1주차 - 트랜스포머 소개 (1.1절~2.2절)

1장 트랜스포머 소개

1.1 인코더 - 디코더 프레임워크

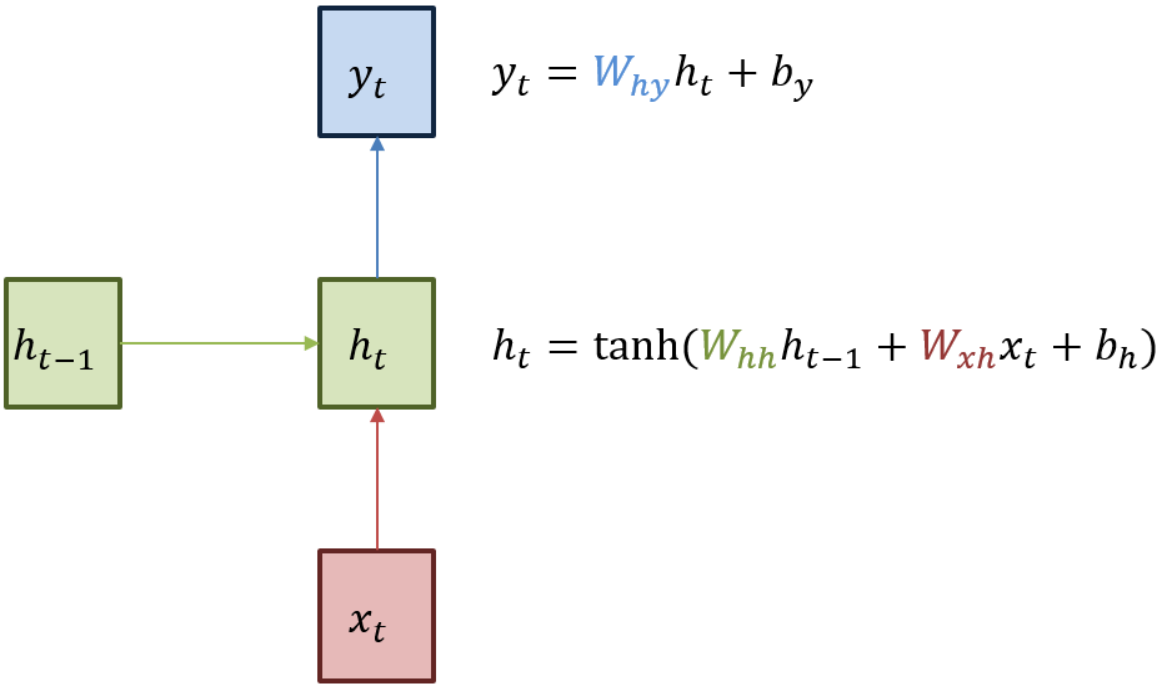
순환신경망(RNN)

```
class RNNCell_Encoder(nn.Module) : # 워드 임베딩 및 RNN cell 정의
    def __init__(self, input_dim, hidden_size) :
        super(RNNCell_Encoder, self).__init__()
        self.rnn = nn.RNNCell(input_dim, hidden_size) # rnn c

    def forward(self, inputs) :
```

```
        bz = inputs.shape[1]
        ht = torch.zeros((bz, hidden_size)).to(device) # 현재
        for word in inputs : # word : 현재 입력 벡터(x_t)
            ht = self.rnn(word, ht) # ht : 이전 상태(h_t-1)
        return ht
```

- `nn.RNNCell`
  - 입력과 이전 타임 스텝의 은닉 상태를 받아 현재 타임 스텝의 은닉 상태를 계산
  - 반복문을 통해 직접 각 단어에 대한 처리해준다.
- RNN 특징:
  - 각 타임 스텝에서 사용되는 가중치를 공유 - 시퀀스의 장기 의존성을 학습
  - 은닉 상태 상태를 출력 - 현재까지의 입력과 이전 타임 스텝에서의 은닉 상태를 기반으로 계산

