ByT5: Towards a token-free future with pre-trained byte-to-byte models

주세준

2021.7.14

ByT5: Towards a token-free future with pre-trained byte-to-byte models

- T5 -> mT5 -> ByT5

- Multilingual 문제에 대한 해법으로 utf-8 인코딩으로 training data를 처리하는 방법을 제시

ByT5: Towards a token-free future with pre-trained byte-to-byte models

Linting Xue* **Aditya Barua*** **Noah Constant*** **Rami Al-Rfou***
Sharan Narang **Mihir Kale** **Adam Roberts** **Colin Raffel**
Google Research

Abstract

Most widely-used pre-trained language models operate on sequences of tokens corresponding to word or subword units. Encoding text as a sequence of tokens requires a tokenizer, which is typically created as an independent artifact from the model. *Token-free* models that instead operate directly on raw text (bytes or characters) have many benefits: they can process text in any language out of the box, they are more robust to noise, and they minimize technical debt by removing complex and error-prone text preprocessing pipelines. Since byte or character sequences are longer than token sequences, past work on token-free models has often introduced new model architectures designed to amortize the cost of operating directly on raw text. In this paper, we show that a standard Transformer architecture can be used with minimal modifications to process byte sequences. We carefully characterize the trade-offs in terms of parameter count, training FLOPs, and inference speed, and show that byte-level models are competitive with their token-level counterparts. We also demonstrate that byte-level models are significantly more robust to noise and perform better on tasks that are sensitive to spelling and pronunciation. As part of our contribution, we release a new set of pre-trained byte-level Transformer models based on the T5 architecture, as well as all code and data used in our experiments.[1]

1 Introduction

Machine learning models for text-based natural language processing (NLP) tasks are trained to perform some type of inference on input text. An important consideration when designing such a model is the way that the text is represented. A historically common representation is to assign a unique *token* ID to each word in a finite, fixed vocabulary. A given piece of text is thus converted into a sequence of tokens by a *tokenizer* before being fed into a model for processing. An issue with using a fixed vocabulary of words is that there is no obvious way to process a piece of text that contains an *out-of-vocabulary* word. A standard approach is to map all unknown words to the same <UNK> token, which prevents the model from distinguishing between different out-of-vocabulary words.

Subword tokenizers (Sennrich et al., 2016; Wu et al., 2016; Kudo and Richardson, 2018) present an elegant solution to the out-of-vocabulary problem. Instead of mapping each word to a single token, subword tokenizers decompose words into smaller subword units with a goal of minimizing the total length of the token sequences for a fixed vocabulary size. As an example, a subword tokenizer might tokenize the word *doghouse* as the pair of tokens *dog* and *house* even if *doghouse* is not in the subword vocabulary. This flexibility has caused subword tokenizers to become the *de facto* way to tokenize text over the past few years.

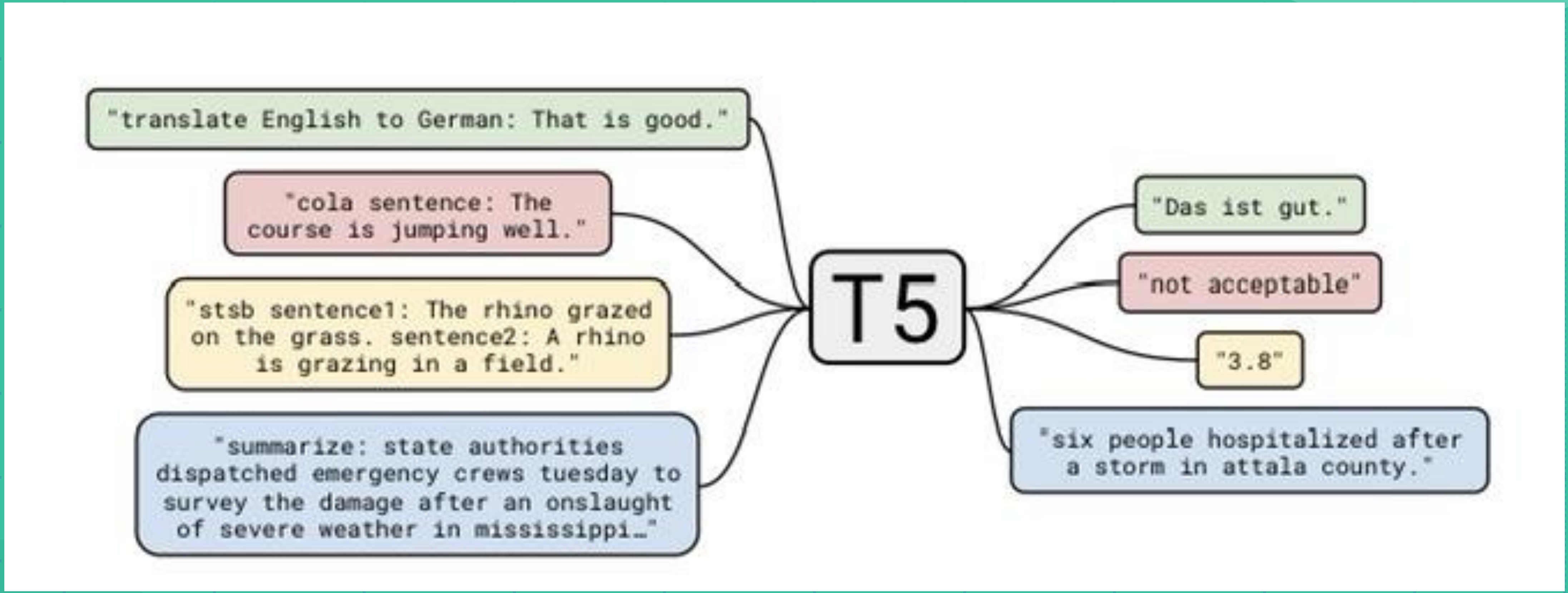
However, subword tokenizers still exhibit various undesirable behaviors. Typos, variants in spelling and capitalization, and morphological changes can all cause the token representation of a root word or phrase to change completely, which can result in the model making mispredictions. Furthermore, unknown characters (e.g. from a new language that was not used when the subword vocabulary was built) are still typically out-of-vocabulary for a subword model. While the *byte-level fallback* feature of tokenizers like SentencePiece (Kudo and Richardson, 2018) can allow for processing out-of-vocabulary characters, it nevertheless will typically result in only training the byte-level tokens' embeddings on a small fraction of the data.

