① Get Dataset & build tokenizer

1.1 : raw dataset from huggingface.

```
# Huggingface
from datasets import load_dataset

raw_dataset = load_dataset(f'{config["datasource"]}', f'{config["lang_src"]}-{config["lang_tgt"]}', split='train')

    # ex) load_dataset('opus_books', 'en-fr', split='train')
                        ↳ Dataset    ↳ Subset
```

1.2 build tokenizer (need tokenizers for encoder and decoder)

- ✱ in our case, encoder_tokenizer is tokenizing English sentence
  decoder_tokenizer is tokenizing French sentence

- ✱ Tokenizer Pipeline : Normalization (optional) → model → Pre-tokenization → train → Post-Processing (optional)

  - Pre-tokenization is the act of splitting a text into smaller objects. A good way to think of pre-tokenization is that it will split your text into words.

```
# Huggingface
from tokenizers import Tokenizer
from tokenizers.model import WordLevel
from tokenizers.trainers import WordLevelTrainer
from tokenizers.pre_tokenizers import Whitespace

tokenizer_src = Tokenizer(WordLevel(unk_token='UNK'))   # model; unk_token: Unknown token
tokenizer_src.pre_tokenizer = Whitespace()  # Split text into words using white spaces.
trainer = WordLevelTrainer(special_tokens=["[UNK]", "[PAD]", "[SOS]", "[EOS]"], min_frequency=2)
tokenizer_src.train_from_iterator(iterator=get_all_sentences(data,"en"), trainer=trainer)
                                    ↳ def get_all_sentences(data,lang):
                                          for item in data:
                                              yield item['translation'][lang]

tokenizer_tgt = Tokenizer(WordLevel(unk_token='UNK'))   # model
tokenizer_tgt.pre_tokenizer = Whitespace()  # Split text into words using white spaces.
trainer = WordLevelTrainer(special_tokens=["[UNK]", "[PAD]", "[SOS]", "[EOS]"], min_frequency=2)
tokenizer_tgt.train_from_iterator(iterator=get_all_sentences(data,"fr"), trainer=trainer)
```

1.3 Split train & test

```
train_size = int(0.9 x len(raw_dataset)
test_size = len(raw_dataset) - train_size
train, val = random_split(raw_dataset, [train_size, val_size])
               ↳ torch.utils.data.random_split
```

1.4 Create a customed dataset by using Tokenizer (torch.utils.data.Dataset)

: A custom Dataset class must implement three functions: __init__, __len__, and __getitem__

```
class BilingualDataset(Dataset):

    def __init__(self, data, tokenizer_src, tokenizer_tgt, src_lang, tgt_lang, seq_len):
        super().__init__()

        self.data = data
        self.tokenizer_src = tokenizer_src
                ⋮

        self.sos_token = torch.tensor([tokenizer_src.token_to_id("[SOS]")], dtype=torch.int64)
        self.eos_token     "
        self.pad_token     "

    def causal_mask(self, size):
        mask = torch.triu(torch.ones(1, size, size), diagonal=1)

    def __getitem__(self, index):
        pairs = self.data[index]
        src_txt = pairs['translation'][self.src_lang]
        tgt_txt = pairs['translation'][self.tgt_lang]
```

src_txt → [src_tokenizer] → Add special tokens → encoder_input

tgt_txt → [tgt_tokenizer] → Add special tokens → decoder_input

```
encoder_mask = (encoder_input != self.pad_token).unsqueeze(0).int()  # size : (B, seq_len)
```

[[1, 1, 1, 1, 1, 1, 0, 0, ⋯ , 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, .. , 0, 0, 0],
⋮
[1, 1, 1, 1, 0, 0, 0, 0, .. , 0, 0, 0]]

Batch size

seq_len : 350

```
decoder_mask = (decoder_input != self.pad_token).unsqueeze(0).int() & self.causal_mask(decoder_input.size(0))  # (Batch, seq_len, seq_len)
```

