

# [Basic NLP] Sentence-transformers 라이브러리를 활용한 SBERT 학습 방법

jaehyeong.an · 2022년 3월 1일

팔로우

BERT

Sentence Embedding

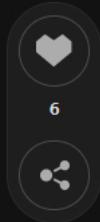
finetuning

sbert

sentencebert

siames-network

triplet-loss



Basic NLP



▼ 목록 보기

5/6



## Intro

이번 포스트에서는 이전 [포스트](#)에서 소개한 SentenceBERT모델의 fine-tuning 방법에 대한 글이다.

우선 기본적으로 SentenceBERT를 학습하기 위한 데이터셋(STS, NLI)에 대해 간단히 소개하고,

STS 단일 데이터를 통한 finetuning 그리고 NLI로 학습된 모델에 STS를 추가 학습시키는 continue learning에 대한 학습 방법 및 결과를 논문 및 sentence-transformers 공식 깃헙을 기준으로 소개하고자 한다.

본 포스트에서 사용된 Colab 실습 코드

- SBERT-only-STS-training
- SBERT-continue-learning-by-Softmaxloss-NLI
- SBERT-continue-learning-by- MNRloss-NLI

## 1. SBERT 학습 데이터

SBERT 학습을 위해 사용되는 데이터로는 문장 사이의 유사도를 측정하는 STS 데이터셋과, 문장 사이의 관계를 파악하는 NLI 데이터셋이 존재한다.

이 두 데이터셋의 경우 활용 가능한 출처는 많지 않지만, 현재 공신력 있게 사용되는 데이터셋은 카카오 브레인에서 공개한 KorNLU와 KLUE 프로젝트에서 공개한 KLUE 벤치마크셋 2가지이다. 여러 공개된 언어모델을 보면 주로 이 데이터셋들을 통해 모델의 성능을 평가하고 있다.

### 1.1. STS (Semantic Textual Similarity)

STS 데이터는 두 개의 문장 쌍과 이 두 문장 사이의 유사도 점수로 구성되어 있으며, 이를 학습하여 문장과 문장간 서로 얼마나 유사한지를 예측하게 된다. 아래는 KLUE-STS 데이터셋의 예시이다.

```
klue_sts_train[0]
{'guid': 'klue-sts-v1_train_00000',
 'labels': {'binary-label': 1, 'label': 3.7, 'real-label': 3.714285714285714},
 'sentence1': '숙소 위치는 찾기 쉽고 일반적인 한국의 반지하 숙소입니다.',
 'sentence2': '숙박시설의 위치는 쉽게 찾을 수 있고 한국의 대표적인 반지하 숙박시설입니다.',
 'source': 'airbnb-rtt'}
```

### 1.2. NLI (Natural Language Inference)

NLI 데이터 또한 두 개의 문장 쌍이 제공되며 두 문장이 서로 수반(entailment) 관계인지, 모순(contradiction) 관계인지, 중립(neutral) 관계인지를 나타내는 라벨 값으로 구성된다.

아래는 KLUE-NLI 데이터셋의 예시이며, 'label'값의 경우 수치형으로 변환되어 있는데, 0은 entailment, 1은 neutral, 2는 contradiction을 나타낸다. (KorNLI 데이터는 라벨값이 텍스트로 되어있다.)

```
klue_nli_train[0]
```

```
{'guid': 'klue-nli-v1_train_00000',  
 'hypothesis': '햇볕 진심 최고로 멋지다.',  
 'label': 0,  
 'premise': '햇볕 진심 최고다 그 어떤 히어로보다 멋지다',  
 'source': 'NSMC'}
```

## 2. SBERT 학습 방법

논문에 의하면 SBERT의 학습 방법은 크게 2가지인데, 첫 번째는 **STS 데이터만을 통해 학습하는 방법**이고, 두 번째는 **NLI 데이터로 fine-tuning된 모델을 STS로 추가 학습 시키는 continue learning 방법**이다.

### 2.1. STS 단일 데이터를 이용한 Fine-tuning

해당 섹션의 **colab** 실습 코드는 [링크](#)를 참조하세요.

SBERT를 학습하는 가장 기본적이면서도 강력한 방법은 STS 데이터를 통해 fine-tuning하는 것이며, regression objective function에 의해 학습된다.

#### 학습 방법

1. 두 개의 문장 쌍 입력
2. 사전학습된 BERT에 의해 각 입력 시퀀스를 임베딩 벡터로 변환
3. 변환된 임베딩 벡터들에 대해 Pooling 연산(일반적으로 Mean-pooling)을 수행하여 문장 임베딩 벡터로 변환
4. 변환된 두 문장 임베딩 벡터를 cosine similarity를 통해 두 문장 벡터의 유사도 값(-1 ~ 1) 계산

