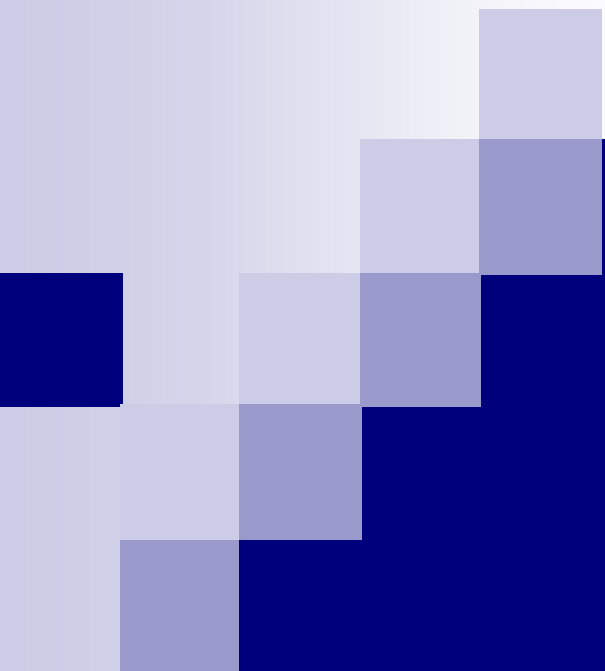


**June 21 2022**

**2022 Samsung AI 교육과정**



# **Text Generation using RNN: Coding Exercise**

**Kyomin Jung**

**Department of Electrical and Computer Engineering  
Seoul National University**

# About Today Class

## ■ Today's TA

- 장윤아 (vn2209@snu.ac.kr)
- 이동렬 (drl123@snu.ac.kr)

# Contents

## ■ Text generation using **RNN & Attention**

### □ Practice

- Load/preprocess data
- Build Seq2Seq with attention model
  - Encoder
  - Attention
  - Decoder
  - Seq2Seq
- Measure with BLEU

## ■ Subword Tokenization

- BPE (Byte Pair Encoding)
- WordPiece
- SentencePiece

# Data

- **Multi30k Dataset**
- **31014개** 사진에 대해 설명하는 같은 뜻의 영어/독일어 문장 **pair**
- **Ex)**
  - (En) Brick layers constructing a wall.
  - (De) Maurer bauen eine Wand.



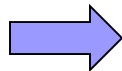
- **Simple way of loading from PyTorch (using torchtext library):**

```
from torchtext.legacy.datasets import Multi30k

SRC = Field(tokenize = tokenize_de, init_token = '<sos>', eos_token = '<eos>',
            lower = True)

TRG = Field(tokenize = tokenize_en, init_token = '<sos>', eos_token = '<eos>',
            lower = True)

train_data, valid_data, test_data = Multi30k.splits(exts = ('.en', '.de'), fields =
            (SRC, TRG))
```



Create Source and Target Tokenizer

```
[3] spacy_de = spacy.load('de_core_news_sm')
    spacy_en = spacy.load('en_core_web_sm')

[4] tokenizer_de = get_tokenizer('spacy', language='de_core_news_sm')
    tokenizer_en = get_tokenizer('spacy', language='en_core_web_sm')

[5] tokenizer_en("he was bored")

['he', 'was', 'bored']
```

# Data

## ■ Tokenizer

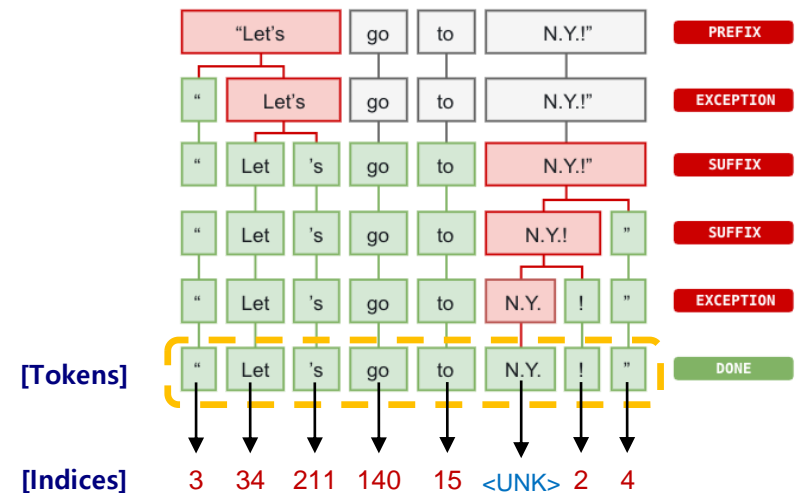
- 언어별로 다른 **tokenizer** 적용

### Create Source and Target Tokenizer

```
[3] spacy_de= spacy.load('de_core_news_sm')  
    spacy_en = spacy.load('en_core_web_sm')
```

```
[4] tokenizer_de= get_tokenizer('spacy', language='de_core_news_sm')  
    tokenizer_en= get_tokenizer('spacy', language='en_core_web_sm')
```

```
[5] tokenizer_en("he was bored")  
  
['he', 'was', 'bored']
```