A more natural solution that avoids the aforemen-

*Equal Contribution. Please direct correspondence to { lintingx, adityabarua, nconstant, rmyeid, sharannarang, mihirkale, adarob } @google.com, craffel@gmail.com

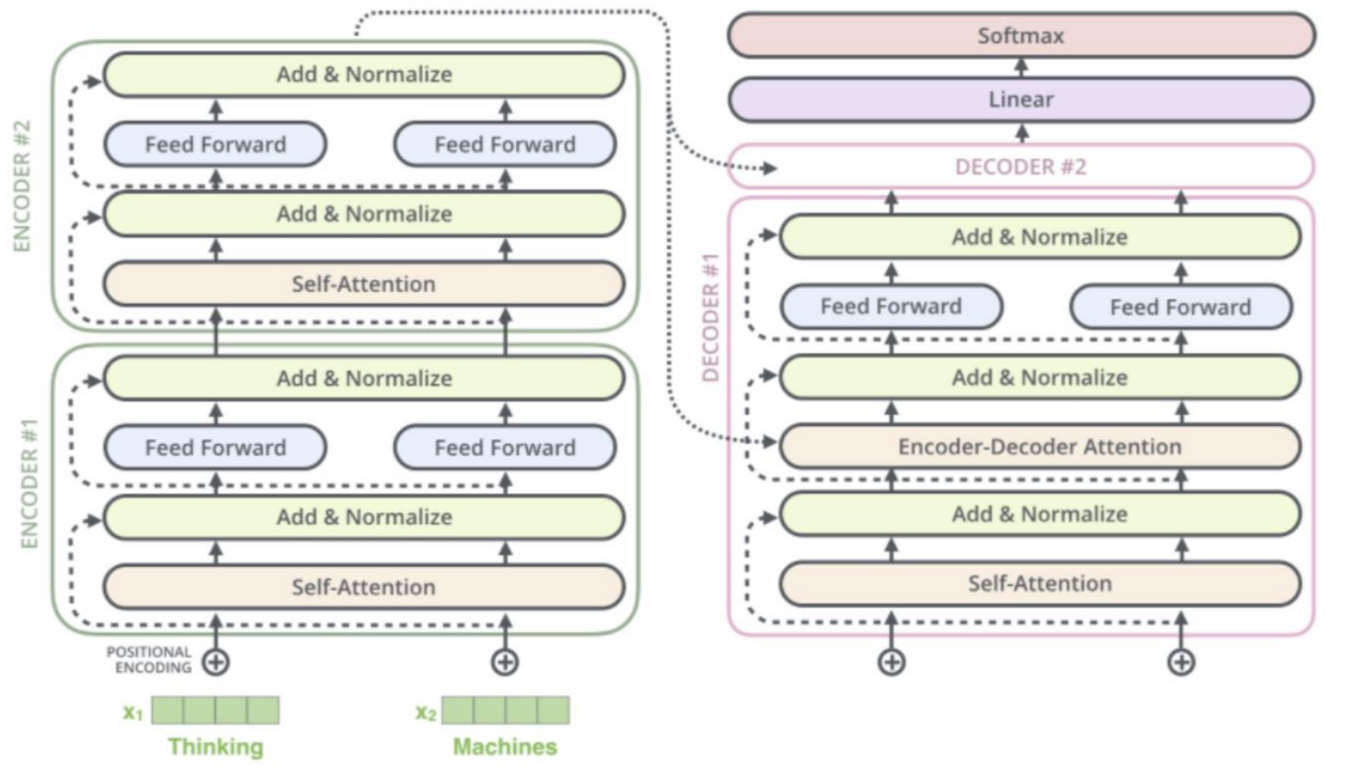
[1] https://github.com/google-research/byt5