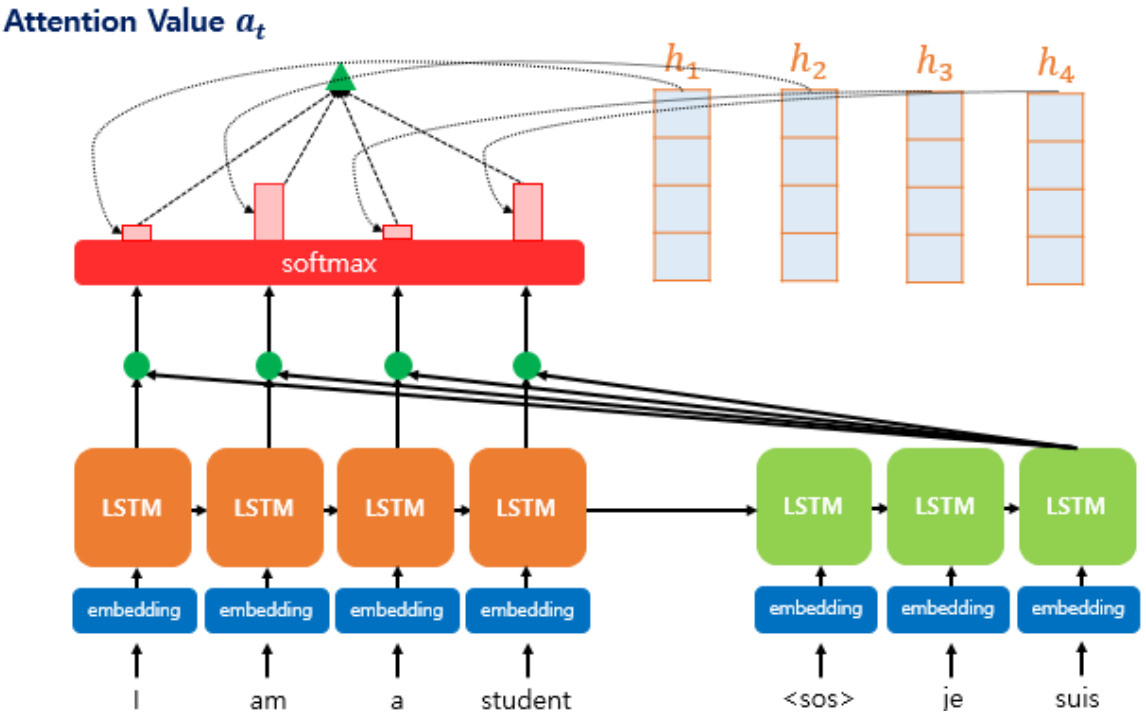


- 2개의 모든 타임 스텝에서 공유된 가중치 사용:
  1. 입력과 은닉 상태 사이의 가중치 행렬
  2. 은닉 상태와 은닉 상태 사이의 가중치 행렬
- 시퀀스-투-시퀀스 구조로 처리:

    - 입력과 출력이 임의의 길이를 가진 시퀀스인 task
- 인코더:
    - 입력 시퀀스의 정보를 last hidden state (vector)로 인코딩
- RNN의 단점:
    - 시퀀스가 길 경우, 초기 시퀀스 정보가 손실될 수 있음.

1.2 어텐션 메커니즘

- 셀프 어텐션 이전:



    - **매 타임 스텝마다** 인코더에서 디코더가 참고할 은 상태를 출력
    - 여전히 입력 시퀀스를 순차적으로 수행되므로, 입력 시퀀스 전체를 한번에 확인하면서 병렬적으로 작동하지 못하는 단점 존재
- 셀프 어텐션 등장:
    - 트랜스포머에서 제안
    - 훨씬 빠른 학습이 가능해짐

1.3 NLP의 전이 학습

- 이전에 컴퓨터 비전:
  - ResNet과 같은 사전 학습된 모델 기반 전이 학습이 당연시 됐음.
- OpenAI에서 제안한 ULMFiT:
  - NLP에서의 전이 학습 방식 제안 (2017, 2018 즈음)
- 언어 모델링 (language modeling):
  - 이전 단어를 바탕으로 다음 단어를 예측하는 것
- GPT:
  - 트랜스포머 아키텍처의 디코더 부분만 사용
- BERT:
  - 트랜스포머 아키텍처의 인코더 부분만 사용
  - 마스크드 언어 모델링 (masked language modeling):
    - 랜덤하게 마스킹된 단어를 예측하는 것

1.4 허깅페이스 트랜스포머스

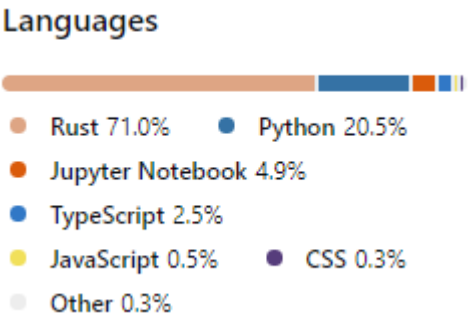
- 파이토치, 텐서플로우, JAX 세 개의 주요 딥러닝 프레임워크를 지원

1.6 허깅페이스 생태계

1.6.1 허깅페이스 허브

- 여러 모델 호스팅 진행

1.6.2 허깅페이스 토크나이저



- 토큰화:
  - 원시 텍스트를 토큰이라는 더 작은 단위로 분할
- 트랜스포머는 토큰의 수치 표현을 훈련한다.
- 허깅페이스 토크나이저는 러스트 백엔드 덕분에 매우 빠르게 텍스트를 토큰화한다.
  - Extremely fast (both training and tokenization), **thanks to the Rust implementation. Takes less than 20 seconds to tokenize a GB of text on a server's CPU.**
- 사용 가능한 모델:
  - Byte-Pair Encoding, WordPiece or Unigram