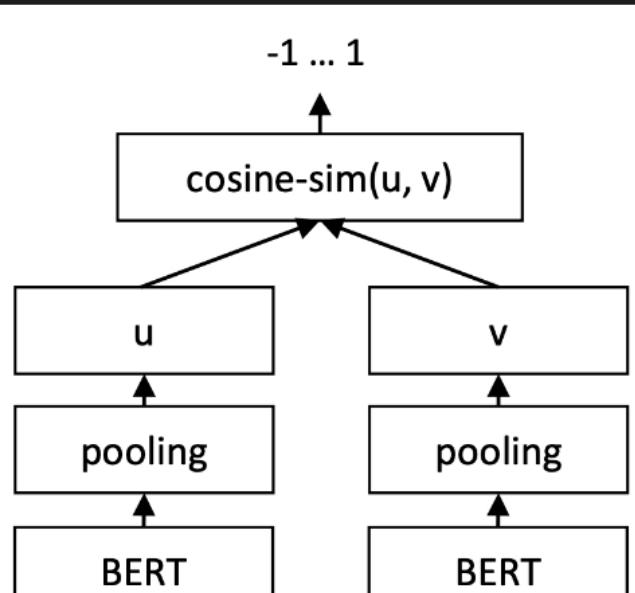




Figure 2: SBERT architecture at inference, for example, to compute similarity scores. This architecture is also used with the regression objective function.

### 2.1.1. Load Dataset

실습에 사용할 데이터는 klue-sts 데이터셋이다. 해당 데이터셋은 'train', 'validation'으로만 구성되어 있지만, 'train'셋의 10%를 샘플링하여 validation 으로 사용하고 기존 'validation'을 test용도로 사용하였다.

```

from datasets import load_dataset

# load KLUE-STS Dataset
klue_sts_train = load_dataset("klue", "sts", split='train[:90%]')
klue_sts_valid = load_dataset("klue", "sts", split='train[-10%:]') # train의 10%를 validation으로 사용
klue_sts_test = load_dataset("klue", "sts", split='validation')

print('Length of Train : ', len(klue_sts_train)) # 10501
print('Length of Valid : ', len(klue_sts_valid)) # 1167
print('Length of Test : ', len(klue_sts_test)) # 519

```

### 2.1.2. Preprocessing

`InputExample()` 클래스를 통해, 두 개의 문장 쌍과 라벨을 묶어 모델이 학습할 수 있는 형태로 변환해준 후,

```

from sentence_transformers.readers import InputExample

def make_sts_input_example(dataset):
    ...
    Transform to InputExample
    ...
    input_examples = []
    for i, data in enumerate(dataset):
        sentence1 = data['sentence1']
        sentence2 = data['sentence2']
        score = (data['labels']['label']) / 5.0 # normalize 0 to 5
        input_examples.append(InputExample(texts=[sentence1, sentence2], label=score))

    return input_examples

sts_train_examples = make_sts_input_example(klue_sts_train)
sts_valid_examples = make_sts_input_example(klue_sts_valid)
sts_test_examples = make_sts_input_example(klue_sts_test)

```

학습에 사용할 train 데이터는 배치학습을 위해 `DataLoader()`로 묶고,

`EmbeddingSimilarityEvaluator()` 을 통해 학습 시 사용할 validation 검증기와 모델 평가 시

사용할 test 검증기를 만들었다.

```
from torch.utils.data import DataLoader
from sentence_transformers.evaluation import EmbeddingSimilarityEvaluator

# Train Dataloader
train_dataloader = DataLoader(
    sts_train_examples,
    shuffle=True,
    batch_size=train_batch_size, # 32 (논문에서는 16)
)

# Evaluator by sts-validation
dev_evaluator = EmbeddingSimilarityEvaluator.from_input_examples(
    sts_valid_examples,
    name="sts-dev",
)

# Evaluator by sts-test
test_evaluator = EmbeddingSimilarityEvaluator.from_input_examples(
    sts_test_examples,
    name="sts-test",
)
```

### 2.1.3. Load Pretrained Model

STS fine-tuning을 위해 사용할 사전학습언어모델을 로드하는 과정이며, 해당 실습에서는 huggingface model hub에 공개되어있는 **klue/roberta-base** 모델을 사용하였다.

Pooling 레이어의 경우, 논문 실험 기준 가장 성능이 좋은 mean pooling을 정의하였다.

```
from sentence_transformers import SentenceTransformer, models

# Load Embedding Model
embedding_model = models.Transformer(
    model_name_or_path="klue/roberta-base",
    max_seq_length=256,
    do_lower_case=True
)

# Only use Mean Pooling -> Pooling all token embedding vectors of sentence.
pooling_model = models.Pooling(
    embedding_model.get_word_embedding_dimension(),
    pooling_mode_mean_tokens=True,
    pooling_mode_cls_token=False,
    pooling_mode_max_tokens=False,
)

model = SentenceTransformer(modules=[embedding_model, pooling_model])
```

### 2.1.4. Training by STS

STS 학습 시 loss function의 경우 `CosineSimilarityLoss()`를 사용하며, 논문과 동일하게 4 epochs, learning-rate warm-up의 경우 train의 10%를 설정하였다.

```
from sentence_transformers import losses

# config
sts_num_epochs = 4
```

```

train_batch_size = 32
sts_model_save_path = 'output/training_sts-' + pretrained_model_name.replace("/", "-") + '-' + d

# Use CosineSimilarityLoss
train_loss = losses.CosineSimilarityLoss(model=model)
# linear learning-rate warmup steps
warmup_steps = math.ceil(len(sts_train_examples) * sts_num_epochs / train_batch_size * 0.1)
# Training
model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    evaluator=dev_evaluator,
    epochs=sts_num_epochs,
    evaluation_steps=int(len(train_dataloader)*0.1),
    warmup_steps=warmup_steps,
    output_path=sts_model_save_path
)

