



Hyper-CL 22 Sep 2024

논문-한양대(김태욱교수님) · 2024. 9. 22. 19:29

출처: 한양대 논문 Hyper-CL 22 Sep 2024

Introduction

가중치가 많으면 기억할 수 있는 역량이 늘어남 데이터가 적은데 가중치가 많아지면 overfitting
이 일어나서 외워버리는 상태 모델이 크다면 데이터도 많음

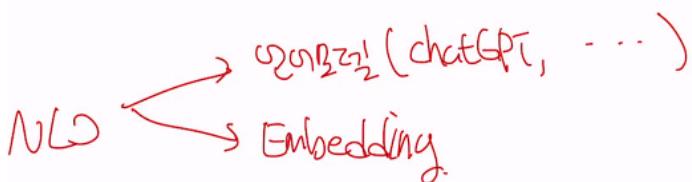
- Kaplan et al., 은 일반적으로 보다 큰 언어 모델이 더 좋은 성능을 가진다는 것을 알아냄
- 따라서 최근 언어 모델들은 계속해서 크기가 커지는 추세
- 이러한 트렌드에서 Embedding 모델 또한 벗어나지 않음
- 문장 Embedding을 만들 때, 보다 큰 모델이 (일반적으로) 더 나은 Embedding을 만드는 것으로 생각됨
- 이렇게 만들어진 Embedding은 일반적으로 유사도 분석 등에 활용됨
- 예를 들어서, 두 문장의 Embedding이 서로 가깝다면, 이 두 문장은 비슷한 뜻을 가지고 있을 것으로 생각됨
- 하지만, 문장의 유사도는 보는 관점에 따라서 달라질 수 있음 (사람과 사람 소통 시)
- 문장 1: A cyclist pedals along a scenic mountain trail, surrounded by lush greenery 산
길에 **자전거**
- 문장 2: A hiker navigates through a dense forest on a winding path, enveloped by the
tranquillity of nature 숲 속에 **걸어가기**

- 관점 1: The mode of transportation

- 관점 2: The speed of travel

아래그림)

언어모델은 비싸서 embedding(두문장의 유사도계산시) 사용



- 이러한 “조건부 유사도”는 단순한 Embedding 모델로는 구현하기가 힘들

- Pande et al., 은 현재 대다수의 문장 Embedding 모델들이 이러한 관점 차이를 제대로 반영하지 못한다는 것을 알아냄

- 이를 해결하기 위해 여러가지 새로운 모델 구조들이 제안됨

- s1, s2 -> 문장 1, 2

- c -> 조건 (관점) = condition

아래그림

문장을 단어화하거나 단어각각 embedding해서 합함.

I eat an apple

- 공통적으로 s1, s2, c를 사용해 두 문장 사이의 유사도를 계산하는 과정

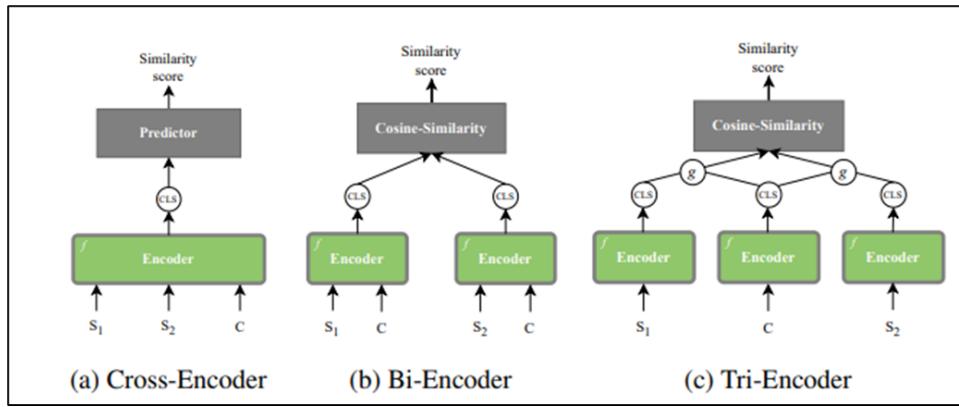
- (기존은 단어를 embedding했지만,

- 요즘은 문장자체를 embedding)

- 문장전체를 단어화하거나

- 문장을 나눔

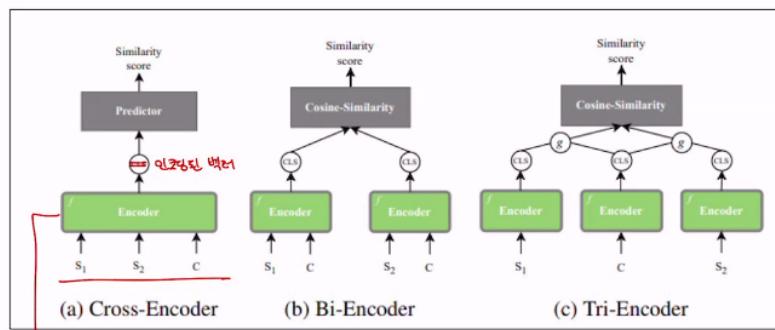
- 입력값들이 인코딩 되고 (CLS) 이후 여러 과정을 거쳐 유사도 계산



아래그림

세문장을 한번에 한줄에 만듬-> encoder(문장 embedding 모델- 문장을 embedding만드는 모델)

-> predictor(예측)하게 만듬 -> 유사도



→ 문장 인코딩 모델
 $S_1 + S_2 + C$
I eat an apple I eat a grape sweetness

아래그림)

입력받는 벡터의 크기는 동일(제한됨)

1. 똑같은 embedding으로 임의로 만듬(입력값 변경)

-> 일반적으로는 너무 긴거를 받거나 그이상은 자르거나

부족한입력값은 padding으로 채움-> 똑같은 embedding을 만든것

2. embedding 통과해서 고정된 크기의 vector로 바꿔준다.

입력받는 벡터의 크기는 동일(제한됨)

어떤 글(들) \rightarrow [] \rightarrow 인식

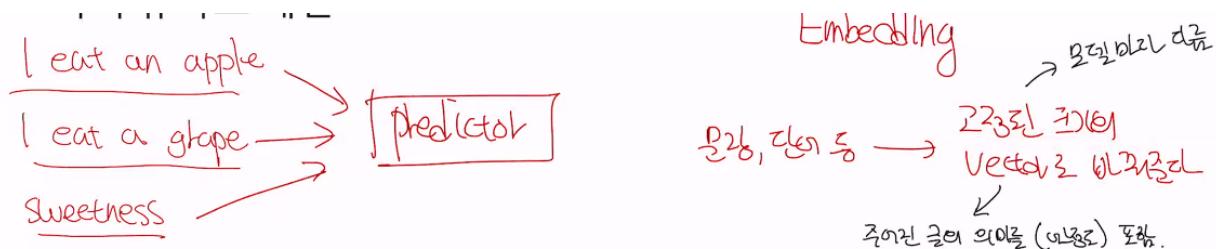
아래그림

Embedding

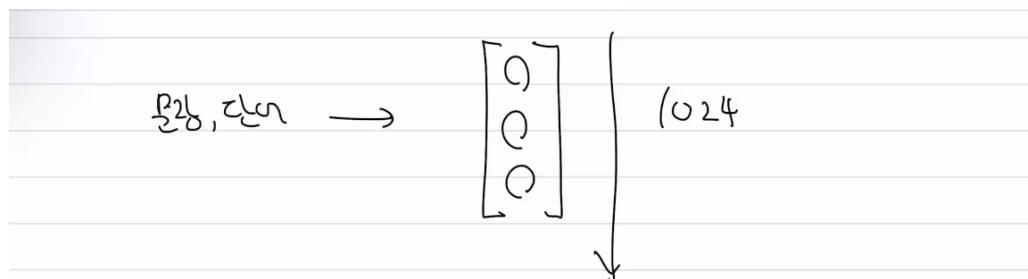
문장, 단어 등 \rightarrow 고정된 크기의 vector로 바꿔준다.