T5



Input: text => output: text

T5/ architecture



Model structure. From: Jay Alammar's blog

일반적인 transformer 구조

SentencePiece tokenizer

mT5

- C4-> mC4

  C4: 기본적으로 영어만 포함하기 위해 생성된 데이터셋

  English terminal punctutation mark으로 filter

  mC4: 여러 언어를 포함하기 위해서 다른 filter 기준

  line length filter: 각줄에 200개 이상의 문자를 포함한 최소 3줄 이상

- T5-> mT5

  GeGLU nonlinearities 사용

  dmodel과 dff 모두 scaling 실행 (dff 만 하는 대신)

  pre-train 과정에서 no dropout

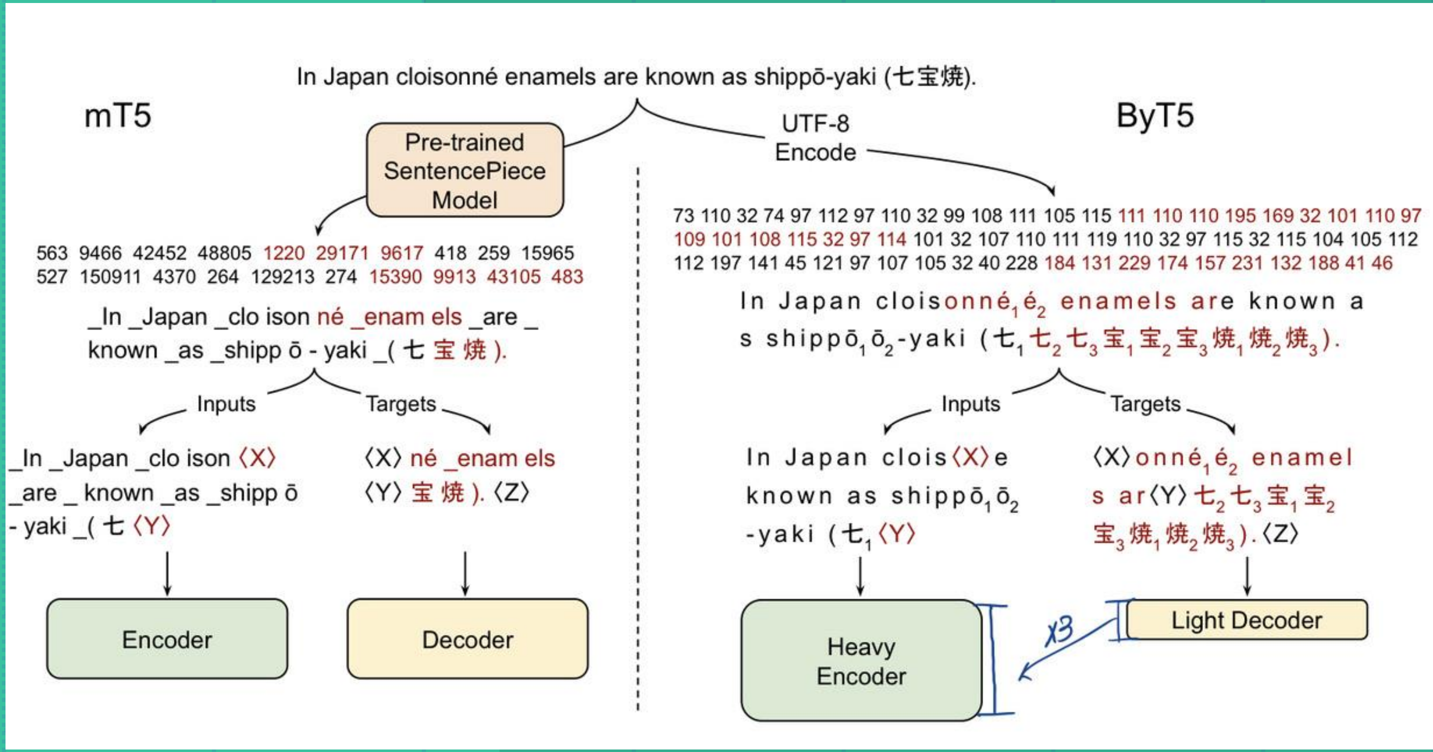
  SentencePiece's "byte-fallback": 중국어처럼 vocab size 큰 언어도 처리

