, 실제 정답을 받는다

Training과 inference 사이의 괴리가 발생한다. 추후 RL이나 Dual Learning등을 통해 보완한다.