vector \rightarrow 주어진 글의 의미를(어느정도) 포함

아래그림



모델의 embedding 크기 고정 - 만드는 사람이 결정



다시정리

목적:

임의의 문장, 조건 등을 동일한 크기의 vector로 변환

1. 긴거 자르고, 남는거 메꾸고 (중요한게 뒤에 오면 버려짐)
2. Embedding 모델을 쓰는거(아주많은 학습한 모델) (여러경우를 학습한 모델이라 보다 유연하게 대처)

아래그림

• Cross-Encoder / Bi-Encoder

• 공통점

• 주어진 문장과 조건을 같이 인코딩함

• 문장들과 조건의 상호작용을 직접적으로 학습할 수 있음

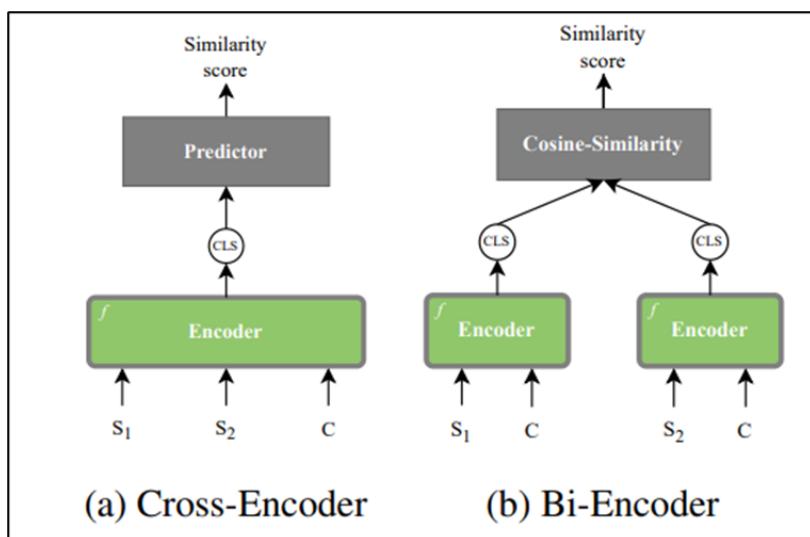
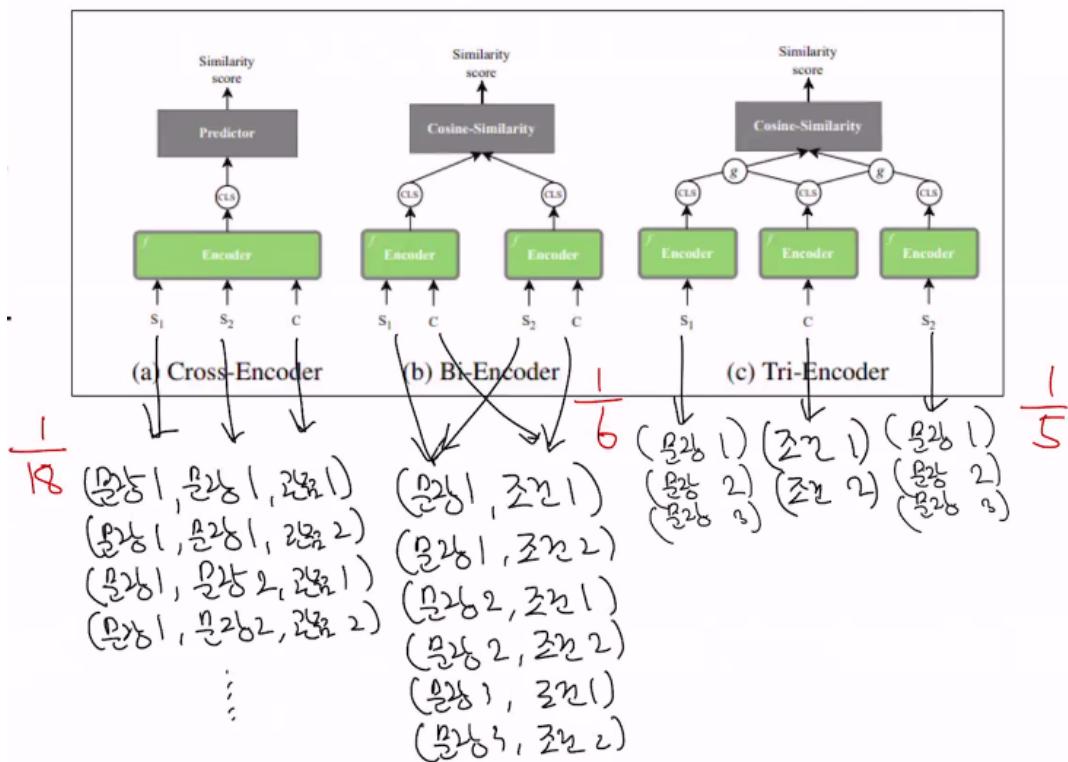
(보다 높은 성능 기대 가능)

• 차이점: cross encoder와 bi encoder 차이점

• bi encode는 문장간의 상호작용은 학습이 힘듬.

모든 경우의 수를 전부 계산해야 한다는 단점이 있음

따라서 훈련에 훨씬 많은 데이터가 필요함



아래그림)

또한 이미 계산된 값 (CLS 등)을 재사용하기가 어려움

아래그림

• 그에 반해 Tri-Encoder는 각 문장과 조건을 따로 인코딩함

• 이후 각 문장과 조건을 합친 다음 (함수 g 를 통해) 유사도 계산

• 이때 계산된 값들 (CLS 등)은 이후 재사용 가능 -> 훨씬 높은 효율

• 다만 Bi-Encoder에 비해 낮은 성능을 보이는데, 이는 문장과 조건 사이의 관계를 직접적으로 학습하지 못하기 때문

연구목적

cross encoder 는 너무 경우의 수가 많아서 학습하기 힘들어서 제외하고,

Bi-Encoder 와 Tri-Encoder 를 활용해서

• Bi-Encoder의 성능은 유지하면서 Tri-Encoder의 효율을 유지하는 방법이 필요함

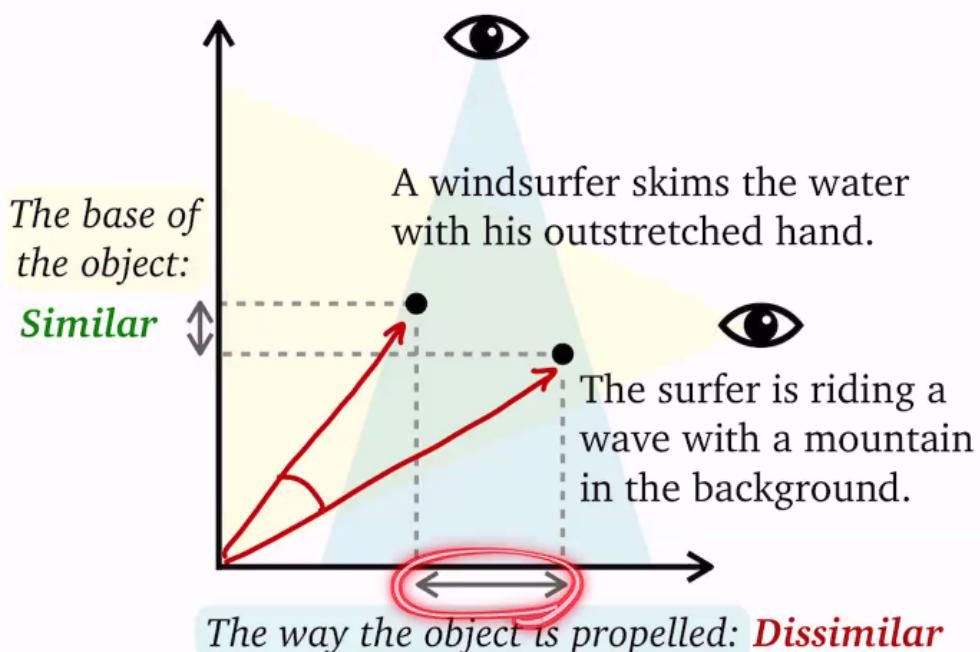
• 본 연구에서는, Tri-Encoder 구조를 개선해 Bi-Encoder 급의 성능을 낼 수 있는 방법을 제안 함