ByT5

mT5: SentencePiece token

Byt5: UTF-8 bytes

-> UTF-8 byte directly 256 possible value+ 3 ID padding, end, <unk>



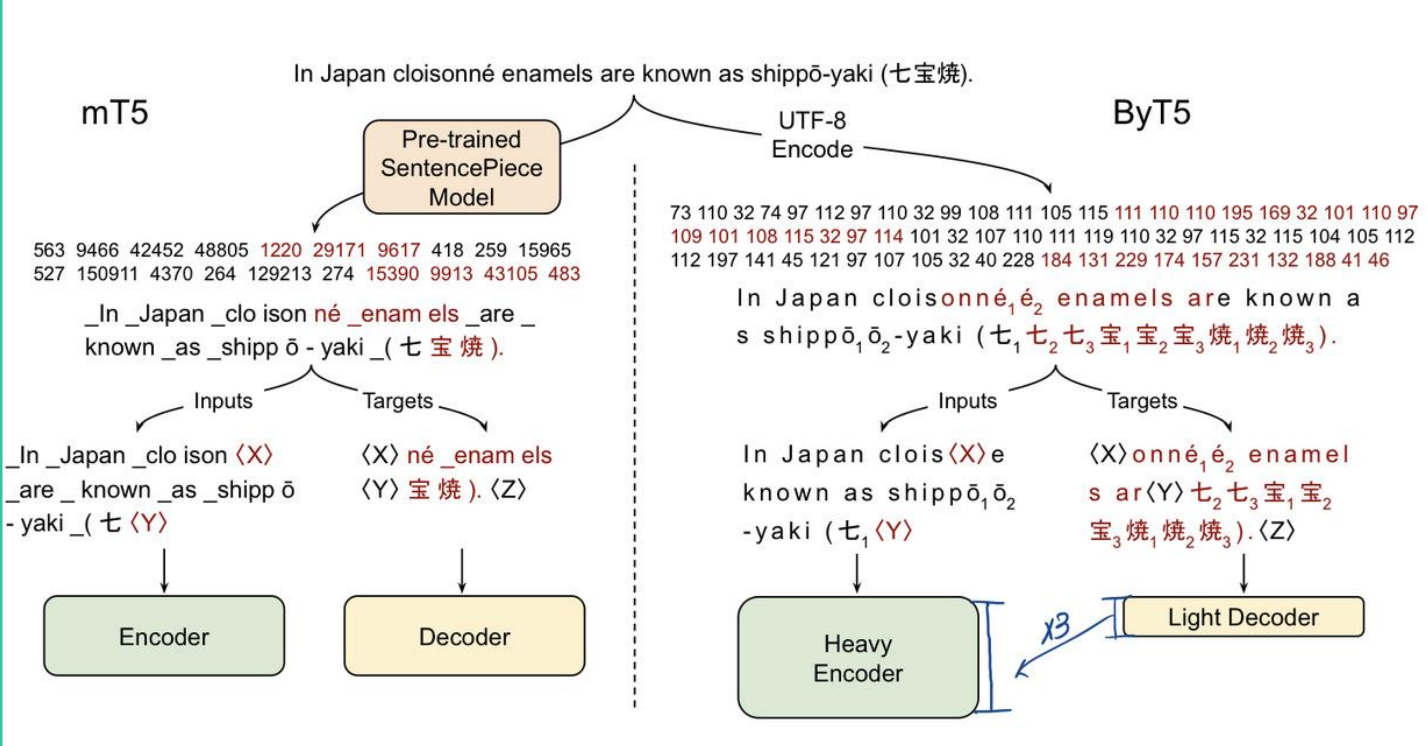
ByT5

mT5 pretraining method “span corruption”

