


# DNN\_03\_3. Seq2Seq & Attention

🕒 생성일	@2022년 6월 10일 오후 1:50
📁 유형	머신러닝/딥러닝
👤 작성자	 동훈 오

논문 리뷰

[NLP 논문 리뷰] Sequence To Sequence Learning With Neural Networks (Seq2Seq)  
Hansu Kim's Development Blog

<https://cpm0722.github.io/paper-review/sequence-to-sequence-learning-with-neural-networks>



## TensorFlow tutorial - Neural machine translation with Attention

텐서플로우에서 제공하는 seq2seq 모델 연습 튜토리얼이다. 해당 프로젝트는 스페인어를 영어로 기계 번역 하는 것을 주제로 하고, 어텐션 기반의 seq2seq 모델을 사용한다. 서브클래싱 방식으로 keras.Model, keras.layers 를 커스텀한다. 이 튜토리얼의 목적은 어텐션 메커니즘에 대해 한층 깊게 이해하는 것에 있다.

Neural machine translation with attention | Text | TensorFlow

This notebook trains a sequence to sequence (seq2seq) model for Spanish to English translation based on Effective Approaches to Attention-based Neural Machine Translation. This is an advanced example that assumes some knowledge of: Sequence to sequence models TensorFlow fundamentals below the keras layer:

 [https://www.tensorflow.org/text/tutorials/nmt\\_with\\_attention](https://www.tensorflow.org/text/tutorials/nmt_with_attention)



### set up

```
# 코랩 환경에서는 !pip
!pip install "tensorflow-text==2.8.*"

import typing
from typing import Any, Tuple    #type checking

import tensorflow as tf

import tensorflow_text as tf_text

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

### shape checker

저수준 api 로 작업하기 때문에 정보의 shape 에 대해 오류가 날 가능성이 많다. shape checker 를 미리 작성해서 계속 활용할 계획.

```
class ShapeChecker():
    def __init__(self):
```

```

# Keep a cache of every axis-name seen
self.shapes = {}

def __call__(self, tensor, names, broadcast=False):
    if not tf.executing_eagerly():
        return

    if isinstance(names, str):
        names = (names,)

    shape = tf.shape(tensor)
    rank = tf.rank(tensor)

    if rank != len(names):
        raise ValueError(f'Rank mismatch:\n'
                        f'    found {rank}: {shape.numpy()}\n'
                        f'    expected {len(names)}: {names}\n')

    for i, name in enumerate(names):
        if isinstance(name, int):
            old_dim = name
        else:
            old_dim = self.shapes.get(name, None)
        new_dim = shape[i]

        if (broadcast and new_dim == 1):
            continue

        if old_dim is None:
            # If the axis name is new, add its length to the cache.
            self.shapes[name] = new_dim
            continue

        if new_dim != old_dim:
            raise ValueError(f"Shape mismatch for dimension: '{name}'\n"
                            f"    found: {new_dim}\n"
                            f"    expected: {old_dim}\n")

```

## Download the dataset

“manythings.org/anki” 에서 제공하는 언어 데이터셋을 사용할 예정이다. 이 데이터셋은 다음과 같이 영어 문장과, 대응되는 스페인어 문장을 쌍으로 가진다.

May I borrow this book? ¿Puedo tomar prestado este libro?

(코랩 환경에서 작업하고 있다고 가정.)

작업의 편의를 위해 미리 데이터를 클라우드 로컬 서버에 저장해둔다.