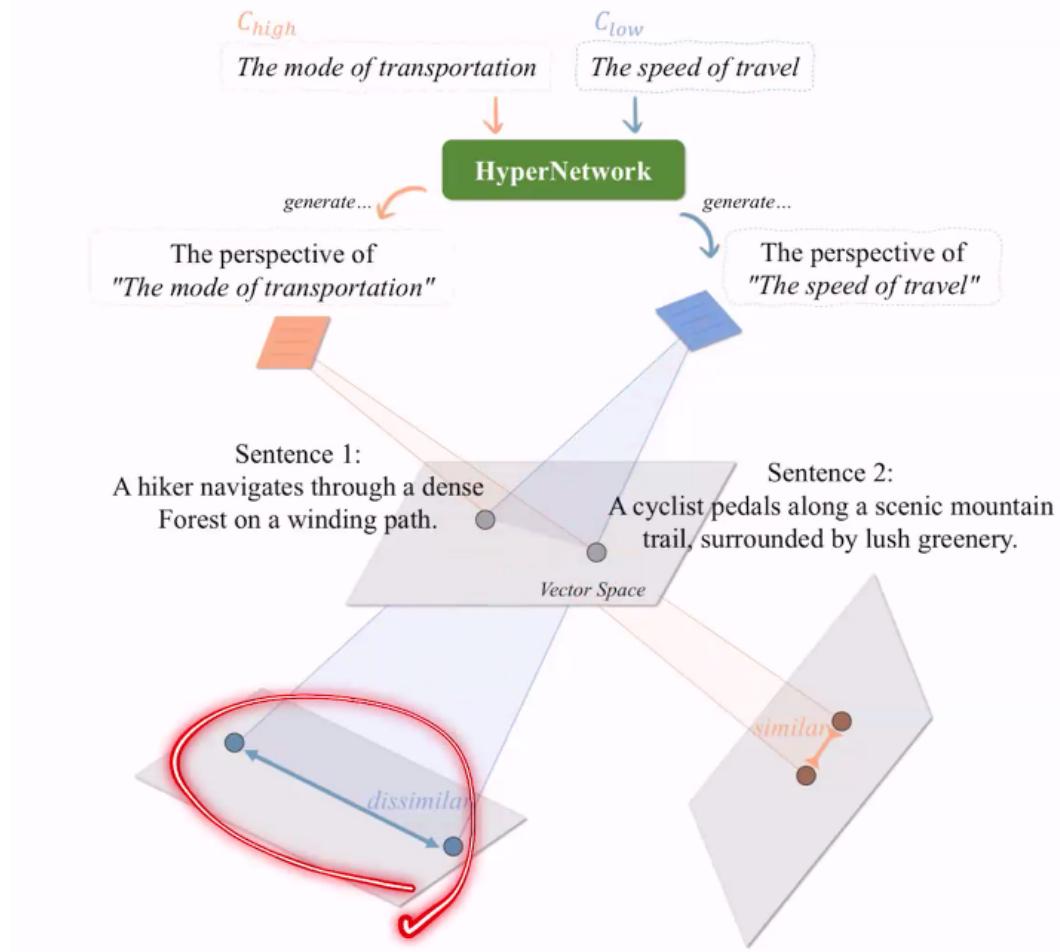
관점에 따라 embedding 사이의 유사도 $\cos\theta$ 가 달라짐

노랑색일 때랑, 파랑색일 때랑 유사도가 달라짐

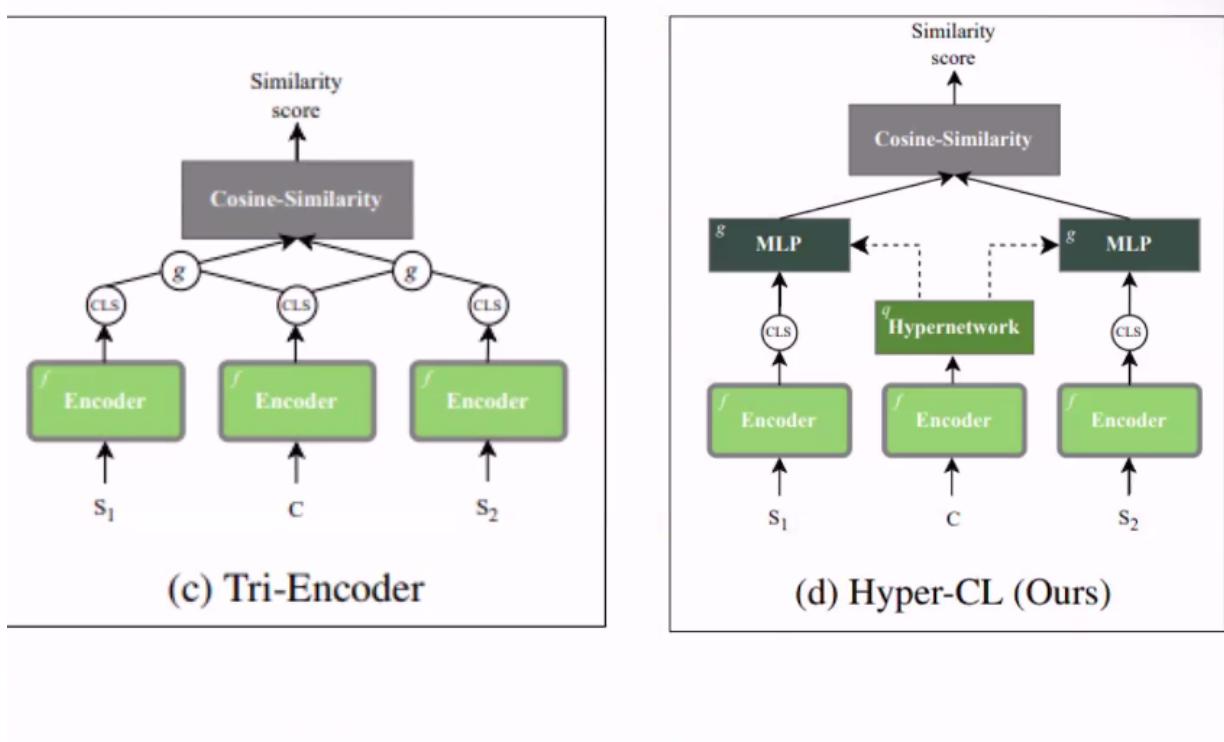
아래그림

출처: C-STS





아래그림)