```

### 2.1.5. Evaluation

위에서 정의해 두었던 `test` 검증기로 모델 성능을 평가한 결과 약 **0.88**의 성능을 나타내었다.

```
# evaluation sts-test
test_evaluator(model, output_path=sts_model_save_path)
```

```

2022-02-25 02:15:39 - EmbeddingSimilarityEvaluator: Evaluating the model on sts-test data
2022-02-25 02:15:43 - Cosine-Similarity : Pearson: 0.8870 Spearman: 0.8873
2022-02-25 02:15:43 - Manhattan-Distance: Pearson: 0.8862 Spearman: 0.8835
2022-02-25 02:15:43 - Euclidean-Distance: Pearson: 0.8869 Spearman: 0.8844
2022-02-25 02:15:43 - Dot-Product-Similarity: Pearson: 0.8775 Spearman: 0.8745
0.887279591001845

```

## 2.2. NLI와 STS 데이터를 활용한 Continue Learning

해당 섹션의 `colab` 실습 코드는 [링크](#)를 참조하세요.

STS 단일 데이터로 학습하는 방법 외에, NLI로 학습 후 STS로 추가학습을 하는 방법도 존재한다. 논문에 의하면 STS 단일 데이터로 학습했을 때 보다 NLI 학습 후 STS를 이어서 학습하는 방법이 약 3~4 point 더 높은 퍼포먼스를 나타내었으며 (아래 Table 2 참조), 이러한 전략이 특히 BERT cross-encoder 방식에 큰 영향을 주었다고 한다.

Model	Spearman
<i>Not trained for STS</i>	
Avg. GloVe embeddings	58.02
Avg. BERT embeddings	46.35
InferSent - GloVe	68.03
Universal Sentence Encoder	74.92
SBERT-NLI-base	77.03
SBERT-NLI-large	79.23

<i>Trained on STS benchmark dataset</i>	
BERT-STSb-base	$84.30 \pm 0.76$
SBERT-STSb-base	$84.67 \pm 0.19$
SRoBERTa-STSb-base	<b><math>84.92 \pm 0.34</math></b>
BERT-STSb-large	<b><math>85.64 \pm 0.81</math></b>
SBERT-STSb-large	$84.45 \pm 0.43$
SRoBERTa-STSb-large	$85.02 \pm 0.76$
<i>Trained on NLI data + STS benchmark data</i>	
BERT-NLI-STSb-base	<b><math>88.33 \pm 0.19</math></b>
SBERT-NLI-STSb-base	$85.35 \pm 0.17$
SRoBERTa-NLI-STSb-base	$84.79 \pm 0.38$
BERT-NLI-STSb-large	<b><math>88.77 \pm 0.46</math></b>
SBERT-NLI-STSb-large	$86.10 \pm 0.13$
SRoBERTa-NLI-STSb-large	$86.15 \pm 0.35$

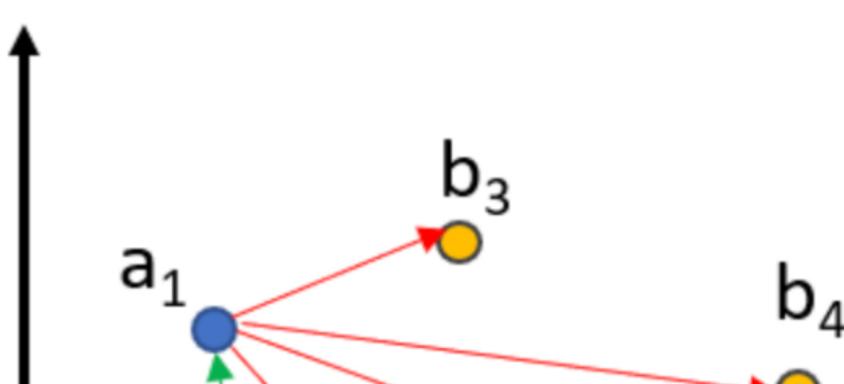
Table 2: Evaluation on the STS benchmark test set. BERT systems were trained with 10 random seeds and 4 epochs. SBERT was fine-tuned on the STSb dataset, SBERT-NLI was pretrained on the NLI datasets, then fine-tuned on the STSb dataset.