```

# Download the file
import pathlib

path_to_zip = tf.keras.utils.get_file(
    'spa-eng.zip', origin='http://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip',
    extract=True)

path_to_file = pathlib.Path(path_to_zip).parent/'spa-eng/spa.txt'

def load_data(path):
    text = path.read_text(encoding='utf-8')

    lines = text.splitlines()
    pairs = [line.split('\t') for line in lines]

    inp = [inp for targ, inp in pairs]
    targ = [targ for targ, inp in pairs]

    return inp, targ

```

## Prepare the dataset

데이터를 사용하기 앞서 몇 가지 전처리 과정을 거쳐야 한다.

- start, end 토큰을 추가
- special characters 를 제거해서 문장을 정리
- 정수 인코딩 : word index (=id) 를 만든다.
- 문장의 최대 길이에 맞추어 패딩

```
BUFFER_SIZE = len(inp) # buffer size : 1 epoch 되는 데이터 수 = 전체 데이터 수
BATCH_SIZE = 64 # batch size : 1 step 에서 사용되는 데이터 수

dataset = tf.data.Dataset.from_tensor_slices((inp, targ).shuffle(BUFFER_SIZE))
dataset = dataset.batch(BATCH_SIZE)
```

가장 간단한 형태의 배치는 단일 우너소를 n개 만큼 쌓는 것이다. `tf.data.Dataset.batch()` 변환은 정확히 이 작업을 수행한다. `tf.data` 가 동일한 `shape` 을 전파하는 동안, `Dataset.batch`는 가장 마지막 배치의 크기를 알 수 없기 때문에 `None shape` 을 default 로 지정한다. 만약 `drop_remainder` 인자를 사용하면 마지막 배치 크기를 무시하고 지정한 배치 크기를 사용할 수 있다.

ex.

```
batched_dataset= tf.data.Dataset.batch(64, drop_remainder=True)
```

위에서 만든 `dataset`을 출력해보면 다음과 같다.

```
for example_input_batch, example_target_batch in dataset.take(1):
    print(example_input_batch[:5])
    print()
    print(example_target_batch[:5])
    break

>>>tf.Tensor(
[b'Ella no habla solamente ingl\x3\xa9s, tambi\x3\xa9n habla franc\x3\xa9s.'
 b'Tom, di algo.' b'Nunca confundas l\x3\xa1stima con amor.'
 b'Esos colores contrastan muy bien.'
 b'Hablar\x3\xa9 contigo ma\x3\xb1ana.'], shape=(5,), dtype=string)

tf.Tensor(
[b'She can speak not only English but also French.' b'Tom, say something.'
 b'Never confuse pity with love.' b'Those colors go well together.'
 b'I will speak to you tomorrow.'], shape=(5,), dtype=string)
```

다음 과정부터 본격적인 전처리 단계라고 할 수 있다. 첫 번째는 유니코드 정규화 이다. 악센트 문자를 분리하고 `ascii` 문자에서 동등한 문자를 대체시킨다. `tensorflow_text` 패키지는 유니코드 정규화 함수를 가지고 있다.

```
def tf_lower_and_split_punct(text):
    # split accented characters.
    text = tf_text.normalize_utf8(text, 'NFKD')
    text = tf.strings.lower(text)
    # keep space, a to z, and select punctuation.
    text = tf.strings.regex_replace(text, '[^a-z.?!,\&]', '')
    # add spaces around punctuation.
    text = tf.strings.regex_replace(text, '[.?!,\&]', r' \0')
    # strip whitespace.
    text = tf.strings.strip(text)

    text = tf.strings.join(['[START]', text, '[END]'], separator=' ')
    return text

print(example_text.numpy().decode())
print(tf_lower_and_split_punct(example_text).numpy().decode())

>>> ¿Todavía está en casa?
[START] ¿ todavia esta en casa ? [END]
```

## 텍스트 벡터화 → 토큰화

앞서 만든 표준화 함수는 `tf.keras.layers.TextVecorization` 에서 사용된다. 이때 이 layer 에서는 단어 추출과 토큰으로의 변환이 이루어진다.