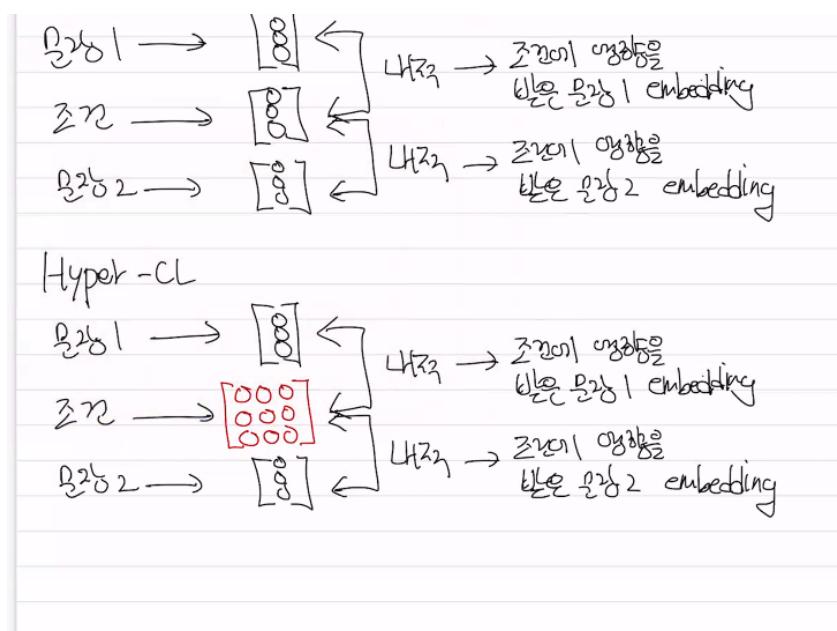


아래그림)

triencoder의 정확도를 높이기 위해

hypernetwork를 통해 주어진 조건에 대해 행렬을 만든후

문장1과 문장2에 곱했다.



Dataset(하나의 dataset)

- Hyper-CL은 두 가지 데이터셋 (Task)을 사용해 각각 훈련되고 평가되었음

• 1. Conditional Semantic Textual Similarity (C-STS)

- 각 Entry는 문장 1, 문장 2, c_low, c_high 의 네 가지 요소로 구성됨

• 문장 1과 문장 2는 c_high 의 조건에서는 매우 가깝고(높은 유사도), 반대로 c_low(낮은 유사도)의 관점에서는 멀리 떨어져 있음

• 이때 모델은 주어진 문장과 조건에 맞는 유사도를 계산해야 함

아래그림

예시)

S_1 = 숲에서 자전거를 탄다, S_2 = 숲에서 걷는다.

C_{high} = 주번 풍경

C_{low} = 이동수단.

$(S_1, C_{high}, S_2) \rightarrow$ 모델 \rightarrow 높은 유사도

$(S_2, C_{low}, S_2) \rightarrow$ 모델 \rightarrow 높은 유사도

$(S_1 = \text{숲에서 자전거를 탄다}, S_2 = \text{숲에서 걷는다}, C_{high} = \text{주번 풍경}, C_{low} = \text{이동수단})$

$(S_1, C_{high}, S_2) \rightarrow$ [모델] \rightarrow 높은 유사도

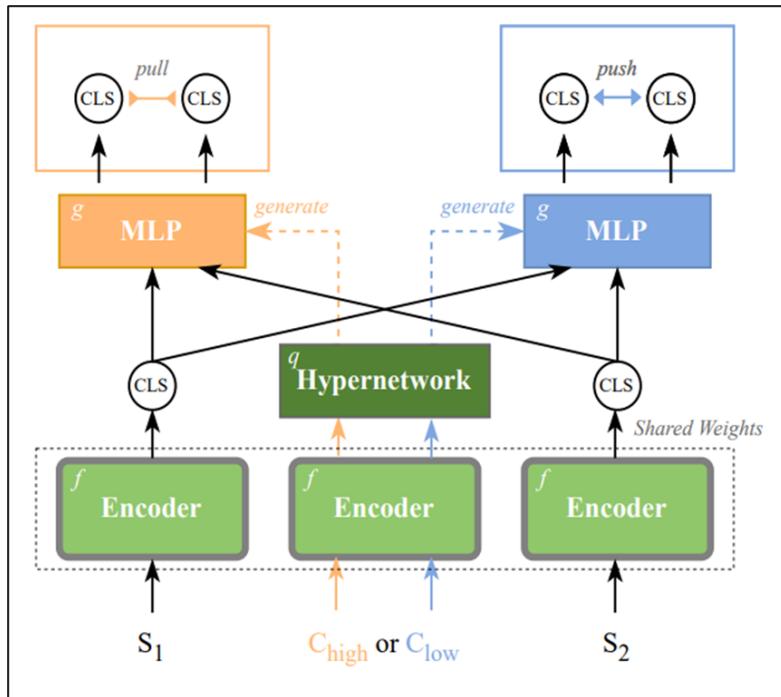
$(S_1, C_{low}, S_2) \rightarrow$ [모델] \rightarrow 낮은 유사도

아래그림

Experiment

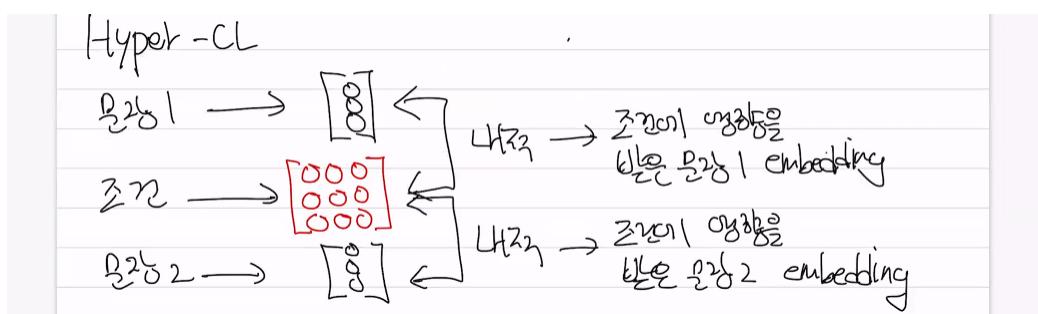
- Hyper-CL은 Contrastive Learning을 통해 훈련됨

- 간단히 말하자면, 가까운 것들은 더욱 가깝게, 먼 것들은 더욱 멀도록 하는 것



- 먼저 C-STS에서는
 - 두개의 문장들은 c_{high} 가 주어졌을 때는 높은 유사도를 가져야 함
 - 반대로, c_{low} 가 주어졌을 때는 낮은 유사도를 가져짐
 - 이를 바탕으로 Hyper-CL 훈련 진행 (역전파 등)
 - 실제 Error 계산 공식 등의 세부적인 내용은 생략됨

Experiment (C-STS)



아래그림

단점:

triencoder에서 조건을 hypernetwork를 적용해서 행렬을 만든 후

기존 문장 embedding에 내적해서

조건이 반영된 새로운 문장 embedding으로 하는데,

만약, 기존 문장 embedding이 엄청 큰 숫자가 되면,
곱해지는 조건을 hypernetwork를 적용해서 행렬도 똑같이 커짐
그리면 hypernetwork도 커져야해서 (모델사이즈가 커짐)
비효율성(돈이 많이 듬)

hypernetwork

$$\begin{bmatrix} \text{입력} \\ \text{입력} \end{bmatrix} = \begin{bmatrix} \text{출력} \end{bmatrix}$$

$$\begin{array}{ccc} 3 \times 3 & 3 \times 1 & = 3 \times 1 \\ \downarrow & \searrow & \downarrow \\ 10000 \times 10000 & 10000 \times 1 & = 10000 \times 1 \end{array}$$

• 이걸 만들어야 됨 (너무 비효율)

$$\begin{bmatrix} 10000 \times 64 \end{bmatrix} \begin{bmatrix} 64 \times 10000 \end{bmatrix} = 10000 \times 10000$$

- 이를 해결하기 위해 상대적으로 작은 Hypernetwork 두개를 만들고, 이 둘이 훨씬 작은 행렬 두개를 만들도록 함

- 이 Hypernetwork들은 $h \times k$ 행렬을 만드는데, 이때 k 는 h 보다 훨씬 작은 값임

- 만약 h 가 10, k 가 3라면, 원본 Hypernetwork는 10×10 의 행렬을 만들었지만, 작은 모델들은 10×3 를 만드는 것

- 이 두개의 작은 행렬들을 곱하면 (둘중 하나를 전치해서)

