⌨️

Seq2Seq

🛠️ 모델의 구조

기본적인 구조는 Encoder를 통해 입력 문장을 하나의 누적된 Cell State로 변환하는 부분과, 이를 기반으로 Output에 해당하는 Sequence를 생성하는 Decoder 부분으로 나뉜다.

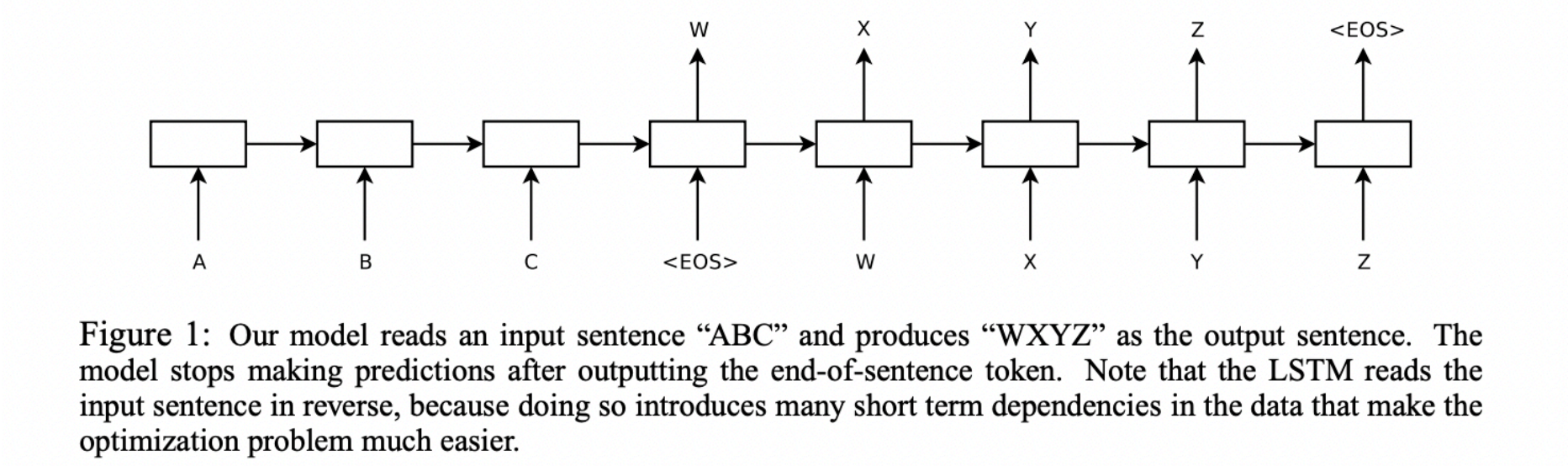


Figure 1: Our model reads an input sentence “ABC” and produces “WXYZ” as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

위 그림을 보면 입력으로 "ABC"가 첫 번째 LSTM에 입력되고, 해당 입력의 결과(번역)인 "WXYZ"가 두 번째 LSTM을 통해 제공된다. 즉, 두 개의 모델이 순차적으로 실행되는 구조다. 그리고 각 문장의 끝은 EOS라는 토큰으로 구분된다.

LSTM을 활용해 가변 길이의 입력을 유연하게 처리할 수 있었다. LSTM의 큰 장점 중 하나는 가변 길이의 입력을 고정 길이의 vector representation으로 변환할 수 있다는 점이다.

Translation을 수행하기 위해 어느 정도의 paraphrasing이 필요하게 되는데, vector representation은 입력 문장의 의미를 재구성(paraphrase)한 형태라고 볼 수 있다. 즉, context vector를 생성하며, 의미가 유사한 문장에 대해 유사한 vector representation을 만들어낸다.

Encoder에서는 입력 문장을 누적된 Cell State로 변환하는 역할을 수행한다. 여기서는 Sequence의 길이에 관계없이 EOS 토큰이 나올 때까지 Input을 계속 모델에 입력한다.

> reference : 이론과 실제 구현 방식이 다를 수도 있다. 실제로는 학습 과정의 편의성과 병렬 처리 효율을 위해 Fixed Length를 설정하는 경우가 많으며, 이 실험에서도 8개의 GPU를 활용하기 위해 문장을 일정한 길이로 분할했다. 따라서 최대 길이를 설정한 후 Padding 또는 Clipping을 적용해야 할 필요가 있었다.