## ■ Special Tokens

- <BOS>, <EOS>: 문장의 시작, 끝을 나타내는 **token**
- <PAD>
- <UNK>: OOV 대응하기 위해 사용

# Building Vocabulary

## ■ 언어별로 vocabulary 각각 생성 – from train dataset

```
from collections import Counter
from torchtext.vocab import vocab

counter_de = Counter()
counter_en = Counter()
for i in train_data:
    counter_en.update((i['src']))
    counter_de.update((i['trg']))
vocab_de = vocab(counter_de, min_freq=1, specials=('<unk>', '<BOS>', '<EOS>', '<PAD>'))
vocab_en = vocab(counter_en, min_freq=1, specials=('<unk>', '<BOS>', '<EOS>', '<PAD>'))
vocab_de.set_default_index(vocab_de['<unk>'])
vocab_en.set_default_index(vocab_en['<unk>'])
```

## ■ Torchtext.vocab.vocab

```
class torchtext.vocab.Vocab(counter, max_size=None, min_freq=1, specials=['<pad>'],
                             vectors=None, unk_init=None, vectors_cache=None, specials_first=True)
```

## ■ collections.Counter

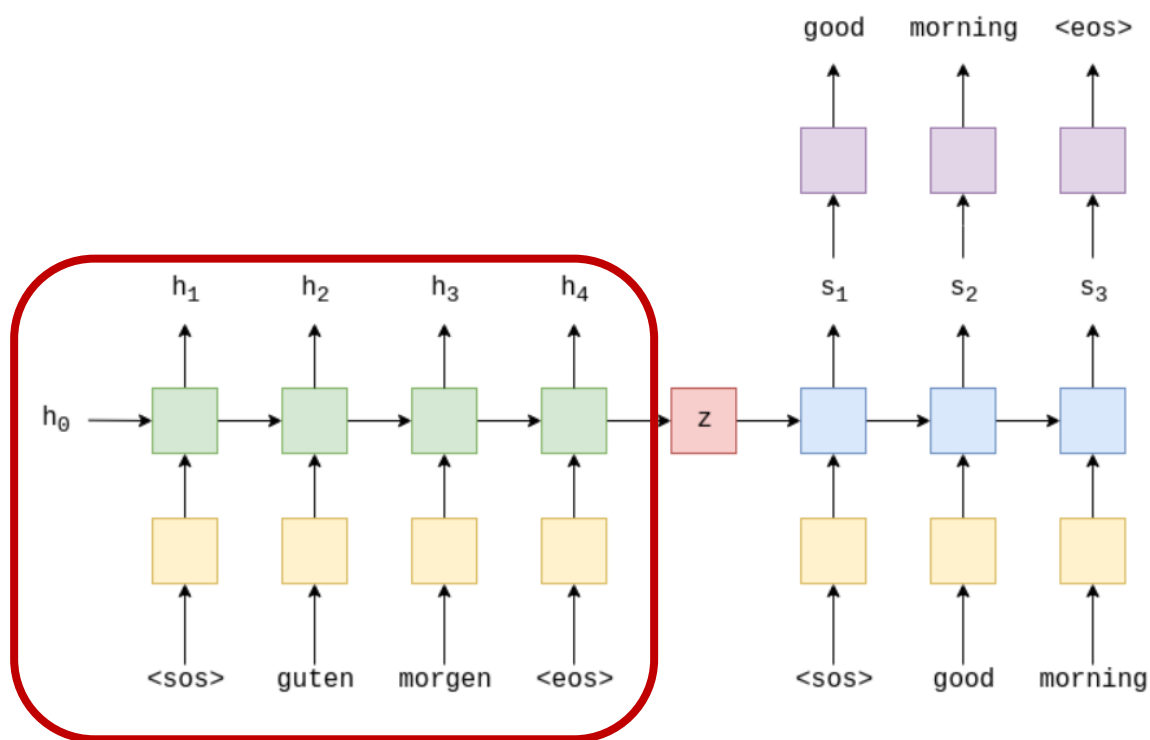
- 동일한 값의 자료가 몇 개인지 파악하는데 사용하는 객체

```
from collections import Counter
Counter('hello world')
# Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

# Model - Encoder

## ■ Encoder

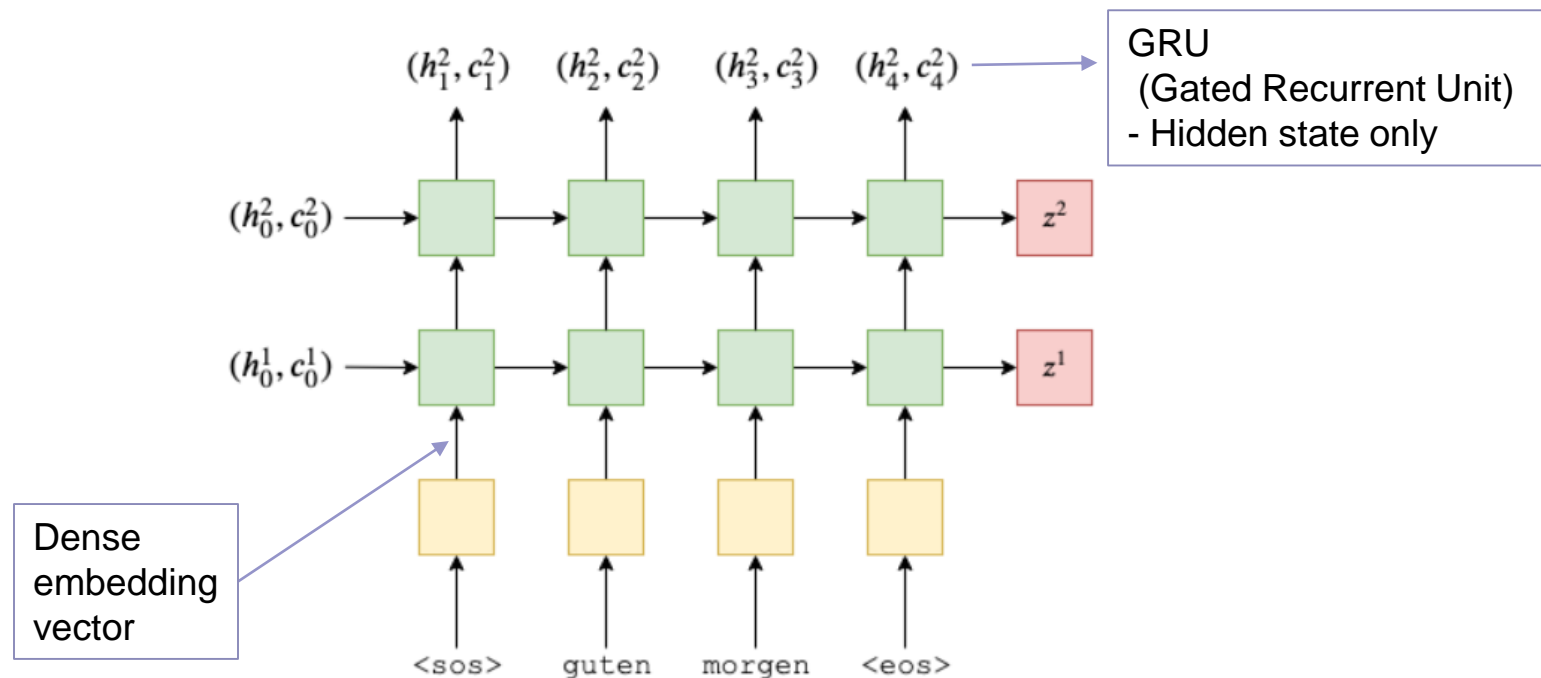
- Embedding Layer + bidirectional RNN layer



# Model - Encoder

## ■ Encoder

- Embedding Layer + bidirectional RNN layer





# Model - Encoder

```
class Encoder(nn.Module):
    def __init__(self, input_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.GRU(emb_dim, enc_hid_dim, bidirectional = True)
        self.fc = nn.Linear(enc_hid_dim * 2, dec_hid_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):

        embedded = self.dropout(self.embedding(src))

