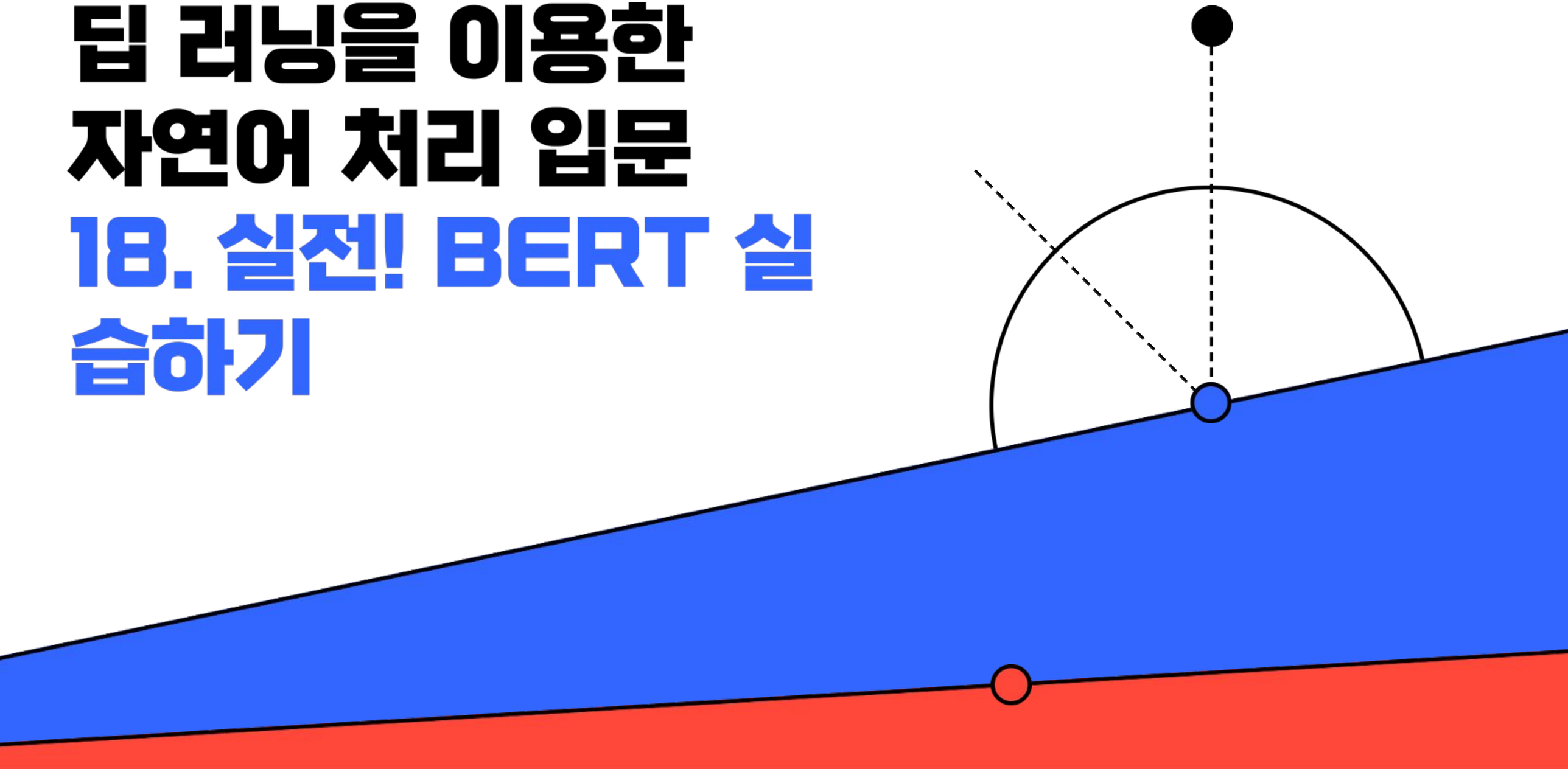
딥 러닝을 이용한 자연어 처리 입문

18. 실전! BERT 실습하기



18-01 코랩(Colab)에서 TPU 사용하기

: GPU보다 더 빠른 TPU 사용

GPU와 TPU 차이?