이렇게 생성된 Vector는 Context Vector이자 Latent Vector로 볼 수 있다. Sequence Data를 처리하기 위해 RNN을 사용하려 했지만, RNN은 장기 의존성(Long-term dependency) 문제에 취약했다. (Vanishing Gradient 문제로 인해)

이를 해결하기 위해 LSTM 모델을 사용했고, 보다 긴 기간에 대한 정보를 유지할 수 있어 성능이 향상되었다. LSTM은 입력과 잠재 벡터 H를 고려해 새로운 Vector H를 만들고, 이를 통해 Output을 생성하는 방식이다.

$$
\begin{aligned}
h_t &= \operatorname{sigm}\left(W^{\mathrm{hx}} x_t + W^{\mathrm{hh}} h_{t-1}\right) \\
y_t &= W^{\mathrm{yh}} h_t
\end{aligned}
$$

LSTM 모델의 목표는 입력 문장 $X=(x_1, x_2, \ldots, x_t)$에 대해 출력$(y_1, y_2, y_3)$의 조건부 확률을 최대화하고, 생성된 예측 문장 T'이 Target Sentence T와 최대한 유사하도록 만드는 것이다.

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$$

이때 결과값에 Softmax를 적용하면 확률값으로 변환되며, 이를 통해 조건부 확률을 최대화하게 된다. 모델은 입력 문장 "ABC"를 순차적으로 읽고, 이를 기반으로 WXYZ라는 Output을 생성하는 방식으로 동작한다.

추가적으로 논문 저자는 4층의 Deep한 LSTM 모델을 사용했으며, 문장을 역순으로 입력하도록 설계했다. 이는 실험 결과, 성능이 훨씬 향상되었기 때문이다.

일반적으로 LSTM은 긴 Sequence에 대해 성능이 저하된다는 연구가 있었지만, 해당 논문에서 제안된 모델은 긴 문장에 대해서도 우수한 성능을 보였다.

💡 Decode, Beam Search

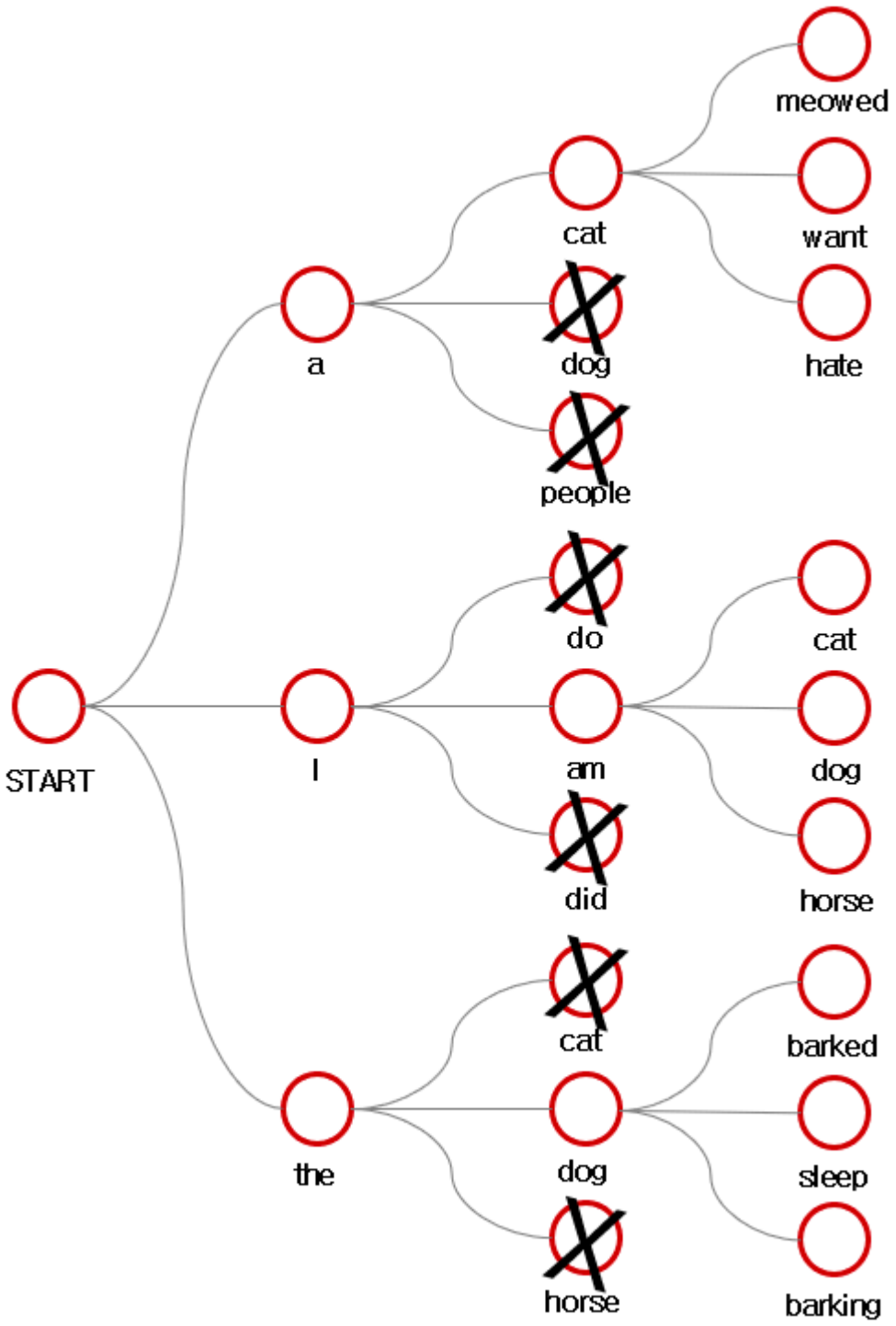
$$1/|\mathcal{S}| \sum_{(T,S)\in\mathcal{S}} \log p(T|S)$$