```
max_vocab_size = 5000

input_text_processor = tf.keras.layers.TextVectorization(
    standardize=tf_lower_and_split_punct,
    max_tokens=max_vocab_size)

# 토큰화
input_text_processor.adapt(inp)

# Here are the first 10 words from the vocabulary:
input_text_processor.get_vocabulary()[:10]
>>> ['', '[UNK]', '[START]', '[END]', '.', 'que', 'de', 'el', 'a', 'no']

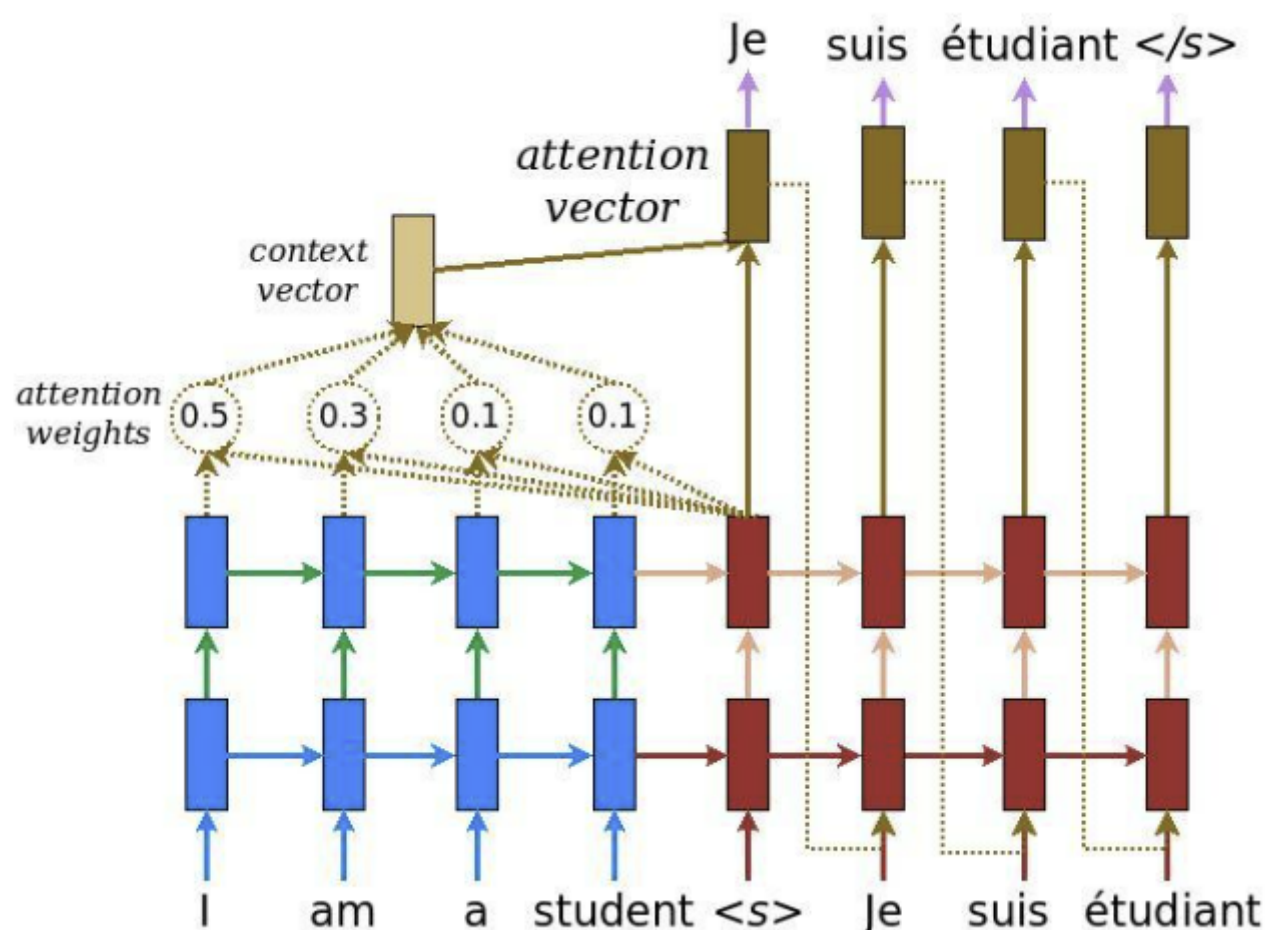
# target 텍스트(영어어 문장)에도 토큰화 처리
output_text_processor = tf.keras.layers.TextVectorization(
    standardize=tf_lower_and_split_punct,
    max_tokens=max_vocab_size)