GPU TPU

GPU(그래픽 처리 장치)
병렬 처리를 통해 그래픽 작업을 가속화하는데 주로 사용되며, 머신러닝과 딥러닝에서도 널리 활용

TPU(텐서 처리 장치) 특히 **딥러닝** 작업을 위해 설계된 **하드웨어**로, 고도로 최적화되어 **효율적인 텐서 연산**을 수행하여 학습과 추론 속도를 크게 향상

18-01 코랩(Colab)에서 TPU 사용하기

: GPU보다 더 빠른 TPU 사용

1. 코랩(Colab)에서 TPU를 선택

Colab에서 런타임 > 런타임 유형 변경 > 하드웨어 가속기에서 'TPU' 선택

2. TPU 초기화
(딥 러닝 모델을 정의하기 전, 초반부에 실행)

```
import tensorflow as tf
import os

resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://' + os.environ['COLAB_TPU_ADD
R'])

tf.config.experimental_connect_to_cluster(resolver)
tf.tpu.experimental.initialize_tpu_system(resolver)
```

18-01 코랩(Colab)에서 TPU 사용하기

: GPU보다 더 빠른 TPU 사용

3. TPU Strategy 셋팅

여러 TPU로 나누어 처리하기 위한 텐서플로 API , 이 API를 사용해서 분산처리

```
strategy = tf.distribute.TPUStrategy(resolver)
```

18-01 코랩(Colab)에서 TPU 사용하기

: GPU보다 더 빠른 TPU 사용

4. 딥 러닝 모델의 컴파일

모델의 층을 쌓는 create_model()라는 함수

```
def create_model():
  return tf.keras.Sequential(
      [tf.keras.layers.Conv2D(256, 3, activation='relu', input_shape=(28, 28, 1)),
       tf.keras.layers.Conv2D(256, 3, activation='relu'),
       tf.keras.layers.Flatten(),
       tf.keras.layers.Dense(256, activation='relu'),
       tf.keras.layers.Dense(128, activation='relu'),
       tf.keras.layers.Dense(10)])
```

18-01 코랩(Colab)에서 TPU 사용하기

: GPU보다 더 빠른 TPU 사용

4. 딥 러닝 모델의 컴파일

create_model() 함수를 호출하고 strategy.scope 내에서 모델을 컴파일

```
with strategy.scope():
  model = create_model()
  model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                metrics=['sparse_categorical_accuracy'])
```

> 이 모델을 fit() 하게되면 해당 모델은 TPU를 사용하며 학습하게 됨
GPU 환경에서 실습하고 싶다면, TPU 진행만을 위한 코드들을 전부 제거해주면 됨

18-02 transformers의 모델 클래스 불러오기

: transformers 라이브러리에서 BERT 위에 출력층을 추가한 모델 클래스 구현체를 제공

다 대 일 유형

```
from transformers import TFBertForSequenceClassification

model = TFBertForSequenceClassification.from_pretrained("모델 이름", num_labels=분류할 레이블의 개수)
```

Copy

다 대 다 유형

```
from transformers import TFBertForTokenClassification

model = TFBertForTokenClassification.from_pretrained("모델 이름", num_labels=분류할 레이블의 개수)
```

18-02 transformers의 모델 클래스 불러오기

: transformers 라이브러리에서 BERT 위에 출력층을 추가한 모델 클래스 구현체를 제공

질의응답 유형

```
from transformers import TFBertForQuestionAnswering

model = TFBertForQuestionAnswering.from_pretrained('모델 이름')
```

Copy

> 이렇게 이미 출력층이 설계된 모델들을 사용하는 것이 훨씬 코드 작성이 간편

18-08 BERT의 문장 임베딩(SBERT)을 이용한 한국어 챗봇

- sentence_transformers: SBERT를 이용하여 문장 임베딩을 얻을 수 있는 패키지
- 100가지 언어를 지원(한국어 포함)하는 다국어 BERT BASE 모델 로드

```
model = SentenceTransformer('sentence-transformers/xlm-r-100langs-bert-base-nli-stsb-mean-tokens')
```

문장 임베딩 값을 구하기

```
train_data['embedding'] = train_data.apply(lambda row: model.encode(row.Q), axis = 1)
```