- $(10 \times 3 \text{ 행렬}) \times (3 \times 10 \text{ 행렬}) \rightarrow 10 \times 10 \text{ 행렬}$

- 이러한 Hyper-CL은 다양한 모델에 사용될 수 있음

- 연구진들은 다양한 Tri-Encoder 모델들에 Hyper-CL을 적용하고 차이점을 분석하였음

- 적용된 모델에는 Hyper-CL이 적용 있으며, 만약 두개의 Hypernetwork로 나뉘었다면 h 의 크기가 같이 적용 있음

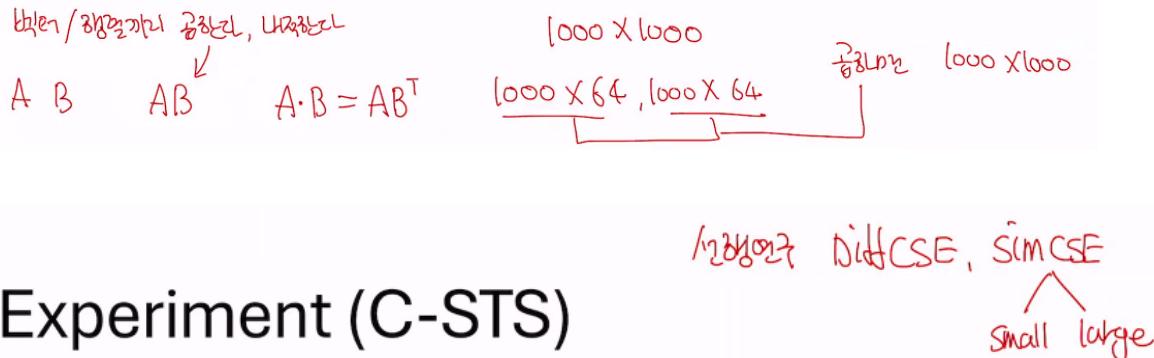
아래그림

다른 연구에서 모델 3개 DiffCSE SimCSE(small, large) 를 가져와서 직접실험비교함.

tri-encoder hyper cl를 적용했고, 이는 너무 커져서 비효율적이어서

작은행렬 2개를 만들어서 곱했을때 원래행렬이 나오도록 한것

예) $1000 \times 1000 \rightarrow 1000 \times 64, 64 \times 1000 \rightarrow$ 두개를 곱함



Experiment (C-STS)

Method	# Params	Spearman	Pearson
<i>tri-encoder architectures</i>			
DiffCSE [†] _{base}	125M	28.9 _{0.8}	27.8 _{1.2}
*DiffCSE _{base+hyper64-cl}	200M	33.10 _{0.2}	31.68 _{0.6}
*DiffCSE _{base+hyper-cl}	578M	33.82 _{0.1}	33.10 _{0.3}
<i>bi-encoder</i>			
SimCSE [†] _{base}	125M	31.5 _{0.5}	31.0 _{0.5}
*SimCSE _{base+hyper64-cl}	200M	38.36 _{0.1}	37.53 _{0.04}
*SimCSE _{base+hyper-cl}	578M	38.75 _{0.3}	38.38 _{0.3}
<i>bi-encoder architectures</i>			
DiffCSE [†] _{base}	125M	43.4 _{0.2}	43.5 _{0.2}
SimCSE [†] _{base}	125M	44.8 _{0.3}	44.9 _{0.3}
SimCSE [†] _{large}	355M	47.5 _{0.1}	47.6 _{0.1}

Table 1: Performance on C-STS measured by Spearman and Pearson correlation coefficients. The best results are in **bold** for each section. *: indicates the results of Hyper-CL. †: denotes results from Deshpande et al. (2023).

아래그림

임의의 두값사이 비례관계 - 맨왼쪽 정비례해서 값=1

$$\text{역비례} - \quad \text{값} = -1$$

$$\text{중간 점들 비례관계 없으면} \quad \text{값} = 0$$

아래그림

- 결과 표를 보면, Hyper-CL을 적용했을 때 Tri-encoder의 점수는 평균적으로 7.25가 올랐음

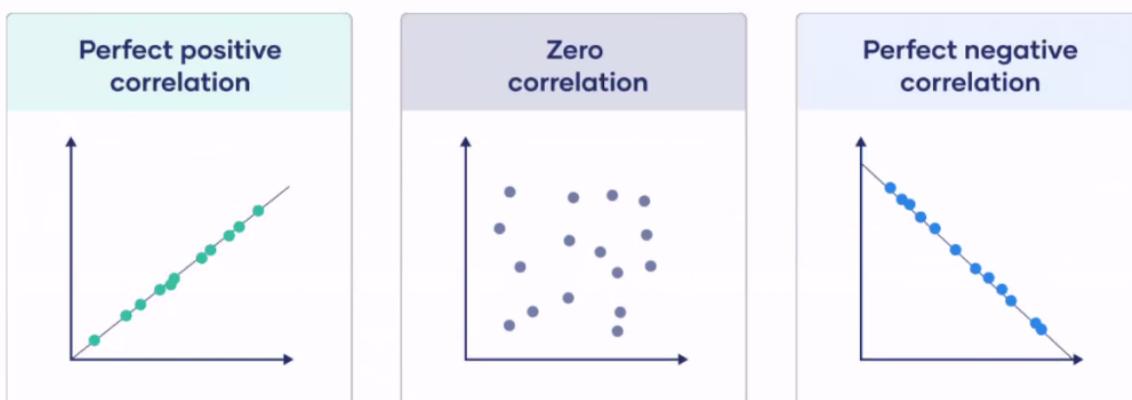
- 원래는 Bi-Encoder와 Tri-Encoder 사이에 13.3점의 차이가 있었지만, Hyper-CL을 적용했을 때는 이 차이가 6.05로 줄어듬

- 또한 이 결과는 두개의 작은 Hypernetwork를 사용했을 때도 거의 변하지 않음 (효율성 증가)

[Tri-encoder] vs [Bi-encoder]
[Tri-encoder + hyper_CL] vs [Bi-encoder]

보다 적은 메모리와 시간을 사용해 (거의) 같은 결과를 낼 수 있음

Experiment (C-STS)



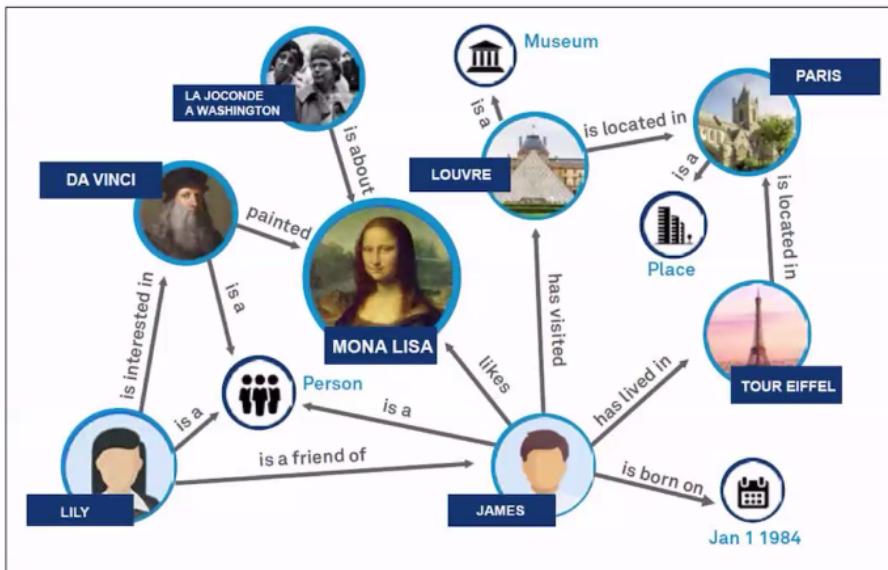
Scribbr

두번째 task (Dataset)