output_text_processor.adapt(targ)
# 토큰화 -> 단어화해서 출력
output_text_processor.get_vocabulary()[:10]
>>> ['', '[UNK]', '[START]', '[END]', '.', 'the', 'i', 'to', 'you', 'tom']

# 문장을 완전히 토큰화 -> 단어화
input_vocab = np.array(input_text_processor.get_vocabulary())
tokens = input_vocab[example_tokens[0].numpy()]
' '.join(tokens)
>>> '[START] ella no habla solamente ingles , tambien habla frances . [END]'
```

## The encoder/decoder model

튜토리얼에서는 scaled dot product attention 을 사용한다.



우선 임베딩 차원과 뉴런 개수를 설정한다.

```
embedding_dim = 256
units = 1024
```

## encoder

- 토큰 id 리스트를 가져온다. (input\_text\_processor을 통해)
- 토큰 각각은 임베딩 층을 거친다.(layers.Embedding)
- 임베딩을 거쳐 새로운 시퀀스로 변환. (layers.GRU)
- return
  - processed sequence
  - internal state(디코더를 초기화하는데 사용된다.)

```
class Encoder(tf.keras.layers.Layer):
    def __init__(self, input_vocab_size, embedding_dim, enc_units):
        super(Encoder, self).__init__()
        self.enc_units = enc_units
        self.input_vocab_size = input_vocab_size

        # embedding layer : token -> vector
        self.embedding = tf.keras.layers.Embedding(self.input_vocab_size,
            embedding_dim)

        # GRU RNN : process vectors sequentially
        self.gru = tf.keras.layers.GRU(self.enc_units,
            return_sequences=True,
            return_state=True,
            recurrent_initializer='glorot_uniform')

    def call(self, tokens, state=None):
        shape_checker = ShapeChecker()
        shape_checker(tokens, ('batch', 's'))

        # embedding layer looks up the embedding for each token.
        vectors = self.embedding(tokens)
        shape_checker(vectors, ('batch', 's', 'embed_dim'))

        # 3. The GRU processes the embedding sequence.
        #   output shape: (batch, s, enc_units)
        #   state shape: (batch, enc_units)
        output, state = self.gru(vectors, initial_state=state)
        shape_checker(output, ('batch', 's', 'enc_units'))
        shape_checker(state, ('batch', 'enc_units'))

        # 4. Returns the new sequence and its state.
        return output, state
```

위에서 작성한 인코더에서 일어나는 일을 확인하려면 input과 return 의 shape 을 출력해보면 된다.

```
# Convert the input text to tokens.
example_tokens = input_text_processor(example_input_batch)

# Encode the input sequence.
encoder = Encoder(input_text_processor.vocabulary_size(),
    embedding_dim, units)
example_enc_output, example_enc_state = encoder(example_tokens)