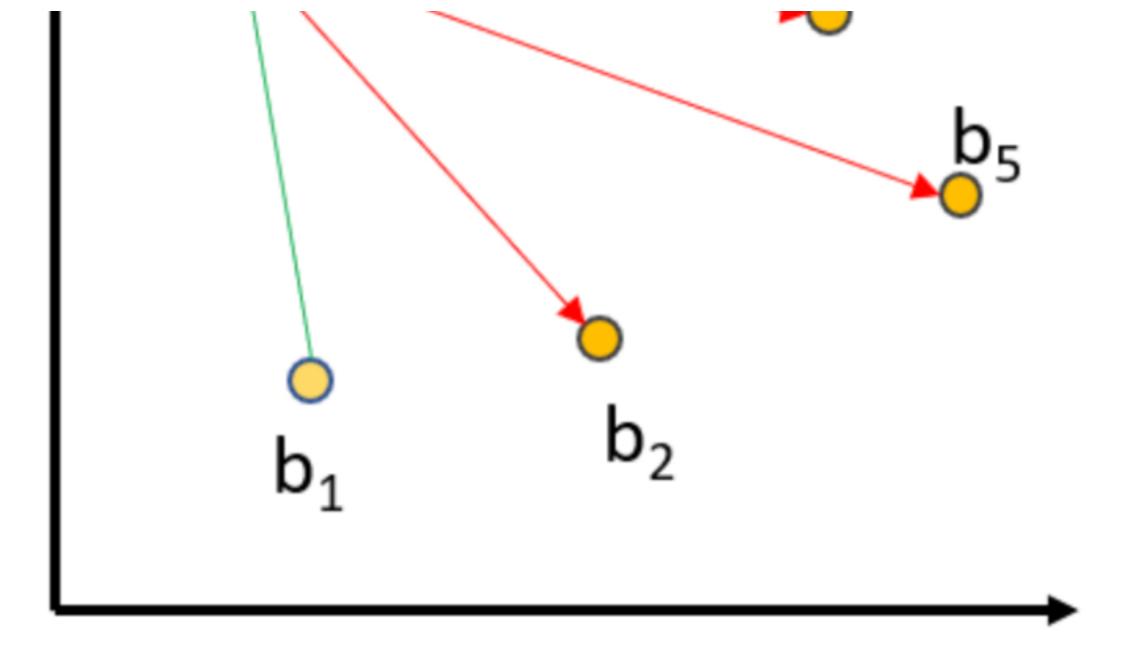
우선 NLI학습을 위해 논문에서는 기본적으로 3개의 클래스(*entailment*, *contradiction*, *neutral*) 학습을 위해 Softmax objective function을 사용한다.

하지만, sentence-transformers의 NLI 학습 예제에 따르면 실험 결과 Softmax loss를 사용하는 것보다 **MultipleNegativesRanking loss (MNR loss)**를 통해 학습하는 것이 더 나은 성능을 보여주었다고 한다.

(실제 klue 데이터셋을 통해 학습한 결과에서도 MNR loss로 학습했을 때 성능이 극소하게 높았다.)

#### MultipleNegativesRanking loss





MNR loss는 triplet loss와 개념적으로 유사하며 anchor와 positive의 거리는 가까워지도록, anchor와 negative의 거리는 멀어지도록 학습을 유도한다. 즉 위 그림에서 ( $a_1, b_1$ )의 distance는 minimize하고, ( $a_1, b_2 \sim 4$ )의 distance는 maximize하도록 학습한다. (본 실습에서도 MNR loss를 활용한다.)

### 학습 방법

1. NLI 데이터를 triplet 형태로 구성 (anchor, positive, negative)
2. 사전학습된 BERT모델에 의해 각 입력 시퀀스를 임베딩 벡터로 변환
3. 변환된 임베딩 벡터들에 대해 Pooling 연산(일반적으로 Mean-pooling)을 수행하여 문장 임베딩 벡터로 변환
2. MNR loss를 objective function으로 NLI 데이터셋 fine-tuning
3. NLI를 통해 fine-tuning된 모델을 로드하여 STS 데이터로 추가 학습
4. 변환된 두 문장 임베딩 벡터를 cosine similarity를 통해 두 문장 벡터의 유사도 값(-1 ~ 1) 계산

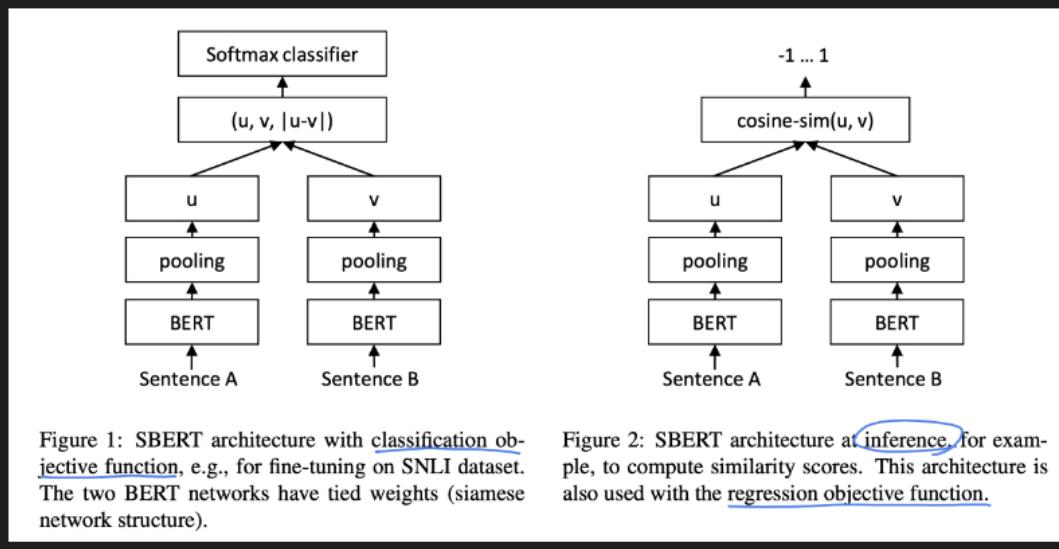


Figure 1: SBERT architecture with classification objective function, e.g., for fine-tuning on SNLI dataset. The two BERT networks have tied weights (siamese network structure).

Figure 2: SBERT architecture at inference, for example, to compute similarity scores. This architecture is also used with the regression objective function.