2. Knowledge Graph Completion

- Knowledge Graph는 단어 그대로, 지식 (정보)를 그래프 형태로 나타낸 것

- 각 Entity (동물, 식물, 건물 등등) 사이의 관계들을 포함함



•Knowledge Graph에서, 어떠한 지식은 세가지 요소로 표현됨

•Head, Relation, Tail (때때로 다른 용어가 사용되기도 함)

•예를 들자면,

•Da Vinci (Head)

•Painted (Relation)

•Mona Lisa (Tail)

•이때 모든 지식을 담는 것은 불가능하므로, 기존에 알고 있는 내용으로부터 새로운 지식을 추론할 필요성이 있음

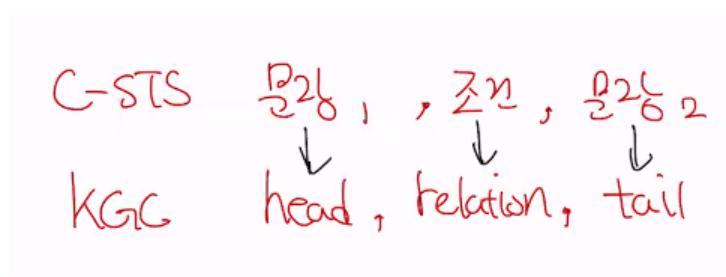
•Head와 Relation, 또는 Relation과 Tail이 주어졌을 때 가장 적절한 요소를 예측하는 것 (다른 예측도 가능)

•예시:

•____, “is the capital city of”, “France”

- “Seoul”, “is the capital city of”, _____

아래그림



아래그림

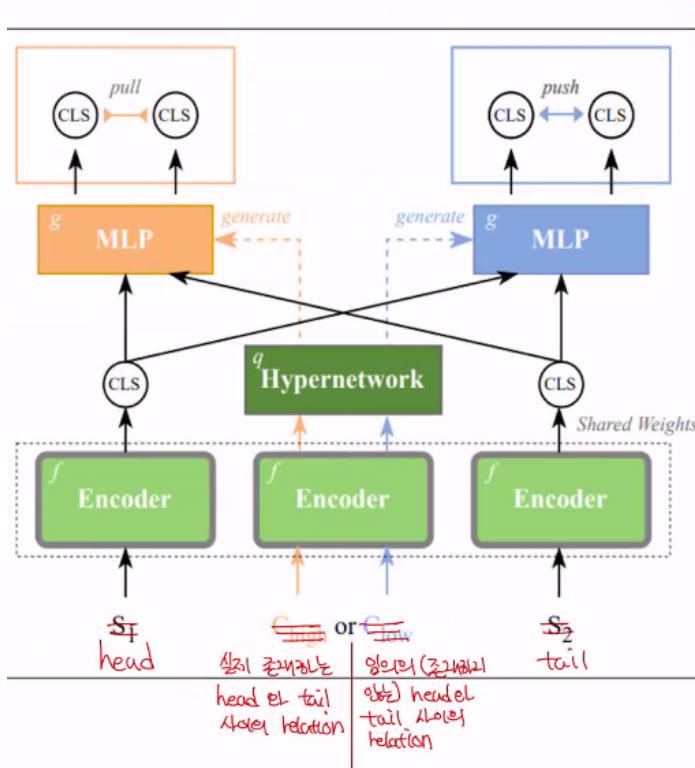
- Hyper-CL은 Contrastive Learning을 통해 훈련됨

간단히 말하자면, 가까운 것들은 더욱 가깝게, 먼 것들은 더욱 멀도록 하는 것

실제 존재하는 head와 tail 사이의 relation -> Chigh와 같은 의미

실제 존재하지 않는 head와 tail 사이의 relation -> Clow와 같은 의미

	head	relation	tail	
Chigh	다른나라	그렇다	모나리자	실제 존재
Clow	개인정보	그렇다	모나리자	임의로 만들것

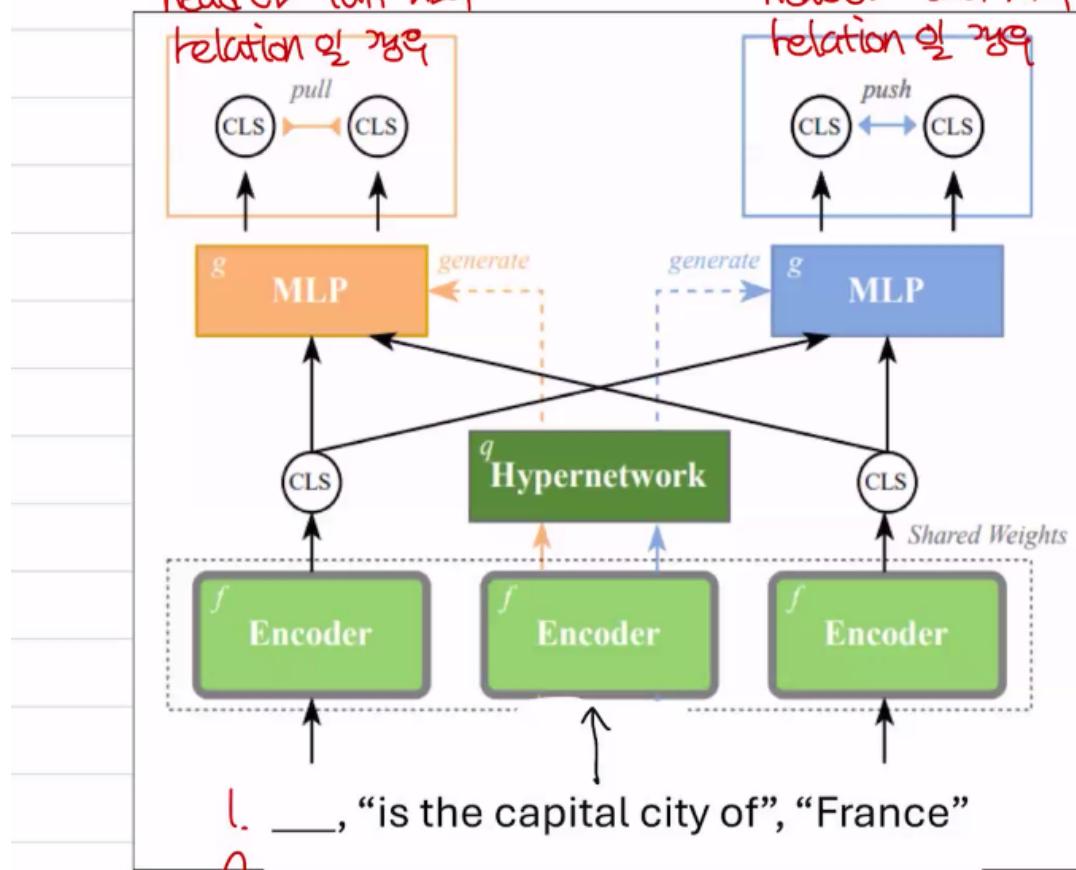


아래그림)

head나 tail를 추측해서 유사도가 높은것을 정답으로 함.