목적 함수는 Log-Likelihood의 최대화이며, 이는 결과적으로 조건부 확률을 도출하는 과정이다. Training Set 전체에서 이 확률을 최대화하는 것이 목표이며, 이를 통해 일부 문장만 과도하게 맞추는 것이 아니라 평균적인 성능을 높이는 방향으로 학습이 진행되었다.

$$\hat{T} = \arg\max_{T} p(T|S)$$

Decoder에서 Target 문장을 생성할 때 Beam Search 기법이 활용되었다.

> Beam Search 간략한 개요
>
> 1. 가능한 모든 다음 스텝을 확장한다.
> 2. 각 후보에 대해 확률을 곱해 점수를 매긴다.
> 3. 가장 확률이 높은 k개의 시퀀스를 선택하고, 나머지는 제거한다.
> 4. 이 과정을 문장이 끝날 때까지 반복한다.



결과적으로 Decoder에서 출력할 때 Greedy Choice를 보완하는 방식으로 단어 선택이 이루어졌다. 또한, 길이에 대한 패널티를 부여해 짧은 문장보다 긴 문장이 불리하지 않도록 조정되었다.

$$s(Y,X) = \log(P(Y|X))/lp(Y) + cp(X;Y)$$

$$lp(Y) = \frac{(5+|Y|)^\alpha}{(5+1)^\alpha}$$

$$cp(X;Y) = \beta * \sum_{i=1}^{|X|} \log(\min(\sum_{j=1}^{|Y|} p_{i,j}, 1.0)),$$

🔥 문장을 왜 거꾸로 읽는가

흥미롭게도 Encoder가 문장을 거꾸로 읽을 때 Translation 성능이 크게 향상되었다. 논문 저자는 이에 대해 아직 가설 단계라고 하면서도, 초기 Translation Output이 가장 최근에 읽은 데이터에 의해 생성되기 때문일 가능성이 있다고 설명했다.

결과적으로 Decoder가 Beam Search를 수행할 때, 초반 단계에서 결정되는 최대 확률의 단어들이 전체 문장 성능을 좌우하게 된다. 따라서 문장을 역순으로 입력하면 보다 정확한 Latent Vector를 만들 수 있기 때문이라고 설명한다.

일반적으로 문장을 역순으로 입력하면 마지막 부분의 정확도가 낮아질 것 같지만, 실험 결과 오히려 그렇지 않았다고 한다. 오히려 성능이 더욱 향상되었다는 결론을 도출했다.

🔥 GPU 병렬처리

논문에서는 8개의 GPU를 사용해 모델을 학습했다.

Our models have 4 layers of LSTMs, each of which resides on a separate GPU. The remaining 4 GPUs were used to parallelize the softmax, so each GPU was responsible for multiplying by a 1000 × 20000 matrix. The resulting implementation achieved a speed of 6,300 (both English and French) words per second with a minibatch size of 128. Training took about a ten days with this implementation.

즉, 4개의 GPU는 LSTM 연산을 수행하고, 나머지 4개는 Softmax 계산을 병렬로 처리해 연산 속도를 극대화했다. 이러한 방식으로 약 6,300단어/초의 처리 속도를 기록했으며, 128개의 Mini-batch 크기를 사용해 10일간 Training이 진행되었다.