Spans of tokens are replaced with single “sentinel” ID

mT5 average span of 3 subword token
-> longer better span length 20 bytes

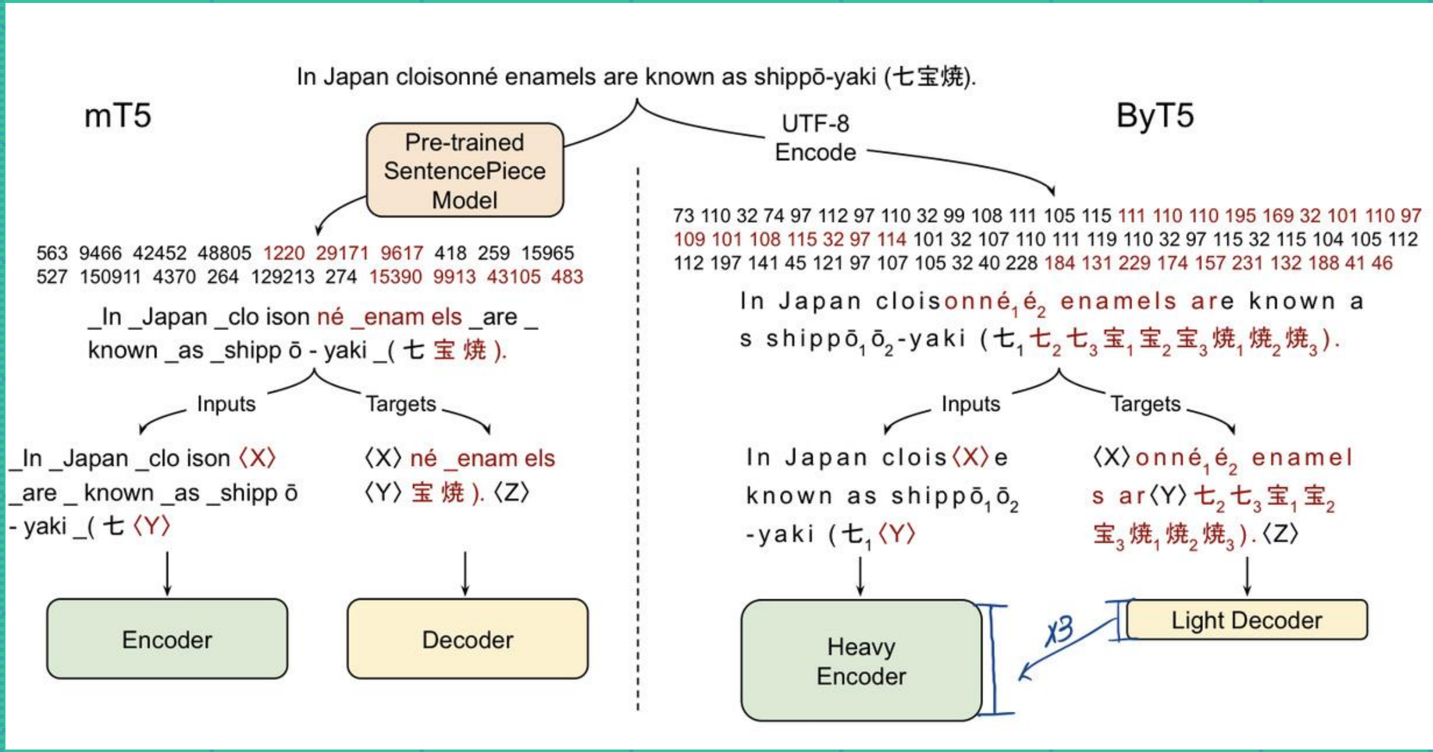
3tokens is too easy -> need longer
20 best for XLNI (classif)
40 best for 나머지 (gener)
longer span -> better on harder task



Model	XNLI (Accuracy)	TyDiQA-GoldP (F1)	GEM-XSum (BLEU)
ByT5-Large (1.23B)	**79.7**	87.7	11.5
mT5-Large (1.23B)	81.1	85.3	10.1
(a) ByT5-36/12-668M	78.3	87.8	12.3
(b) ByT5-24/24-718M	75.4	83.0	7.1
(c) ByT5-12/36-768M	73.5	83.1	8.3
(d) mT5-36/12-1.18B	81.5	87.1	10.8
(e) ByT5-Large-Span3	79.4	87.4	10.2
(f) ByT5-Large-Span40	78.9	**88.3**	**12.6**
(g) CharT5-36/12-1.23B	79.0	87.6	11.2

Table 8: Ablation results on XNLI zeroshot, TyDiQA-GoldP, and GEM-XSum.

ByT5



Decouple the depth of encoder and decoder

Encoder = decoder*3 (~BERT)

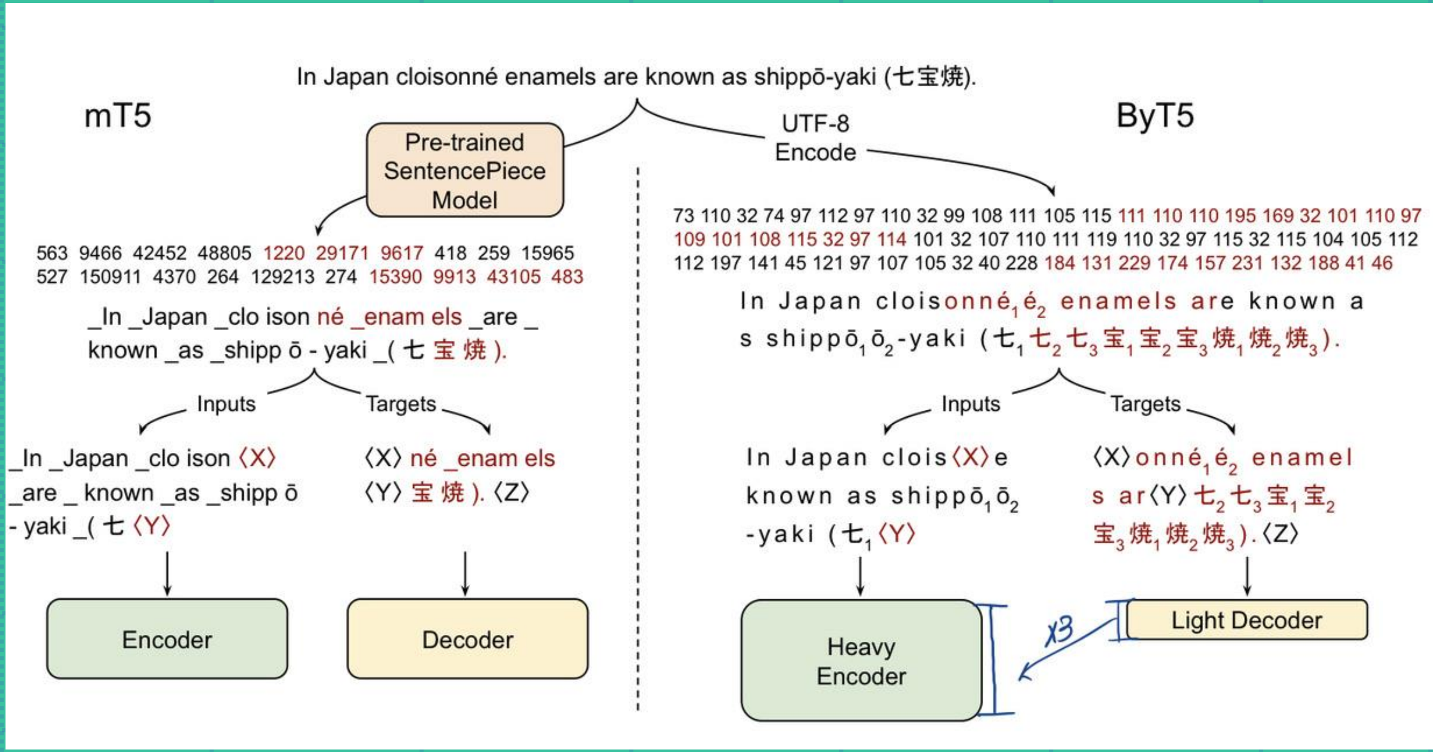
Maybe because the decoder is run autoregressively during interference