아래 실습코드에서는 위에서 설명한 *STS finetuning*에 대한 내용은 생략한다.

### 2.2.1. Load Dataset (NLI)

continue learning을 위하여 nli, sts 데이터를 모두 로드할 것이며, nli의 경우 train 데이터만 로드한다.

```
# load KLUE-NLI Dataset
klue_nli_train = load_dataset("klue", "nli", split='train')

print('Length of Train : ', len(klue_nli_train)) # 24998
```

### 2.2.2. Preprocessing (NLI)

MNR loss를 통한 학습을 위해 triplet (anchor sentence, positive sentence, negative sentence) 포맷으로 맞춰준 후 마찬가지로 `InputExample()`로 변환시켜준다.

```
def make_nli_triplet_input_example(dataset):
    ...
    Transform to Triplet format and InputExample
    ...
    # transform to Triplet format
    train_data = {}
    def add_to_samples(sent1, sent2, label):
        if sent1 not in train_data:
            train_data[sent1] = {'contradiction': set(), 'entailment': set(), 'neutral': s
        train_data[sent1][label].add(sent2)

    for i, data in enumerate(dataset):
        sent1 = data['hypothesis'].strip()
        sent2 = data['premise'].strip()
        if data['label'] == 0:
            label = 'entailment'
        elif data['label'] == 1:
            label = 'neutral'
        else:
            label = 'contradiction'

        add_to_samples(sent1, sent2, label)
        add_to_samples(sent2, sent1, label) #Also add the opposite

    # transform to InputExmaples
    input_examples = []
    for sent1, others in train_data.items():
        if len(others['entailment']) > 0 and len(others['contradiction']) > 0:
            input_examples.append(InputExample(texts=[sent1, random.choice(list(others['en
            input_examples.append(InputExample(texts=[random.choice(list(others['entailmen

    return input_examples

nli_train_examples = make_nli_triplet_input_example(klue_nli_train)
nli_train_examples[0].texts # ['헛걸 진심 최고다 그 어떤 히어로보다 멋진다', '헛걸 진심 최고로 멋진다']
```

이후 학습에 사용할 train 데이터는 배치학습을 위해 `DataLoader()`로 변환한다. 참고로 위 NLI 학습시에는 검증 데이터로 STS 데이터셋을 사용하기 때문에 따로 검증기를 만들지 않는다.

```

# Train Dataloader
train_dataloader = DataLoader(
    nli_train_examples,
    shuffle=True,
    batch_size=train_batch_size,
)

```

### 2.2.3. Load Pretrained Model

NLI fine-tuning을 위해 사용할 사전학습언어모델을 로드하는 과정이며, 해당 실습에서는 huggingface model hub에 공개되어있는 [klue/roberta-base](#) 모델을 사용하였다.

Pooling 레이어의 경우, 논문 실험 기준 가장 성능이 좋은 mean pooling을 정의하였다.

```

from sentence_transformers import SentenceTransformer, models

# Load Embedding Model
embedding_model = models.Transformer(
    model_name_or_path="klue/roberta-base",
    max_seq_length=256,
    do_lower_case=True
)

# Only use Mean Pooling -> Pooling all token embedding vectors of sentence.
pooling_model = models.Pooling(
    embedding_model.get_word_embedding_dimension(),
    pooling_mode_mean_tokens=True,
    pooling_mode_cls_token=False,
    pooling_mode_max_tokens=False,
)

model = SentenceTransformer(modules=[embedding_model, pooling_model])

```

### 2.2.4. Training by NLI

위에서 설명했듯이 loss function의 경우 `MultipleNegativesRankingLoss()`를 사용하며, 논문과 동일하게 1 epochs, learning-rate warm-up의 경우 train의 10%를 설정하였다.

```

from sentence_transformers import losses

# config
sts_num_epochs = 1
train_batch_size = 32
nli_model_save_path = 'output/training_nli_by_MNRLoss_' + pretrained_model_name.replace("/", "")

# Use MultipleNegativesRankingLoss
train_loss = losses.MultipleNegativesRankingLoss(model)
# warmup steps
warmup_steps = math.ceil(len(nli_train_examples) * nli_num_epochs / train_batch_size * 0.1)
logging.info("Warmup-steps: {}".format(warmup_steps))
# Training
model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    evaluator=dev_evaluator,
    epochs=nli_num_epochs,
    evaluation_steps=int(len(train_dataloader)*0.1),
    warmup_steps=warmup_steps,
    output_path=nli_model_save_path,
    use_amp=False           #Set to True, if your GPU supports FP16 operations
)

```

## 2.2.5. Continue Learning by STS

NLI 데이터셋을 통해 finetuning된 모델에 STS를 추가학습 하는 과정이다. 학습이 완료된 모델을 로드 후,

```
# Load model of fine-tuning by NLI
model = SentenceTransformer(nli_model_save_path)
```

STS 데이터셋을 통해 continue learning을 수행한다.

```
# config
sts_num_epochs = 4
train_batch_size = 32
sts_model_save_path = 'output/training_sts_continue_training-' + pretrained_model_name.replace(
    '-', '_')

# Use CosineSimilarityLoss
train_loss = losses.CosineSimilarityLoss(model=model)
# warmup steps
warmup_steps = math.ceil(len(sts_train_examples) * sts_num_epochs / train_batch_size * 0.1)
logging.info("Warmup-steps: {}".format(warmup_steps))
# Training
model.fit(
    train_objectives=[(train_dataloader, train_loss)],
    evaluator=dev_evaluator,
    epochs=sts_num_epochs,
    evaluation_steps=int(len(train_dataloader)*0.1),
    warmup_steps=warmup_steps,
    output_path=sts_model_save_path
)
```