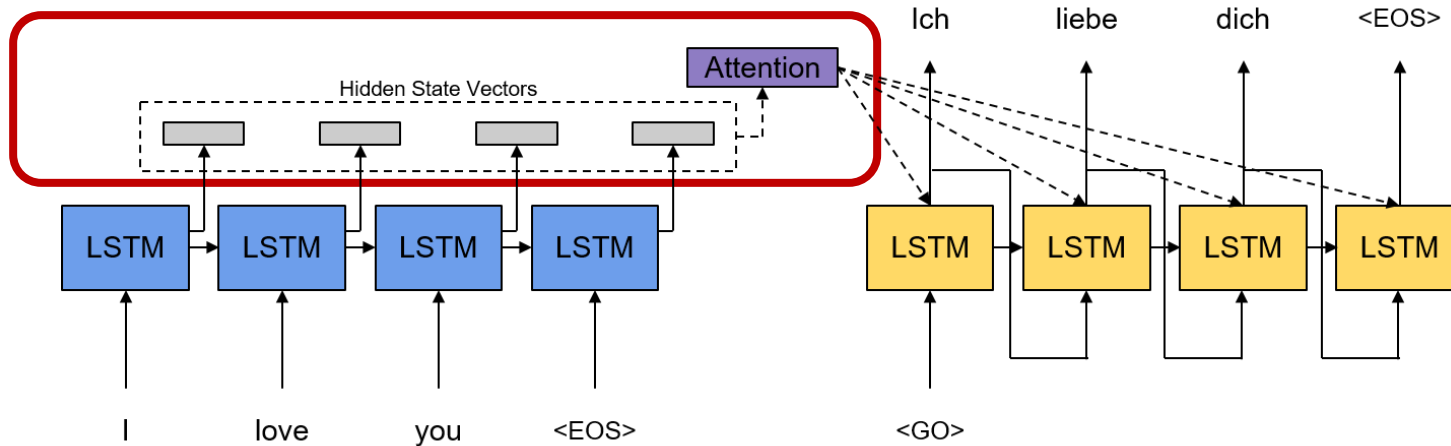
        outputs, hidden = self.rnn(embedded)
        hidden = torch.tanh(self.fc(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)))

        return outputs, hidden
```

```
src = [src_len, bs]
embedded = [src_len, bs, emb_dim]
outputs = [src_len, bs, enc_hid_dim*2]
hidden = [2, bs, enc_hid_dim]
```

# Model - Attention

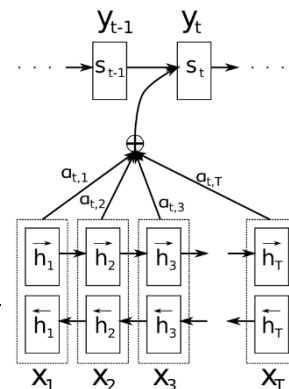
## ■ Attention



- Fixed source representation is suboptimal  
→ Attention: Focus on different parts of the input

# Model - Attention

$$\text{score}(s_{i-1}, h_j) = v_a^\top \tanh(W_a[s_{i-1}; h_j])$$



```
class Attention(nn.Module):
    def __init__(self, enc_hid_dim, dec_hid_dim):
        super().__init__()

        self.attn = nn.Linear((enc_hid_dim * 2) + dec_hid_dim, dec_hid_dim)
        self.v = nn.Linear(dec_hid_dim, 1, bias = False)

    def forward(self, hidden, encoder_outputs):
        batch_size = encoder_outputs.shape[1]
        src_len = encoder_outputs.shape[0]
        hidden = hidden.unsqueeze(1).repeat(1, src_len, 1)
        encoder_outputs = encoder_outputs.permute(1, 0, 2)

        energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim = 2)))

        Score → attention = self.v(energy)
        attention = attention.squeeze(2)

        return F.softmax(attention, dim=1)
```

```
hidden.unsqueeze: [bs, 1, dec_hid_dim]
hidden.repeat: [bs, src_len, dec_hid_dim]
encoder_outputs = [src_len, bs, enc_hid_dim*2]
→ [bs, src_len, enc_hid_dim*2]
```

**Normalization**

# Model - Decoder

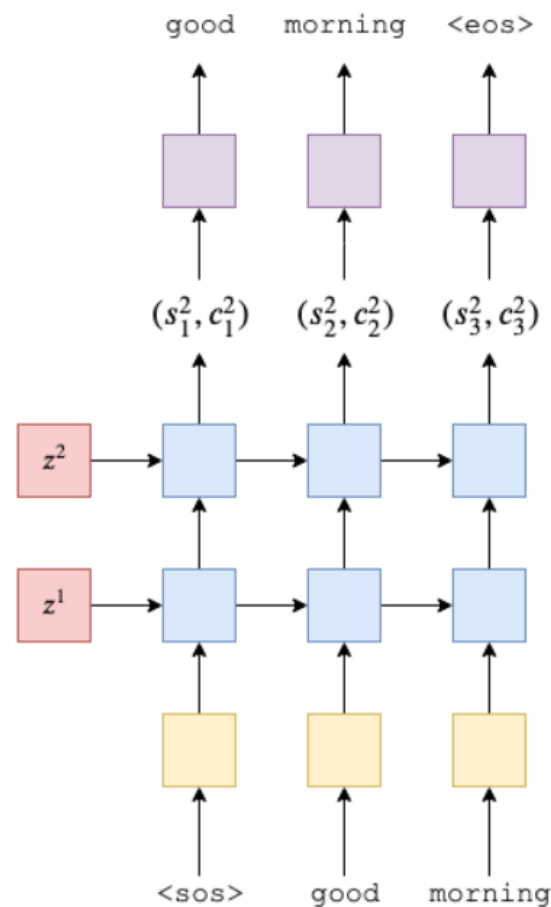
## ■ Decoder

- previous hidden state + attention + new input  
→ output

- Embedding Layer

- + RNN layer

- + Fully Connected layer



# Model - Decoder