1.6.3 허깅페이스 데이터셋

- 수천 개의 데이터셋에 대한 표준 인터페이스 제공
- 스마트한 캐싱을 제공
  - The cache is one of the reasons why 🤗 Datasets is so efficient.
  - so when you need to use them again, they are reloaded directly from the cache.
  - avoids having to download a dataset all over again, or reapplying processing functions. **Even after you close and start another Python session**, 🤗 Datasets will reload your dataset directly from the cache!
  - Datasets assigns a fingerprint to the cache file. A fingerprint keeps track of the current state of a dataset.
  - The initial fingerprint is computed using a hash from the Arrow table, or a hash of the Arrow files if the dataset is on disk.
  - when caching is disabled:
    - the cache files are generated every time and they get written to a temporary directory
- 메모리 매핑이라는 메커니즘을 활용해 램 부족을 피한다.
  - 파일 내용을 가상 메모리에 저장하고 여러 개의 프로세스로 더 효율적으로 파일을 수정
  - 🤗 Datasets uses Arrow for its local caching system

- It allows datasets to be backed by an on-disk cache (memory-mapped for fast lookup)
- allows for large datasets to be used on machines with relatively small device memory.
- 적은 메모리 사용 이유:
  - the Arrow data is **actually memory-mapped from disk**, and not loaded in memory
  - Memory-mapping **allows access to data on disk**, and **leverages virtual memory capabilities for fast lookups.**

1.6.4 허깅페이스 액셀러레이트

- 허깅페이스의 Accelerated Inference API
- 미세 튜닝을 위해 사용
- 🤗 Accelerate is a library that enables the same PyTorch code to be run across any distributed configuration by adding just four lines of code!
- training and inference at scale made simple, efficient and adaptable.
- 하나의 머신에 여러 개의 GPU를 사용하든 여러 머신에 여러 개의 GPU를 사용하든 모든 유형의 분산 설정에서 🤗 Transformers 모델을 쉽게 훈련할 수 있도록 돕기 위해 🤗 **Accelerate** 라이브러리가 등장

2장 텍스트 분류

- 체크포인트 (checkpoint)

2.1 데이터셋

2.1.1 허깅페이스 데이터셋 처음 사용하기

- Dataset 객체:

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label', 'label_text'],
        num_rows: 16000
```

```
    })
    test: Dataset({
        features: ['text', 'label', 'label_text'],
        num_rows: 2000
    })
    validation: Dataset({
        features: ['text', 'label', 'label_text'],
        num_rows: 2000
    })
})
```

2.1.2 데이터셋에서 데이터프레임으로

- `set_format()`
  - Dataset의 출력 포맷 변경

```
emotions.set_format(type="pandas")
df = emotions['train'][:]
```

- maximum context size:
  - 모델마다 최대 입력 길이가 있다.

2.2 텍스트에서 토큰으로

- 트랜스포머 모델은 텍스트를 토큰화해서 수치 벡터로 인코딩한다.

2.2.1 문자 토큰화

- 각 문자를 개별로 모델에 주입
- 장점:
  - 철자 오류, 희귀한 단어를 처리하는데 유용
- 단점:
  - 상당량의 계산, 데이터 필요

2.2.2 단어 토큰화

- 이 세상의 다양한 텍스트 표현을 모두 개별적으로 토큰으로 볼 경우 = 어휘사전을 비효율적으로 구성한 경우:
  - 신경망의 파라미터가 많이 필요
- 어휘사전에 없는 단어: ‘unknown’ 토큰 (`UNK` 토큰)으로 매칭
  - 너무 많은 단어를 UNK로 두면 정보 손실 많음

2.2.3 부분단어 토큰화

- subword tokenization
  - 문자 토큰화와 단어 토큰화의 장점을 결합
- 드물게 등장하는 단어:
  - 더 작은 단위로 나눔
  - 복잡한 단어나 철자 오류를 처리
- WordPiece:
  - 텍스트를 서브워드(subword) 단위로 분할하는 토큰화 알고리즘 중 하나
  - 언어의 단어를 작은 부분으로 나누어 처리하는 데에 유용
  - 단어를 구성하는 부분(subword)을 찾아 나가는 bottom-up 방식을 사용
    - 초기에는 각 글자가 하나의 서브워드로 간주되며, 이후에는 자주 나타나는 서브워드 쌍을 합쳐가면서 텍스트를 효과적으로 토큰화
  - 자주 나타나는 서브워드를 우선 선택하는 빈도 기반의 접근을 취함
    - 자주 사용되는 어휘에 대해 높은 확률로 정확한 토큰화를 수행
  - 자주 등장하는 서브워드 쌍을 병합하면서 점진적으로 언어의 구조를 학습
    - 어휘를 효과적으로 확장하면서도 적은 수의 토큰으로 언어를 표현
- 토큰화 예시:
  - ‘##izing’, ‘##p’
  - #은 앞의 문자열이 공백이 아님을 의미함
  - 토큰을 문자열로 다시 바꿀 때 앞의 토큰과 합친다.

2.2.4 전체 데이터셋 토큰화하기

- DatasetDict 객체의 map() 메서드 사용

- [`PAD`] 토큰:
  - 길이를 맞추기 위해 끝에 동일한 길이만큼 `0`으로 추가
  - `0`은 어휘사전에 있는 [`PAD`] 토큰
- attention_mask 배열:
  - 추가된 패딩 토큰 때문에 모델이 혼동하지 않게 하려는 조치
  - 이를 통해 패딩된 부분을 무시함.