## 2.2.6. Evaluation

위에서 정의해 두었던 test 검증기로 모델 성능을 평가한 결과 약 0.89의 성능을 나타내었으며, 단일 STS를 통해 학습했을 때에 비해 약 1% 가량 성능 향상이 있었다.

```
# evaluation sts-test
test_evaluator(model, output_path=sts_model_save_path)
```

```
2022-02-25 04:28:11 - EmbeddingSimilarityEvaluator: Evaluating the model on sts-test datas
2022-02-25 04:28:15 - Cosine-Similarity : Pearson: 0.8962 Spearman: 0.8964
2022-02-25 04:28:15 - Manhattan-Distance: Pearson: 0.8895 Spearman: 0.8845
2022-02-25 04:28:15 - Euclidean-Distance: Pearson: 0.8908 Spearman: 0.8859
2022-02-25 04:28:15 - Dot-Product-Similarity: Pearson: 0.8847 Spearman: 0.8810
0.896394981925387
```

# 3. Conclusion

STS만 학습, softmax loss를 활용한 continue learning, MNR loss를 활용한 continue learning 총 3가지의 케이스로 실험해본 결과 근소한 차이지만, MNR loss를 활용한 continue learning

방식이 성능이 가장 좋았다.

	Cosine-Similarity		Manhattan-Distance		Euclidean-Distance		Dot-Product-Similarity	
	Pearson	Spearman	Pearson	Spearman	Pearson	Spearman	Pearson	Spearman
Only STS	0.887	0.8873	0.8862	0.8835	0.8869	0.8844	0.8775	0.8745
NLI(softmax)+STS	0.879	0.9797	0.876	0.873	0.8761	0.873	0.8716	0.8692
NLI(MNR)+STS	0.8962	0.8964	0.8895	0.8845	0.8908	0.8859	0.8847	0.881

일반적으로는 sts 성능 평가를 위해 위에서도 언급했던 kornlu 데이터셋을 많이 사용하고 있는 추세이다.

하지만 kornlu 데이터의 경우 klue에 비해 데이터의 수는 월등히 많지만 데이터의 복잡도가 굉장히 낮은 데이터이기 리얼 월드의 sentence embedding이 목적이라면 klue 데이터를 활용하는 것이 경험상 더 좋은 품질의 임베딩이 가능했고, 성능도 더 좋았다.

이쯤에서 분명 그렇다면 "klue와 kornlu를 모두 학습하면 좋지 않을까?"라는 생각을 하는 사람이 있을텐데, klue와 kornlu의 유사도를 측정하는 기준 즉, 라벨링 기준이 다르기 때문에 오히려 모델에 혼란만 주게되는 결과를 가져오게 되므로 경험상 단일 데이터를 사용하는 것이 좋다고 생각된다.



jaehyeong.an\_

🌙 Don't be a knew-it-all, Be a Learn-it-all

팔로우



이전 포스트

[Basic NLP] HuggingFace에 내 ...

다음 포스트

[Basic NLP] Google Cloud-TPU...



7개의 댓글

댓글을 작성하세요



옐란

2022년 10월 19일

해당 섹션의 `colab` 실습 코드는 링크를 참조하세요.

=> 실습 코드(google drive) 파일경로가 사라졌거나, 잘못됬나봅니다. ("죄송합니다. 요청한 파일이 없습니다." 라고 메세지가 링크페이지에서 뜹니다)

=> (자답) 상단 '본 포스트에서 사용된 Colab 실습 코드'에 링크를 클릭하니깐 확인 가능하네요

~~

#### 田 1개의 답글



hoonding

2022년 11월 10일

혹시 학습시킨 transformer모델을 저장하는 방법 알 수 있을까요?

#### 田 1개의 답글



연찐두빵

2023년 7월 14일

2.15 과정에서 만들어둔 사전데이터가 아닌 입력값을 따로 입력해 출력하는 `query`문은 사용이 불가한가요?

#### 田 1개의 답글



김도은

2023년 12월 7일

`klue_sts_train[0]`에서 `'labels': {'label': 3.7, 'real-label': 3.714285714285714, 'binary-label': 1}` 이렇게 출력되는데 이 라벨은 인덱스 값인가요?

#### 田 답글 달기

## 관심 있을 만한 포스트



# Sentence-BERT: Sentence Embedding using Siamese BERT-Networks

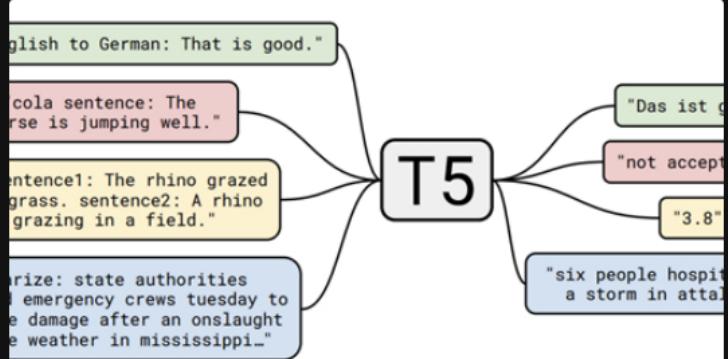
## [Paper Review] Sentence-BERT: Sentence Embedding us...