실제 존재하는
header tail 쌍

존재하지 않는
header tail 쌍



1. ___, “is the capital city of”, “France”

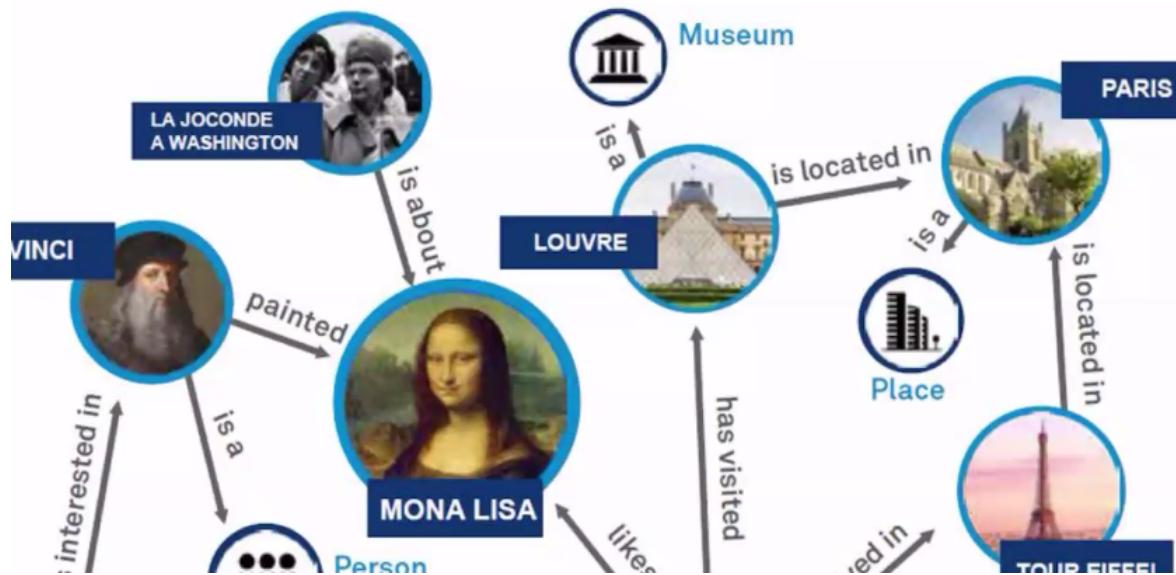
2. “Seoul”, “is the capital city of”, __

head relation tail

작은 것 그 러 르 L 끝

길리 는 시 갈 능 이 끝

↓



- 비슷하게, Knowledge Graph 예측에 대해서도 실험을 진행함

- 총 두개의 데이터셋 (WN18RR, FB15K-237) 을 사용하였음 → 두개의 knowledge graph
- 실제 존재하는 Head, Relation, Tail이 주어졌을 때는 높은 유사도를 예측하도록 모델을 훈련시키고
- 반대로 랜덤하게 만들어낸 Head, Relation, Tail은 낮은 유사도를 가지도록 훈련

is the capital city of France

→ Paris 100
Seoul 50
James 10

entity ↗

아래그림

예측순위 숫자가 낮을수록 더 좋은 모델임.

단순 순위 평균을 하게되면 예측순위 숫자가

-> 나쁜모델 -> 더 높은 점수

-> 더 좋은 모델 -> 이 낮은 점수

를 받게되어서 직관적이지 않아서

역수취함

모델 1	3, 2, 5 <small>(실제 정답)</small>	$\rightarrow (3+2+5) \frac{1}{3} = 3.3$ <small>역수</small>	$\rightarrow \frac{1}{3.3} \approx 0.303$
모델 2	30, 40, 50 <small>(나온 순위)</small>	$\rightarrow (30+40+50) \frac{1}{3} = 40$ <small>역수</small>	$\rightarrow \frac{1}{40} \approx 0.025$

아래그림)

• Hits@n

• 실제 정답이 n 순위 안에 등장한 비율

• Hits@5

데이터셋에서, 실제 정답이 5위 안에 등장한 비율

• 예시 (데이터셋에 10개의 Entry가 있을때)

• 만약 6개의 Entry에서 실제 정답이 10 순위 안에 등장했으면

• Hits@10 = 0.6

	질문1	질문2	질문3
Hits@3	예측1	예측1	예측1
= $\frac{2}{3} = 66\%$	예측2	예측2	예측2
	예측3	예측3	예측3
	예측4	:	:

아래그림)

- Hyper-CL 을 적용한 SimKGC는 원본과 (원래는 Bi-Encoder) 비교했을때 큰 성능 차이가 나지 않음

- 또한, 두개의 Hypernetwork로 나눴을 때에도 큰 성능 저하가 일어나지 않았음

SimKGC 는 선행연구에서 갖고온 bi-encoder 예측모델이고,
아래그림) SimKGC 를 tri-encoder 로 개조했는데, 성능이 잘 안나옴

tri-encoder architectures

SimKGC _{hadamard-product}	0.164	0.004	0.243	0.481	0.153	0.092	0.162	0.274
SimKGC _{concatenation}	0.335	0.226	0.382	0.550	0.271	0.193	0.292	0.430

아래그림) Tri-encoder (SimKGC) hyper CL 적용하니, bi-encoder(SimKGC)만큼 성능이 나옴

bi-encoder architectures

StAR [†]	0.401	0.243	0.491	0.709	0.296	0.205	0.322	0.482
SimKGC [‡]	0.666	0.587	0.717	<u>0.800</u>	0.336	<u>0.249</u>	<u>0.362</u>	0.511

tri-encoder architectures

SimKGC _{hadamard-product}	0.164	0.004	0.243	0.481	0.153	0.092	0.162	0.274
SimKGC _{concatenation}	0.335	0.226	0.382	0.550	0.271	0.193	0.292	0.430
*SimKGC _{hyper-cl}	0.616	0.506	<u>0.690</u>	0.810	<u>0.318</u>	0.231	0.344	0.496
*SimKGC _{hyper64-cl}	0.548	0.427	<u>0.626</u>	0.770	<u>0.305</u>	0.219	<u>0.331</u>	0.479

아래그림) 하지만, Tri-encoder (SimKGC) hyper CL 를 적용하면 hypernetwork를 통해 행렬 만들어서

기존 embedding에 곱한다.

이 과정에 더 많은 시간, 메모리등이 소요된다.(모델도 커지고, 비효율적)

Method	# Params	Spearman	Pearson
<i>tri-encoder architectures</i>		모델의 크기(모델의 가중치가 커짐)	
DiffCSE [†] _{base}	125M	28.9 _{0.8}	27.8 _{1.2}
*DiffCSE _{base+hyper64-cl}	200M	33.10 _{0.2}	31.68 _{0.6}
*DiffCSE _{base+hyper-cl}	578M	33.82 _{0.1}	33.10 _{0.3}
<i>tri-encoder</i>			
SimCSE [†] _{base}	125M	31.5 _{0.5}	31.0 _{0.5}
*SimCSE _{base+hyper64-cl}	200M	38.36 _{0.1}	37.53 _{0.04}
*SimCSE _{base+hyper-cl}	578M	38.75 _{0.3}	38.38 _{0.3}
SimCSE [†] _{large}		355M	35.3 _{1.0}
*SimCSE _{large+hyper85-cl}	534M	38.12 _{1.4}	37.47 _{1.4}
*SimCSE _{large+hyper-cl}	1431M	39.60 _{0.2}	39.96 _{0.3}
<i>bi-encoder architectures</i>			
DiffCSE [†] _{base}	125M	43.4 _{0.2}	43.5 _{0.2}
SimCSE [†] _{base}	125M	44.8 _{0.3}	44.9 _{0.3}
SimCSE [†] _{large}	355M	47.5 _{0.1}	47.6 _{0.1}
<i>bi-encoder</i>			

Tri-encoder (SimKGC) hyper64cl은 원래 Tri-encoder (SimKGC) **hyperCL을 두개로 쪼개**서, 적용하면 성능을 약간 희생하고,
효율을 훨씬 높일수 있다.