```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, enc_hid_dim, dec_hid_dim, dropout, attention):
        super().__init__()

        self.output_dim = output_dim
        self.attention = attention
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.GRU((enc_hid_dim * 2) + emb_dim, dec_hid_dim)
        self.fc_out = nn.Linear((enc_hid_dim * 2) + dec_hid_dim + emb_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, encoder_outputs):
        input = input.unsqueeze(0)

        embedded = self.dropout(self.embedding(input))
        a = self.attention(hidden, encoder_outputs)
        ...
        weighted = torch.bmm(a, encoder_outputs)
        weighted = weighted.permute(1, 0, 2)
        rnn_input = torch.cat((embedded, weighted), dim = 2)
        output, hidden = self.rnn(rnn_input, hidden.unsqueeze(0))
        ...
        prediction = self.fc_out(torch.cat((output, weighted, embedded), dim = 1))

        return prediction, hidden.squeeze(0)
```

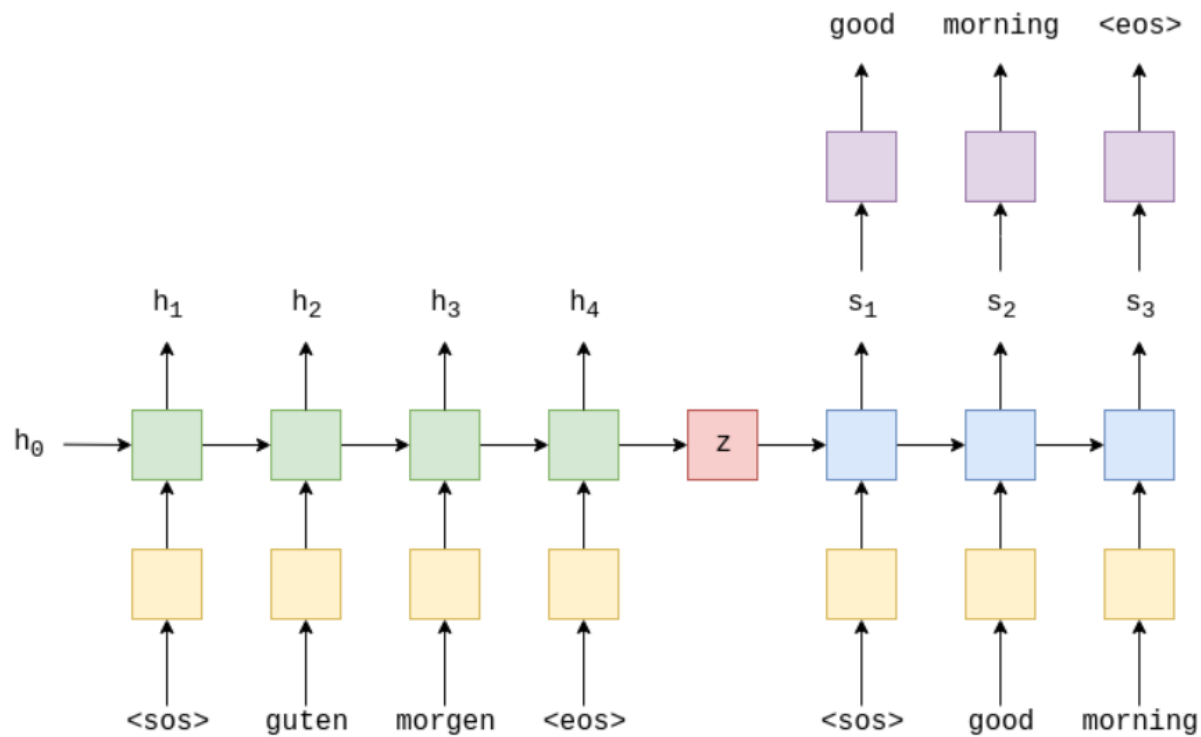
<bmm: 행렬 곱>