18-08 BERT의 문장 임베딩(SBERT)을 이용한 한국어 챗봇

코사인 유사도 구하는 함수 정의

```
def cos_sim(A, B):
  return dot(A, B)/(norm(A)*norm(B))
```

임의의 질문의 임베딩값과 가장 유사함 질문을 찾아 답변을 리턴

```
def return_answer(question)
    embedding = model.encode(question)
    train_data['score'] = train_data.apply(lambda x: cos_sim(x['embedding'], embedding), axis=1)
    return train_data.loc[train_data['score'].idxmax()]['A']
```

18-08 BERT의 문장 임베딩(SBERT)을 이용한 한국어 챗봇

챗봇 테스트 결과

```
return_answer('결혼하고싶어')
```

```
좋은 사람이랑 결혼할 수 있을 거예요.
```

```
return_answer('나랑 커피먹을래?')
```

```
카페인이 필요한 시간인가 봐요.
```

```
return_answer('반가워')
```

```
저도 반가워요.
```

```
return_answer('사랑해')
```

```
상대방에게 전해보세요.
```

18-09 Faiss와 SBERT를 이용한 시맨틱 검색기 (Semantic Search)

시맨틱 검색(Semantic search)이란?
기존의 키워드 매칭이 아닌 문장의 의미에 초점을 맞춘 정보 검색 시스템

> SBERT와 FAISS를 사용하여 간단한 검색 엔진을 구현
텍스트 데이터를 의미론적으로 임베딩(벡터화)하여 유사도를 측정 > 검색기능
ex. 검색엔진, 자연어 질의 응답 시스템, 추천 시스템, 문서 클러스터링

Faiss: 데이터에 대한 효율적인 검색을 수행하기 위한
Facebook AI에서 구축한 C ++ 기반 라이브러리

18-09 Faiss와 SBERT를 이용한 시맨틱 검색기 (Semantic Search)

1. 모든 샘플에 대해서 SBERT 임베딩

```
model = SentenceTransformer('distilbert-base-nli-mean-tokens')
encoded_data = model.encode(data)
print('임베딩 된 벡터 수 :', len(encoded_data))
```

```
임베딩 된 벡터 수 : 1082168
```

18-09 Faiss와 SBERT를 이용한 시맨틱 검색기 (Semantic Search)

2. 인덱스 정의 및 데이터 추가
(768차원 벡터를 가진 데이터를 Faiss를 이용하여 검색 가능한 인덱스에 추가하는 작업)

```
index = faiss.IndexIDMap(faiss.IndexFlatIP(768))
index.add_with_ids(encoded_data, np.array(range(0, len(data))))


faiss.write_index(index, 'abc_news')
```

- 768차원의 평탄한(Flat) 인덱스를 생성 'IP는 Inner Product = 코사인 유사도

- 인덱스를 사용하여 ID 매핑을 추가한 IndexIDMap을 생성 > 데이터 포인트를 고유한 ID에 매핑

- 데이터의 개수에 해당하는 ID 배열을 생성하여 해당 ID에 맞추어 데이터를 추가

- 생성한 인덱스를 'abc_news'라는 파일명으로 저장

>> 벡터 검색을 효율적으로 수행

18-09 Faiss와 SBERT를 이용한 시맨틱 검색기 (Semantic Search)

3. 검색 및 시간 측정
(유사도가 높은 상위 5개의 샘플을 추출)

```python
def search(query):
    t = time.time()
    query_vector = model.encode([query])
    k = 5
    top_k = index.search(query_vector, k)
    print('total time: {}'.format(time.time() - t))
    return [data[_id] for _id in top_k[1].tolist()[0]]
```

```python
query = str(input())
results = search(query)

print('results :')
for result in results:
    print('\t', result)
```

```
Underwater Forest Discovered
total time: 1.069244384765625
results :
	underwater loop
	thriving underwater antarctic garden discovered
	baton goes underwater in wa
	underwater footage shows inside doomed costa
	underwater uluru found off wa coast
```

'Underwater Forest Discovered'라는
임의의 문장을 입력
> 약 108만개의 문서에 대해서 시맨틱 검색 수행했으나,
약 1초 내외의 시간밖에 걸리지 않음