<i>bi-encoder architectures</i>								
StAR [†]	0.401	0.243	0.491	0.709	0.296	0.205	0.322	0.482
SimKGC [†]	0.666	0.587	0.717	<u>0.800</u>	0.336	<u>0.249</u>	<u>0.362</u>	0.511
<i>tri-encoder architectures</i>								
SimKGC _{hadamard-product}	0.164	0.004	0.243	0.481	0.153	0.092	0.162	0.274
SimKGC _{concatenation}	0.335	0.226	0.382	0.550	0.271	0.193	0.292	0.430
*SimKGC _{hyper-cl}	<u>0.616</u>	0.506	<u>0.690</u>	0.810	<u>0.318</u>	0.231	0.344	0.496
*SimKGC _{hyper64-cl}	0.548	0.427	0.626	0.770	0.305	0.219	0.331	0.479

Method	WN18RR				FB15K-237			
	MRR	Hits@1	Hits@3	Hits@10	MRR	Hits@1	Hits@3	Hits@10
<i>cross-encoder architectures</i>								
KG-BERT [†]	0.216	0.041	0.302	0.524	-	-	-	0.420
MTL-KGC [†]	0.331	0.203	0.383	0.597	0.267	0.172	0.298	0.458
<i>encoder-decoder architectures</i>								
GenKGC [†]	-	0.287	0.403	0.535	-	0.192	0.355	0.439
KG-S2S [†]	0.574	<u>0.531</u>	0.595	0.661	0.336	0.257	0.373	<u>0.498</u>
<i>bi-encoder architectures</i>								
StAR [†]	0.401	0.243	0.491	0.709	0.296	0.205	0.322	0.482
SimKGC [†]	<u>0.666</u>	0.587	0.717	<u>0.800</u>	0.336	<u>0.249</u>	<u>0.362</u>	0.511
<i>tri-encoder architectures</i>								
SimKGC _{hadamard-product}	0.164	0.004	0.243	0.481	0.153	0.092	0.162	0.274
SimKGC _{concatenation}	0.335	0.226	0.382	0.550	0.271	0.193	0.292	0.430
*SimKGC _{hyper-cl}	<u>0.616</u>	0.506	<u>0.690</u>	0.810	0.318	0.231	0.344	0.496
*SimKGC _{hyper64-cl}	0.548	0.427	0.626	0.770	0.305	0.219	0.331	0.479

Table 2: Results on the WN18RR and FB15K-237 datasets for KGC, measured by MRR and Hits@K. The best results are highlighted in **bold**, while the next best results are underlined for each column. *: indicates the results of applying Hyper-CL. †: denotes results from Chen et al. (2022). ‡: denotes results from Wang et al. (2022). Other results are implemented and evaluated by the authors.

- 추가적으로, C-STS와 KGC 둘에 대한 추론 시간 (실행 시간), Cache Hit rate, Cache(재활용, FAQ) 사이즈에 대한 분석을 진행함

아래그림)

- 공통적으로, Bi-Encoder 모델보다 Tri-Encoder 모델이 훨씬 낮은 실행 시간을 보여줌
- 또한 더 높은 Cache-Hit rate를 가짐 (이전에 계산했던 값을 효율적으로 재사용함)

Experiment (C-STS and KGC)

Method	Time	HitRate	Cache
<i>bi-encoder architectures</i>			
SimCSE _{base}	791.71s	1.46%	110.87MB
SimCSE _{large}	1498.65s	1.46%	147.26MB
<i>tri-encoder architectures</i>			
SimCSE _{base}	441.17s	64.11%	60.57MB
SimCSE _{base+hyper64-cl}	525.62s	64.11%	2.17GB
SimCSE _{base+hyper-cl}	541.55s	64.11%	12.81GB
SimCSE _{large}	832.19s	64.11%	80.45MB
SimCSE _{large+hyper64-cl}	990.94s	64.11%	3.82GB
SimCSE _{large+hyper-cl}	960.84s	64.11%	22.75GB

Table 3: Analysis of inference time, cache hit rate, and memory usage for different architectures and methods on the entire C-STS dataset.

Method	Time	HitRate	Cache
<i>bi-encoder architectures</i>			
SimKGC _{base}	994.41s	46.65%	295.29MB
SimKGC _{large}	1806.18s	46.65%	392.2MB
<i>tri-encoder architectures</i>			
SimKGC _{base+hadamard}	435.571s	85.32%	121.86MB
SimKGC _{base+concatenation}	449.46s	85.32%	121.86MB
SimKGC _{base+hyper-cl}	448.955s	85.32%	146.57MB
SimKGC _{large+hadamard}	781.45s	85.32%	161.85MB
SimKGC _{large+concatenation}	783.228s	85.32%	161.85MB
SimKGC _{large+hyper-cl}	774.41s	85.32%	205.81MB

Table 4: Analysis of inference time, cache hit rate, and memory usage for different architectures and methods on the entire WN18RR dataset.

- 마지막으로, Hyper-CL이 학습중 보지 못했던 조건 (관점)(새로운조건)에 대해 얼마나 좋은 성능을 보여주는지 마지막 평가 진행

- C-STS 데이터셋 평가에서, Validation 셋을 학습에 포함된 조건들과 포함되지 않았던 조건들로 나눔 (기존에는 합쳐서 진행)

- 대략 26%의 Validation 셋이 학습에 포함되지 않았던 조건들이

- 이후 학습된 모델 (Hyper-CL 적용 / 미적용)을 두가지 Validation 셋에 각각 테스트

아래그림)

- 전반적으로, Hyper-CL을 적용했을때 명확한 성능 향상이 이루어지는것을 확인할 수 있음

- 특히 학습때 보여주지 않았던 조건들에 대한 성능 향상이 두드러짐

Method (Metric: Spearman)	Overall	Unseen	Seen
SimCSE _{large}	32.13	13.93	25.02
SimCSE _{large+hyper-cl}	38.59	36.25	41.14

Table 5: Generalization capabilities of Hyper-CL on the C-STS validation set. We compare the *tri-encoder* baseline and Hyper-CL in both ‘unseen’ and ‘seen’ settings, using Spearman’s correlation as the evaluation metric.

Conclusion

- Embedding을 통해 문장간 유사도를 계산할 수 있음
- 다만 일반적인 Embedding 모델은 보는 관점에 따른 유사도 차이를 잘 고려하지 못함
- 이를 해결하기 위해 여러 모델들이 제안되었음
- 이때 Bi-Encoder는 Tri-Encoder에 비해 상대적으로 성능이 높지만 효율이 떨어짐
- 이를 해결하기 위해 본 연구는 Hyper-CL을 제안하였고, 이를 적용한 Tri-Encoder는 효율을 유지하며 Bi-Encoder의 성능을 상당히 따라잡는데 성공함

Limitations

- 본 연구에서는 (Embedding) Encoder에만 집중하였으며, Decoder에는 관심을 주지 않았음
- 또한 보다 다양한 모델과 데이터셋을 통한 후속 연구 필요



구독하기

자연어(NLP)

네이처2024 님의 블로그입니다.



구독하기 +

댓글 7



익명

비밀댓글입니다.

2024. 9. 22. 23:19



트래블러7

귀중한 정보 감사합니다! 👍 저도 비슷한 주제로 포스팅하고 있으니 방문해주시면 좋겠어요!

2024. 9. 23. 00:51 · 답글



저스트맥스

명쾌한 설명 감사해요! 🙌 제 블로그도 방문해주시면 좋겠어요.

2024. 9. 23. 06:36 · 답글



익명

비밀댓글입니다.

2024. 10. 10. 08:46



익명

비밀댓글입니다.

2024. 10. 10. 08:46



익명

비밀댓글입니다.

2024. 10. 10. 08:51



익명

비밀댓글입니다.

2024. 10. 10. 08:51



이름

비밀번호

내용을 입력하세요.



등록