Intro 문장 간(혹은 문장 간) 유사도 분석에서 좋은 성능을 내고 있는 Sentence-BERT에 대해 알아보려고 한다. 논문 주제는 Sentence-BERT: Sentence Embedding using Siamese BERT-Networks이며, 최근 성능이...

2021년 10월 10일 · 0개의 댓글

by jaehyeong.an\_

3



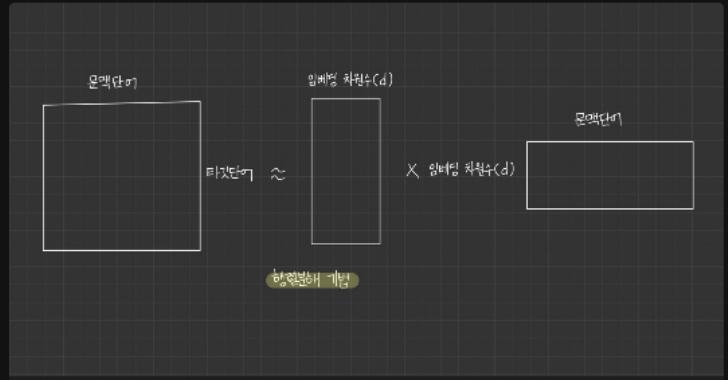
## T5(Text-to-Text Transfer Transformer) 논문 리뷰

Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer(2019) 읽어보기

2023년 4월 27일 · 0개의 댓글

by 정예슬

3



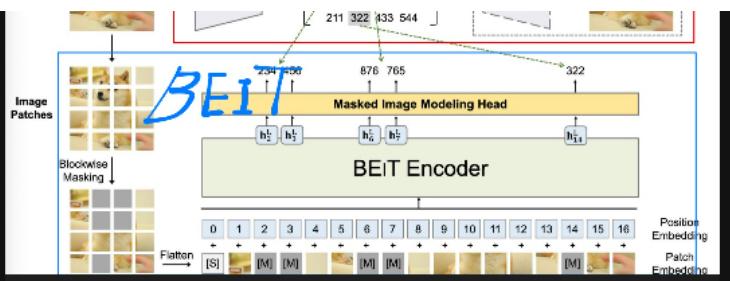
## 임베딩(Embedding)이란?

임베딩이란 자연어처리에서 사람이 쓰는 자연어를 기계가 이해할 수 있도록 숫자형태인 vector로 바꾸는 과정 혹은 일련의 전체 과정을 의미합니다. 단어나 문장 각각을 벡터로 변환해 벡터 공간(Vector space)으로 ...

2022년 8월 7일 · 0개의 댓글

by AI Scientist 를 목표로

4



## [논문리뷰]BEiT : Pre-Training of Image Transformer

Title : BEiT : Pre-Training of Image Transformer Date : 15 Jun 2021  
Keywords : Autoencoder, Self-Supervised, Vision Transformer, BERT, Tokenize Title ...

2022년 3월 7일 · 0개의 댓글

by seong taek

4



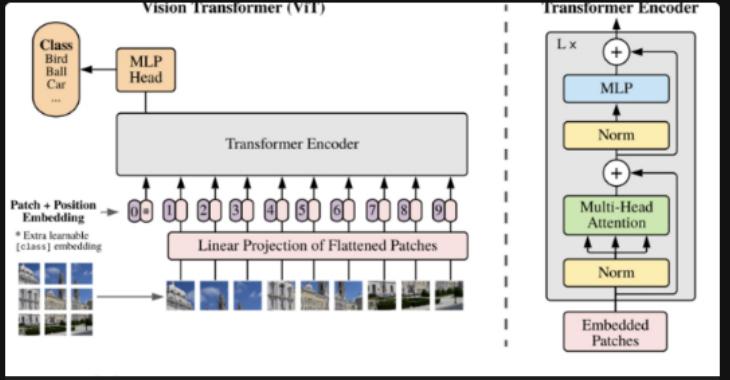
## [파이썬]일상/연애 주제의 한국어 대화 'BERT'로 이진 분류 ...

BERT를 이용한 프로젝트 - 이론편입니다!

2021년 6월 22일 · 0개의 댓글

by Seolini

6



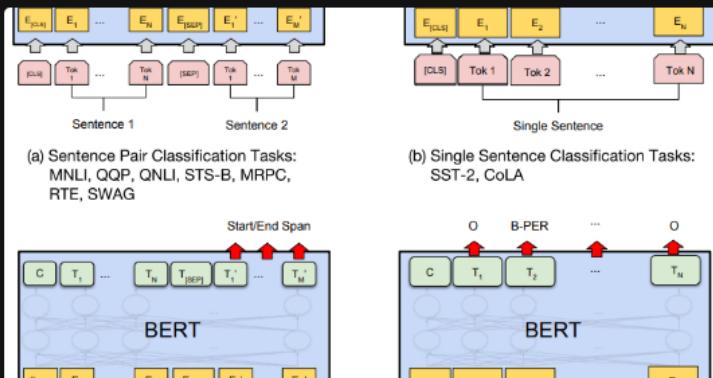
## Vision Transformer(ViT) 논문 리뷰

ViT(비전 트랜스포머) 논문 읽기

2023년 4월 9일 · 2개의 댓글

by 정예슬

1



BERT

<https://keep-steady.tistory.com/19> BERT설명내용-

<https://github.com/chullhwan-song/Reading-Paper/issues/202> BERT 참고링크 1) Illustrated Bert: <http://jalamarra...>