ByT5

Drop illegal bytes

-> seq len 1024 “tokens”(=bytes)

1million step, batches of 2^20 tokens



Best scenario for ByT5

- Model size under 1billion parameters
- Generative task (XLNI)
- Multilingual task with in-language labels
- In the presence of various types of noise

ByT5

#params < 1 billion shows significant performance

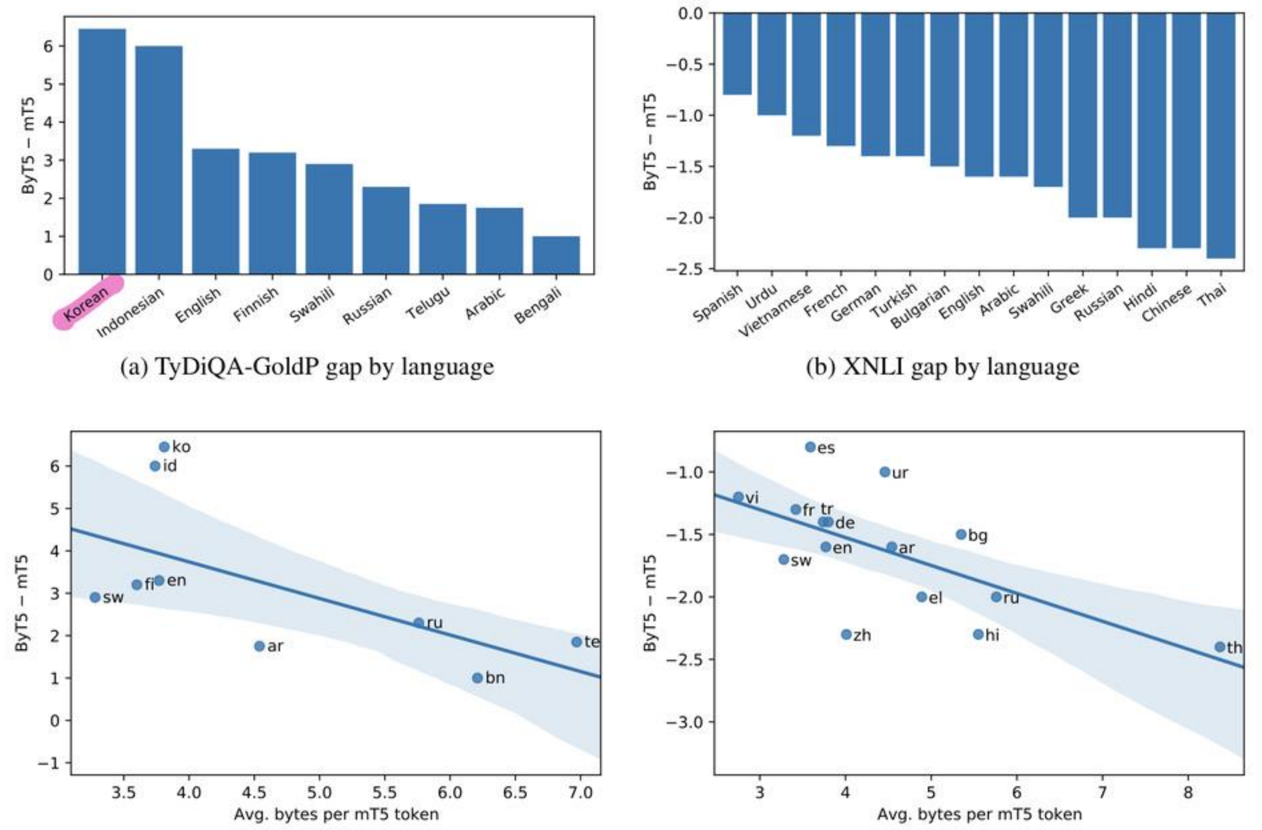
Multilingual-task

	Small		Base		Large		XL		XXL	
	mT5	ByT5	mT5	ByT5	mT5	ByT5	mT5	ByT5	mT5	ByT5
In-language multitask (models fine-tuned on gold data in all target languages)										
WikiAnn NER	86.4	**90.6**	88.2	**91.6**	89.7	**91.8**	91.3	**92.6**	92.2	**93.7**
TyDiQA-GoldP	74.0 / 62.7	**82.6 / 73.6**	79.7 / 68.4	**86.4 / 78.0**	85.3 / 75.3	**87.7 / 79.2**	87.6 / 78.4	**88.0 / 79.3**	88.7 / 79.5	**89.4 / 81.4**
Translate-train (models fine-tuned on English data plus translations in all target languages)										
XNLI	72.0	**76.6**	79.8	**79.9**	**84.4**	82.8	**85.3**	85.0	**87.1**	85.7
PAWS-X	79.9	**88.6**	89.3	**89.8**	**91.2**	90.6	**91.0**	90.5	91.5	**91.7**
XQuAD	64.3 / 49.5	**74.0 / 59.9**	75.3 / 59.7	**78.5 / 64.6**	81.2 / 65.9	**81.4 / 67.4**	82.7 / 68.1	**83.7 / 69.5**	**85.2 / 71.3**	84.1 / 70.2
MLQA	56.6 / 38.8	**67.5 / 49.9**	67.6 / 48.5	**71.9 / 54.1**	73.9 / 55.2	**74.4 / 56.1**	75.1 / 56.6	**75.9 / 57.7**	**76.9** / 58.3	**76.9 / 58.8**
TyDiQA-GoldP	49.8 / 35.6	**64.2 / 50.6**	66.4 / 51.0	**75.6 / 61.7**	75.7 / 60.1	**80.1 / 66.4**	80.1 / 65.0	**81.5 / 67.6**	**83.3** / 69.4	83.2 / **69.6**
Cross-lingual zero-shot transfer (models fine-tuned on English data only)										
XNLI	67.5	**69.1**	**75.4**	**75.4**	**81.1**	79.7	**82.9**	82.2	**85.0**	83.7
PAWS-X	82.4	**84.0**	**86.4**	86.3	**88.9**	87.4	**89.6**	88.6	90.0	**90.1**
WikiAnn NER	50.5	**57.6**	55.7	**62.0**	58.5	**62.9**	**65.5**	61.6	**69.2**	67.7
XQuAD	58.1 / 42.5	**66.3 / 49.7**	**67.0 / 49.0**	66.6 / 48.1	**77.8 / 61.5**	61.5 / 43.9	**79.5 / 63.6**	57.7 / 43.0	**82.5 / 66.8**	79.7 / 63.6