print(f'Input batch, shape (batch): {example_input_batch.shape}')
print(f'Input batch tokens, shape (batch, s): {example_tokens.shape}')
print(f'Encoder output, shape (batch, s, units): {example_enc_output.shape}')
print(f'Encoder state, shape (batch, units): {example_enc_state.shape}')
```

## Attention head

디코더는 input sequences 의 부분들을 선별적으로 focus 하기 위해 어텐션 메커니즘을 사용한다. 어텐션은 input으로 일련의 벡터(=행렬)를 사용하고, 각각의 샘플에 대한 어텐션 벡터를 리턴한다. 어텐션 층은 layers.GlobalAveragePooling1D 와 닮았지만 가중합을 수행한다는 점에서 다르다.

$$\alpha_{ts} = \frac{\exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s))}{\sum_{s'=1}^S \exp(\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_{s'}))} \quad \text{[Attention weights]} \quad (1)$$

$$\mathbf{c}_t = \sum_s \alpha_{ts} \bar{\mathbf{h}}_s \quad \text{[Context vector]} \quad (2)$$

- $s$ : 인코더 인덱스
- $t$ : 디코더 인덱스
- $\mathbf{h}_s$  sequence of encoder outputs being attended to
- $\mathbf{h}_t$  decoder state attending to the sequence

attention weights 를 구하기 위한 단어별 score 를 구하는 공식은 다음과 같다.

$$\text{score}(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^\top \mathbf{W} \bar{\mathbf{h}}_s & \text{[Luong's multiplicative style]} \\ \mathbf{v}_a^\top \tanh(\mathbf{W}_1 \mathbf{h}_t + \mathbf{W}_2 \bar{\mathbf{h}}_s) & \text{[Bahdanau's additive style]} \end{cases}$$

튜토리얼에서는 Bahdanau의 수식을 사용한다.

```
class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
        super().__init__()

        # 위에 나온 바다나우 스코어 함수를 구현하기 위한 세팅
        self.W1 = tf.keras.layers.Dense(units, use_bias=False)
        self.W2 = tf.keras.layers.Dense(units, use_bias=False)

        self.attention = tf.keras.layers.AdditiveAttention()

    def call(self, query, value, mask):
        shape_checker = ShapeChecker()
        shape_checker(query, ('batch', 't', 'query_units'))
        shape_checker(value, ('batch', 's', 'value_units'))
        shape_checker(mask, ('batch', 's'))

        # 바다나우 스코어 함수에서 w1@ht
        w1_query = self.W1(query)
        shape_checker(w1_query, ('batch', 't', 'attn_units'))

        # 바다나우 스코어 함수에서 w2@hs
        w2_key = self.W2(value)
        shape_checker(w2_key, ('batch', 's', 'attn_units'))
```

```

query_mask = tf.ones(tf.shape(query)[: -1], dtype=bool)
value_mask = mask

context_value, attetion_weights = self.attention(
    inputs = [w1_query, value, w2_key],
    mask=[query_mask, value_mask],
    return_attetion_scores=True,
)

shape_checker(context_vector, ('batch', 't', 'value_units'))
shape_checker(attention_weights, ('batch', 't', 's'))

return context_vector, attention_weights

```

## Test the Attention layer

어텐션 레이어를 생성해본다.

```
attetion_layer = BahdanauAttention(units)
```

해당 레이어는 3개의 input을 받는다.

- query : 나중에 디코더에 의해 만들어진다.
- value : 인코더의 output
- mask: 패딩을 제외하기 위해 example\_tokens  $\neq$  0 이다.

## decoder

디코더의 역할은 다음 ouput token에 대한 예측을 만드는 것이다.

1. 디코더는 인코더 output 전체를 받는다.
2. 인코더 output 의 생성을 추적하기 위해 RNN을 사용한다.
3. It uses its RNN output as the query to the attention over the encoder's output, producing the context vector
4. 아래의 식을 사용해서 RNN output 과 context vector 를 결합시키고 attention vector 를 생성.
5. It generates logit predictions for the next token based on the “attention vector”.

$$\mathbf{a}_t = f(\mathbf{c}_t, \mathbf{h}_t) = \tanh(\mathbf{W}_c[\mathbf{c}_t; \mathbf{h}_t]) \quad [\text{Attention vector}] \quad (3)$$

```

class Decoder(tf.keras.layers.Layer):
    def __ini__(self, output_vocab_size, embedding_dim, dec_units):
        super(Decoder, self).__init__()
        self.dec_units = dec_units
        self.output_vocab_size = output_voca_size
        self.embedding_dim = embedding_dim