2020년 8월 7일 · 0개의 댓글

ml by ml test

5



트랜스포머(Transformer)와 어텐션 매커니즘(Attention Mechanism)

1. 배경(Background) 2017년에 Attention is All You Need라는 논문이 나온 이후로 어텐션 구조는 최초에 제안되었던 NLP 분야는 물론 Computer Vision 분야와 Time Series 분야까지 다양한 분야에서 적용...

2022년 12월 14일 · 4개의 댓글

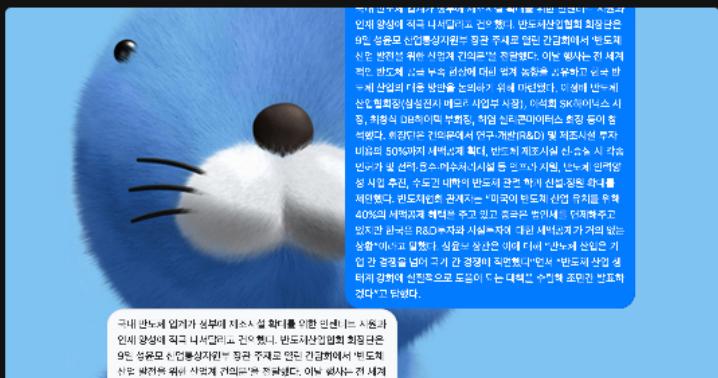
by 환공지능

22



[Basic NLP] Transformers와 Tensorflow를 활용한 BERT Fine-tuning

이번 포스트에서는 🤖 HuggingFace의 Transformers 라이브러리와



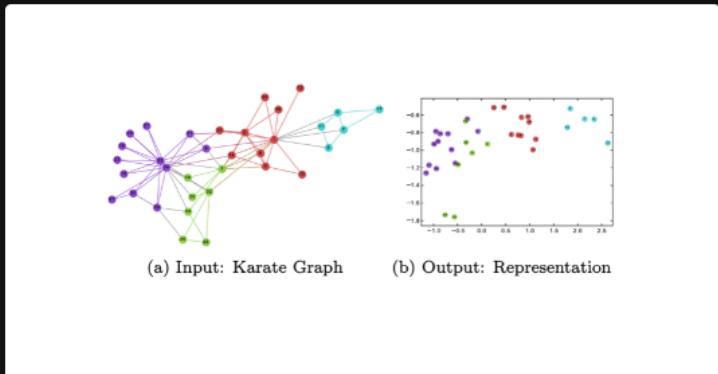
## BERT를 활용한 한국어 문서 추출요약 봇

딥러닝 기반의 여러 요약 모델을 공부하고 있던 중, 한국어 데이터로 학습한 추출요약 모델이 있으면 좋겠다 싶어서 만들어 보았습니다. 보노보노는 뭔가 허전해서 넣었습니다. 감사합니다.

2021년 4월 10일 · 21개의 댓글

 by **raqoon**

17



DeepWalk - graph neural network

오늘은 그래프 형태의 데이터를 효과적으로 임베딩할 수 있는 방법 중 하나인 DeepWalk에 대해서 써보려고 합니다.

2020년 4월 25일 · 0개의 댓글

by 황태용

8

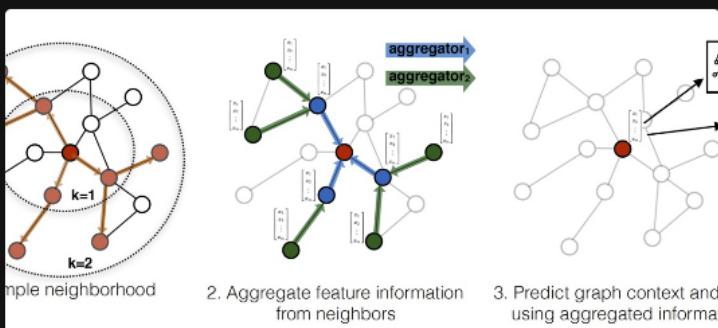


Figure 1: Visual illustration of the GraphSAGE sample and aggregate approach.

[논문리뷰] GraphSage : Inductive Representation Learning...

오늘은 graphsage라는 프레임워크를 제안한 논문에 대해 다루어 보겠습니다.

Tensorflow를 통해 사전 학습된 BERT모델을 Fine-tuning하여 Multi-Class Text Classification을 수행하는 방법에 대해 알아보고자 한다. 특...

2021년 8월 7일 · 1개의 댓글

 by [jaehyeong.an\\_](#)

♥ 6

니다. Node2vec이나 DeepWalk와 같은 shallow node embedding learning을 통해 고정된 node들에 대한 representation을 학습할 수 있...

2022년 7월 13일 · 1개의 댓글

 by [Dong Jun](#)

♥ 3