MLQA	54.6 / 37.1	**60.9 / 43.3**	64.4 / 45.0	**66.6 / 47.3**	**71.2 / 51.7**	65.6 / 45.0	**73.5 / 54.4**	65.1 / 46.5	**76.0 / 57.4**	71.6 / 54.9
TyDiQA-GoldP	36.4 / 24.4	**54.9 / 39.9**	59.1 / 42.4	**69.6 / 54.2**	68.4 / 50.9	**75.4 / 59.4**	**77.8 / 61.8**	63.2 / 49.2	**82.0 / 67.3**	75.3 / 60.0

ByT5

Multilingual-task

XLNI 왜 -??



(a) TyDiQA-GoldP gap by language

(b) XNLI gap by language

(c) TyDiQA-GoldP gap by compression

(d) XNLI gap by compression

Figure 3: Per-language performance gaps between ByT5-Large and mT5-Large. **Top**: (a) Gaps on TyDiQA-GoldP. (b) Gaps on XNLI zero-shot. **Bottom**: The same gaps as a function of each language's "compression rate".

ByT5

1. Drop
2. Add/Drop/Mutate
3. Repetition
4. Antspeak
5. Uppercase
6. Random Case

We experiment with six different noising schemes: (1) **Drop**: Each character has a 10% chance of being dropped. (2) **Add/Drop/Mutate**: At each character position, there is a 10% chance of applying one of three actions, with equal likelihood: *Add* (inserts a random character from the input), *Drop* (deletes this character) or *Mutate* (replaces this character with a random character from the input). (3) **Repetitions**: Each character has a 20% chance of being selected for repetition. If selected, 1–3 repetitions (with equal likelihood) are appended after the original character. (4) **Antspeak**: Each character is capitalized and padded with spaces. For example, "abc def" becomes " A B C D E F ". (5) **Uppercase**: Each character is converted to uppercase. Here, we restrict to languages whose scripts distinguish case (for XNLI: Bulgarian, English, French, German, Greek, Russian, Spanish, Swahili, Turkish, Vietnamese; for TyDiQA-GoldP: English, Finnish, Indonesian, Russian, Swahili). (6) **Random case**: Each character is set to a random case (upper or lower). Again, only languages whose scripts distinguish case are considered.