    # The embedding layer converts token IDs to vectors
    self.embedding = tf.keras.layers.Embedding(self.output_vocab_size, embedding_dim)

    # The RNN keeps track of what's been generated so far.
    self.gru = tf.keras.layers.GRU(self.dec_units,
                                    return_sequences=True,
                                    return_state=True,
                                    recurrent_initializer='glorot_uniform')

```



```

# The RNN output will be the query for the attentino layer.
self.attention = BahdanauAttention(self.dec_units)

# converting 'ct' to 'at'
self.Wc = tf.keras.layers.Dense(dec_units, activation=tf.math.tanh,
                                use_biase=False)

# This fully connected layer produces the logits for each ouput tokens.
self.fc = tf.keras.layers.Dense(self.output_vocab_size)

```

디코더의 call() 메서드를 클래스 안에 바로 넣지 않고 나중에 합쳐준다. 디코더의 input, output 이 다소 복잡한 tensors로 이루어져 있기 때문이다. 우선 container class 로 디코더의 input, output을 구성해주고 이것을 call 함수에 인자로 추가.

```

class DecoderInput(typing.NamedTuple):
    new_tokens: Any
    enc_output: Any
    mask: Any

class DecoderOutput(typing.NamedTuple):
    logits: Any
    attention_weights: Any

```

call 함수를 만들어 준다.

```

def call(
    self,
    inputs: DecoderInput,
    state=None
) -> Tuple[DecoderOutput, tf.Tensor]:

    shape_checker = ShapeChecker()
    shape_checker(inputs.new_tokens, ('batch', 't'))
    shape_checker(inputs.enc_output, ('batch', 's', 'enc_units'))
    shape_checker(inputs.mask, ('batch', 's'))

    if state is not None:
        shape_checker(state, ('batch', 'dec_units'))

    # Lookup the embeddings
    vectors = self.embedding(inputs.new_tokens)
    shape_checker(vectors, ('batch', 't', 'embedding_dim'))

    # Process one step with the RNN
    rnn_output, state = self.gru(vectors, initial_state=state)
    shape_checker(rnn_output, ('batch', 't', 'dec_units'))
    shape_checker(state, ('batch', 'dec_units'))

    # Use the RNN output as the query for the attention over the encoder output
    context_vector, attention_weights = self.attention(
        query=rnn_outptu, value=inputs.enc_output, mask=inputs.mask
    )
    shape_checker(context_vector, ('batch', 't', 'dec_units'))
    shape_checker(attention_weights, ('batch', 't', 's'))

    # Join the context_vector and rnn_output
    # [ct; ht] shape: (batch t, value_units + query_units)
    context_and_rnn_output = tf.concat([context_vector, rnn_output], axis=-1)

    # 'at = tanh(Wc@[ct; ht])'
    attention_vector = self.Wc(context_and_rnn_output)
    shape_checker(attention_vector, ('batch', 't', 'dec_units'))

    # Generate logit predictions
    logits = self.fc(attention_vector)
    shape_checker(logits, ('batch', 't', 'output_vocab_size'))

    return DecoderOutput(logits, attention_weights), state

```

위에서 만든 call 함수를 디코더 클래스의 call 메서드로 정의.



```
Decoder.call = call
```

## Training

이제 모든 모델 주요 구성요소를 완성했고, 추가적으로 필요한 부분은 다음과 같다.

- 손실 함수와 optimizer
- input/target batch에 대한 모델의 업데이트를 정의하는 training step function
- 훈련과 체크포인트를 저장하는 training loop

## loss function

우선 손실함수 부터 정의한다.