```
a = [bs, 1, src_len]
encoder_outputs = [bs, src_len, enc_hid_dim*2]
weighted = [bs, 1, enc_hid_dim*2]
```

```
embedded = [1, bs, emb_dim]
weighted = [1, bs, dec_hid_dim*2]
rnn_input = [1, bs, emb_dim+dec_hid_dim*2]
hidden = [bs, dec_hid_dim]
        → [1, bs, dec_hid_dim]
```

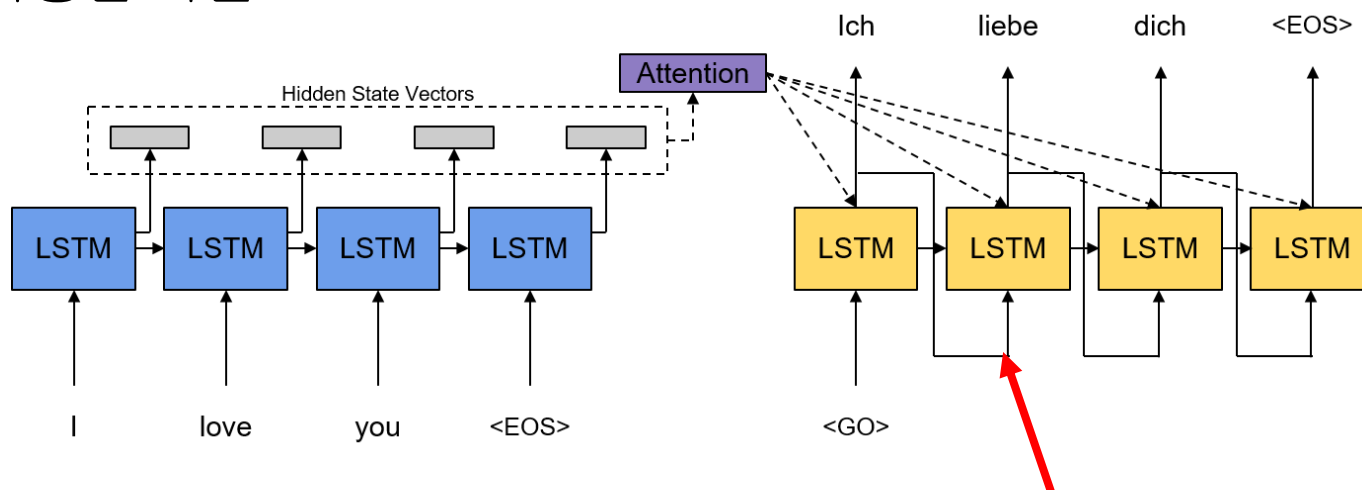
# Model - Seq2Seq

- Encapsulate Encoder-Attention-Decoder



# Model - Seq2Seq

- **Decoder**에서 앞쪽 단어들에 대한 예측이 틀릴 경우, 뒤쪽 단어들에  
게 큰 영향
- **Training** 과정에서 수렴 속도를 느리게 하며, 모델을 불안정하게 만  
들 수 있음
- **Teacher forcing ratio**
  - **training** 과정에서, 앞에서 예측한 단어 대신 정답 단어를 입력으  
로 사용할 확률



# Model - Seq2Seq

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):
        batch_size = src.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)

        encoder_outputs, hidden = self.encoder(src)
        input = trg[0,:]

        for t in range(1, trg_len):
            output, hidden = self.decoder(input, hidden, encoder_outputs)
            outputs[t] = output

            teacher_force = random.random() < teacher_forcing_ratio

            top1 = output.argmax(1)

            input = trg[t] if teacher_force else top1

        return outputs
```

**True word**      **Predicted word**



# Model

## ■ Initialize & Train

```
INPUT_DIM = len(vocab_en)
...
DEC_DROPOUT = 0.5

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM, N_LAYERS, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, HID_DIM, N_LAYERS, DEC_DROPOUT)

model = Seq2Seq(enc, dec, device).to(device)

optimizer = optim.Adam(model.parameters())
TRG_PAD_IDX = new_stoi2['<PAD>']
criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)

def train(model, iterator, optimizer, criterion, clip):
    model.train()
    epoch_loss = 0
    for src, trg in iterator:
        optimizer.zero_grad()
        output = model(src, trg)
        ...
    return epoch_loss/len(iterator)
```

# Measure

	1	2	3	4	5
Reference	학생이	인공지능에	관한	수업을	듣습니다
Candidate1	인공지능에	관한	수업을	듣습니다	학생이
Candidate2	학생이	화학에	관한	수업을	합니다