		Learnable Noise		**Unseen Noise**
	Model	XNLI (accuracy)	TyDiQA-GoldP (F1)	XNLI (accuracy)
Clean	mT5	81.1	85.3	81.1
	ByT5	79.7	87.7	79.7
Drop	mT5	−10.2	−19.9	−18.3
	ByT5	**−8.2**	**−18.4**	**−11.4**
Add/Drop/Mutate	mT5	−9.2	−28.5	−11.4
	ByT5	**−8.0**	**−24.3**	**−10.9**
Repetitions	mT5	−8.5	−11.0	−12.3
	ByT5	**−4.1**	**−3.1**	**−5.9**
Antspeak	mT5	−32.0	−17.5	−34.4
	ByT5	**−8.7**	**−4.3**	**−24.4**
Uppercase	mT5	−7.0	−7.6	−8.1
	ByT5	**−1.5**	**−1.0**	**−1.7**
Random Case	mT5	−25.7	−13.9	−19.2
	ByT5	**−1.5**	**−1.2**	**−5.9**

ByT5

Generative task

	GEM-XSum		TweetQA	
Model	mT5	ByT5	mT5	ByT5
Small	6.9	**9.1**	54.4 / 58.3	**65.7 / 69.7**
Base	8.4	**11.1**	61.3 / 65.1	**68.7 / 72.2**
Large	10.1	**11.5**	67.9 / 72.0	**70.0 / 73.6**
XL	11.9	**12.4**	68.8 / 72.4	**70.6 / 74.7**
XXL	14.3	**15.3**	70.8 / 74.3	**72.0 / 75.7**

Tokenizer

word tokenizer(transformerXL, 267735)

Subword tokenizer (BERT, 30000)

byte pair encoding(GPT, roBERTa, 50000)

sentencepiece(XLNet, T5, 30000)

Word tokenizer

TransformerXL

-> 공백 구두점을 기준으로 tokenize

Dictionary 크기가 엄청 커진다는 단점

```
["Do", "n't", "you", "love", "Transformers", "?", "We", "sure", "do", "."]
```

Subword Tokenizer

BERT

의미있는 단어 혹은 subword 표현을 학습하면서도, 합리적인 dictionary size 유지.

unk 처리를 효과적으로 할 수 있다

```
from transformers import BertTokenizer
 tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
tokenizer.tokenize("I have a new GPU!")
>>> ["i", "have", "a", "new", "gp", "##u", "!"]
```

기본 사전은 [b , g , h , n , p , s , u]

Byte-Pair Encoding

GPT, roBERTa

우선 training data를 단어 단위로 분절하는

pre-tokenize 과정 거쳐야 됨(자유롭게)

1. 기본 사전은 ['b', 'g', 'h', 'n', 'p', 's', 'u']
2. 가장 많이 등장한 캐릭터 쌍이 무엇인지 살펴봅니다. “hu”는 총 15번, “ug”는 총 20번이 나와 가장 많이 등장한 쌍은 “ug”가 됩니다. 따라서 “u”와 “g”를 합친 “ug” “ug”를 사전에 새로이 추가합니다.
3. 다음에 가장 많이 나온 쌍은 16번 등장한 “un”이므로, “un”을 사전에 추가해줍니다. 그 다음은 15번 등장한 “hug”이므로 “hug”도 사전에 추가해줍니다.

이처럼 계속 훈련을 통해 학습 배운 규칙으로 분절 진행

Dic size (기본 단어 개수+ # 합쳐진 subword) = hyperparameter

```
('hug', 10), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)
```

```
('h' 'u' 'g', 10), ('p' 'u' 'g', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'u' 'g' 's', 5)
```

```
('h' 'ug', 10), ('p' 'ug', 5), ('p' 'u' 'n', 12), ('b' 'u' 'n', 4), ('h' 'ug' 's', 5)
```

```
('hug', 10), ('p' 'ug', 5), ('p' 'un', 12), ('b' 'un', 4), ('hug' 's', 5)
```

SentencePiece XLNetTokenizer

```
from transformers import XLNetTokenizer tokenizer = XLNetTokenizer.from_pretrained("xlnet-base-cased") tokenizer.tokenize(" Don't you love transformers? We sure do.") >>>
 ["▁Don", "'", "t", "▁you", "▁love", "▁", "Transform", "ers", "?", "▁We", "▁sure", "▁do", "."]
```

입력 문장을 Raw Stream으로 취급해 공백을 포함한 모든 캐릭터를 활용해, BPE 혹은 Unigram을 적용하며 사전을 구축

_ 가 공백을 나타냄

단순히 모든 토큰들을 붙여준 후, "_" 캐릭터만 공백으로 바꿔주면 되기 때문에 decode 작업이 매우 쉬워짐

Thank you

- Further study
- Unigram (subword tokenizer algorithm/ Sentencepiece)
- Wordpiece(subword tokenizer algorithm)