```
class MaskedLoss(tf.keras.losses.Loss):
    def __init__(self):
        self.name = 'masked_loss'
        self.loss = tf.keras.losses.SparseCategoricalCrossentropy(
            from_logits=True, reduction='none')

    def __call__(self, y_true, y_pred):
        shape_checker = ShapeChecker()
        shape_checker(y_true, ('batch', 't'))
        shape_checker(y_pred, ('batch', 't', 'logits'))

        # calculate the loss for each item in the batch.
        loss = self.loss(y_true, y_pred)
        shape_checker(loss, ('batch', 't'))

        # mask off the losses on padding.
        # 실제값이 0 이라면(padding 이라면) 0을 반환해서 masking.
        mask = tf.cast(y_true != 0, tf.float32)
        shape_checker(mask, ('batch', 't'))
        loss *= mask

        # return the total
        return tf.reduce_sum(loss)
```

- `__init__` 은 객체가 생성될 때 불러와지고, `__call__` 은 객체가 호출되었을 때 실행된다. 그 동안 사용했다. `call()` 메서드는 keras Layer 의 `call` 방식이었다.
- `tf.cast()` 함수는 텐서를 새로운 형태로 캐스팅하는데 사용한다. 부동소수점형에서 정수형으로 바꾼 경우 소수점 버림을 한다. Boolean 의 경우 True 이면 1, False 는 0을 출력한다.

## training step 구현

train step 을 구현하기 위한 클래스를 하나 만들 예정이다. 이 클래스에는 훈련 과정을 진행하는 `train_step()` 이란 메서드가 있게되고, 추후 만 들어질 `'_train_step'` 구현체를 감싸주는 역할을 한다. 이러한 wrapper 는 디버깅을 더 쉽게 하기 위해 `tf.function` 을 on/off 하는 스위치 기능을 포함한다.

```
class TrainTranslator(tf.keras.Model):
    def __init__(
        self, embedding_dim, units,
```

```

input_text_processor,
output_text_processor,
use_tf_function=True):
    super().__init__()

    # build the encoder and decoder
    encoder = Encoder(input_text_processor.vocabulary_size(),
                      embedding_dim, units)
    decoder = Decoder(output_text_processor.vocabulary_size(),
                      embedding_dim, units)
    self.encoder = encoder
    self.decoder = decoder
    self.input_text_processor = input_text_processor
    self.output_text_processor = output_text_processor
    self.use_tf_function = use_tf_function
    self.shape_checker = ShapeChecker()

def train_step(self, inputs):
    self.shape_checker = ShapeChecker()
    if self.use_tf_function():
        return self._tf_train_step(inputs)
    else:
        return self._train_step(inputs)

```

- tf.data.Dataset 으로부터 input\_text, target\_text 의 1개 배치를 받는다.
- raw text input 을 token-embedding 과 마스크 처리된 데이터로 변환한다.

아래의 \_preprocess 메서드는 데이터셋 배치를 받아서 토큰화하고 masking 처리까지 한다.

```

def _preprocess(self, input_text, target_text):
    self.shape_checker(input_text, ('batch',))
    self.shape_checker(target_text, ('batch',))

    # convert the text to tokens IDs
    input_tokens = self.input_text_processor(input_text)
    target_tokens = self.output_text_processor(target_text)
    self.shape_checker(input_tokens, ('batch', 's'))
    self.shape_checker(target_tokens, ('batch', 't'))

    # convert IDs to masks.
    input_mask = input_tokens != 0
    self.shape_checker(input_mask, ('batch', 's'))
    target_mask = target_tokens != 0
    self.shape_checker(target_mask, ('batch', 't'))

    return input_tokens, input_mask, target_tokens, target_mask

# 메서드 -> 클래스
TrainTranslator._preprocess = _preprocess

```

- 인코더는 input\_tokens(토큰화 된 데이터) 을 입력으로 받아 encoder\_output, encoder\_state 를 리턴.
- decoder state, loss 를 초기화한다.
- Loop over the target\_tokens:
  - Run the decoder one step at a time.
  - Calculate the loss for each step.
  - Accumulate the average loss.
- Calculate the gradient of the loss and use the optimizer to apply updates to the model's trainable\_variables.

위에서 언급된 training step 의 나머지 과정을 구현해준다.

-- 내용 추가중 --