- 학습된 모델의 **text generation** 성능 평가 방법
  - **cross-entropy loss**는 실제 번역 성능과 완벽히 일치하지 않음
  - **cross-entropy loss**는 위의 예제에서 **candidate 2**를 더 높이 평가할 수 있음
  - 문장 유사도를 측정하기 위한 다른 방법 필요

# Measure

- $n - \text{gram Precision} = \frac{\text{일치하는 } n\text{-gram 수}}{\text{전체 } n\text{-gram 수}}$

- **Clipping**

- 중복된 **n-gram**은 **true**에 있는 **count**가 최댓값
- **Ex) True sentence**에 **apple**이 **2** 번 있지만 **predicted sentence**에는 **3**번 들어가 있을 경우 **2**번만 인정

- **Brevity Penalty**

- $$\text{BP} = \min \left( 1, \exp \left( 1 - \frac{\text{output length}}{\text{reference length}} \right) \right)$$

- **BLEU Score**

$$\text{BLEU} = \text{BP} \times \prod_{n=1}^4 (\text{n-gram Precision})^{1/4}$$

# Measure

$$\text{BLEU} = \text{BP} \times \prod_{n=1}^4 (\text{n-gram Precision})^{1/4}$$

## ■ Reference output

- 학생이 인공지능에 관한 수업을 듣습니다

## ■ Candidate output

- 어린이 학생이 인공지능에 관한 수업을 수업을 합니다

- $1\text{-gram Precision} = \frac{4}{7}$  [학생이, 인공지능에, 관한, 수업을]

- $2\text{-gram Precision} = \frac{3}{6}$  [(학생이 인공지능에), (인공지능에 관한), (관한 수업을)]

- $3\text{-gram Precision} = \frac{2}{5}$  [(학생이 인공지능에 관한), (인공지능에 관한 수업을)]

- $4\text{-gram Precision} = \frac{1}{4}$  [(학생이 인공지능에 관한 수업을)]

- $\text{BP} = \min\left(1, \exp\left(1 - \frac{7}{5}\right)\right) = 1$

- **BLUE = 0.393**

# Measure

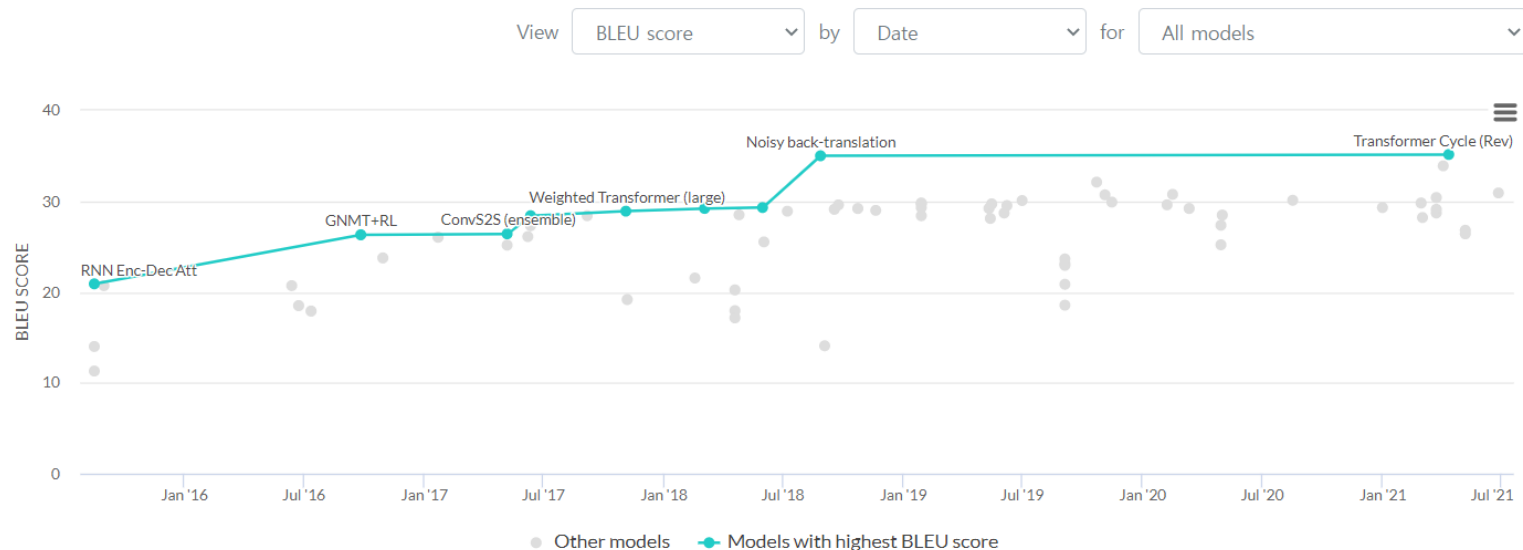
## ■ `Torchtext.data.metrics.bleu_score`

(`candidate_corpus`, `references_corpus`, `max_n`, ...)

- `candidate_corpus` = 점수를 측정할 후보 문장 모음
- `references_corpus` = 참조할 정답 문장 모음
- `max_n` = 계산할 최대 `n-gram`

Leaderboard

Dataset



# Contents

- Text generation using RNN & Attention

- Practice

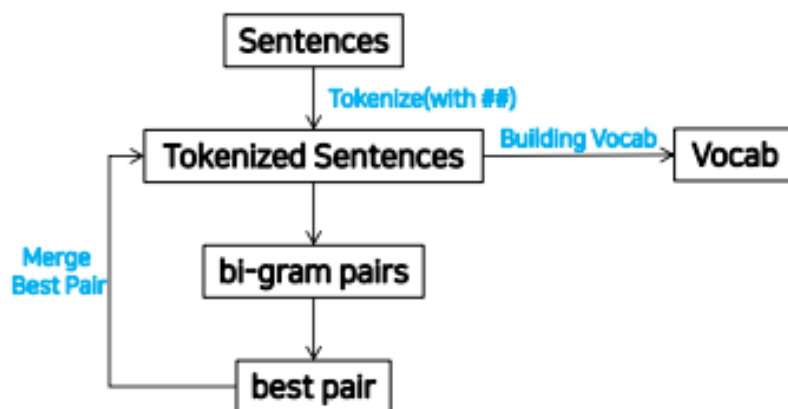
- Load/preprocess data
    - Build Seq2Seq with attention model
      - Encoder
      - Attention
      - Decoder
      - Seq2Seq
    - Measure with BLEU

- Subword Tokenization

- BPE (Byte Pair Encoding)
  - WordPiece
  - SentencePiece

# BPE (Byte Pair Encoding)

- BERT 기반 모델들에 쓰이는 **subword tokenization** 기법
- OOV(Out-of-Vocabulary) 문제 해결
- **Idea:** 의미 있는 단위로 단어를 자르자
  - 의미 있는 패턴 (**subword**)는 자주 등장
  - 반복되는 패턴은 의미를 가지는 **subword**일 확률이 높음.



Merge Best Pair  
- 가장 많이 등장하는  
bi-gram pair를 찾아 merge

# BPE (Byte Pair Encoding)

## ■ Example: aaabdaabac

- 1) Vocab: (a, b, c, d) → character 단위
- Tokenize: (a a a b d a a a b a c)
- Bi-gram pairs: (aa aa ab bd da aa aa ab ba ac)
  
- 2) Vocab: (a, b, c, d, aa)
- Tokenizing: (aa a b d aa a b a c) = (X a b d X a b a c)
- Bi-gram pairs: (Xa ab bd dX Xa ab ba ac)
  
- 3) Vocab: (a, b, c, d, aa, ab)
- Tokenizing: (aa ab d aa ab a c) = (X Y d X Y a c)
- Bi-gram pairs: (XY Yd dX XY Ya ac)
  
- 4) Vocab: (a, b, c, d, aa, ab, aaab)



# WordPiece

- BPE의 변형 알고리즘

- BPE: 빈도수가 가장 높은 쌍을 병합

- WPM (Wordpiece Model)

- Corpus의 likelihood 를 가장 높이는 쌍으로 결합
  - $P(c1, c2)/P(c1)P(c2)$
  - WPM을 수행하기 이전의 문장:
    - Jet makers feud over seat width with big orders at stake
  - WPM을 수행한 결과(wordpieces):
    - \_J et \_makers \_fe ud \_over \_seat \_width \_with \_big \_orders \_at \_stake

- BERT, DistilBERT, Electra

# SentencePiece

- 사전 토큰화 작업없이 단어 분리 토큰화 수행
  - 중국어, 일본어, 한국어 → **pre-tokenization** 까다로움
  - **End-to-end system, Use raw text**
  - **Google**에서 공개
- **Normalizer, Trainer, encoder, Decoder**
  - **Normalizer**: 의미적으로 동일한 문자들 표준형식으로 정규화
  - **Trainer**: 서브 워드 분절 모델 학습
  - **Encoder**: 학습된 모델을 이용하여 서브워드 나열로 분절
  - **Decoder**: 서브 워드의 나열을 문장형태로 변환
- **ALBERT, XLNet, T5**