[[[1, 0, 0, 0, ⋯ , 0, 0, 0],
[1, 1, 0, 0, ⋯ , 0, 0, 0],
[1, 1, 1, 0, ⋯ , 0, 0, 0],
[1, 1, 1, 1, .. , 0, 0, 0],
⋮
[1, 1, 1, 1, .. , 0, 0, 0]],

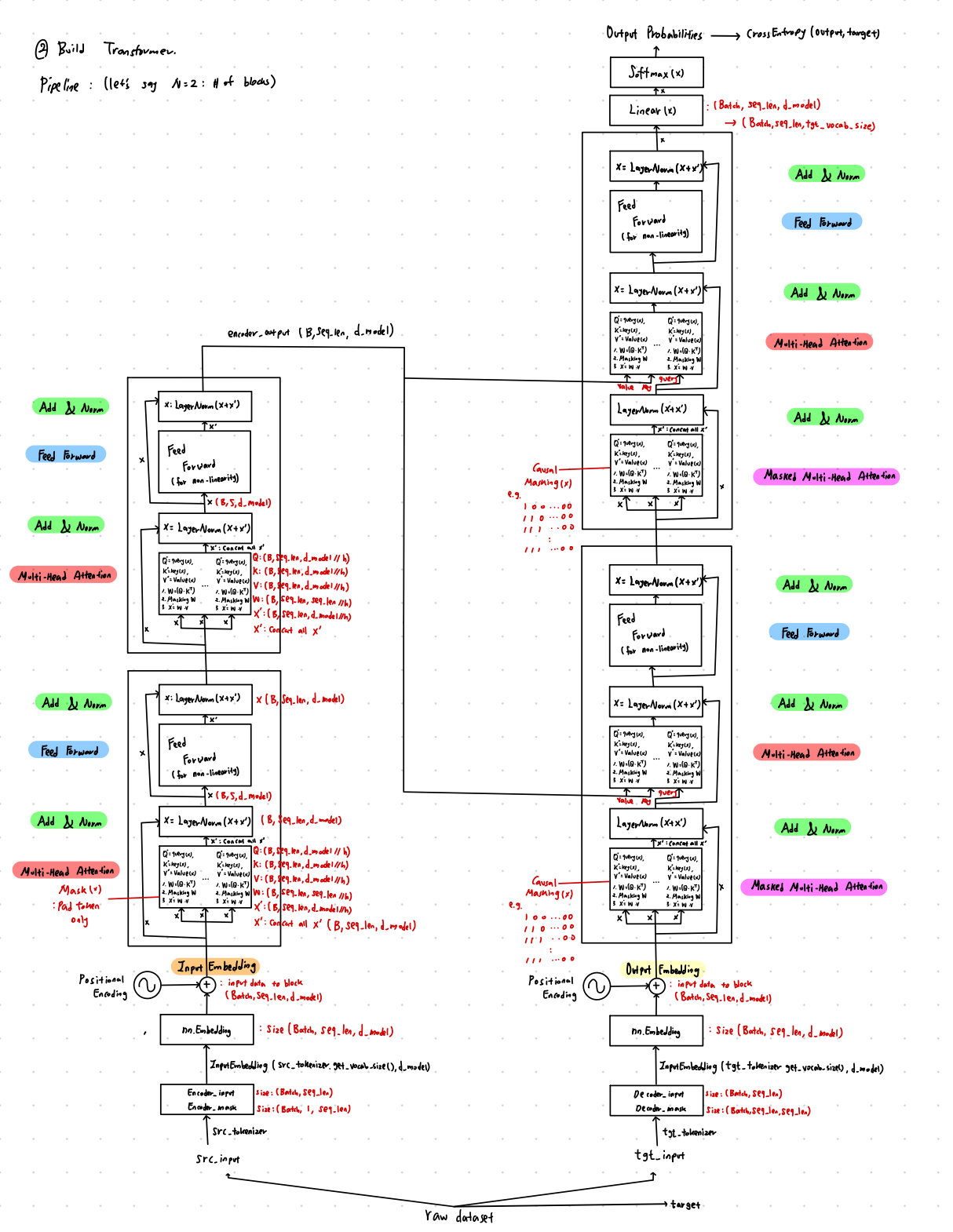
seq_len

seq_len

Batch_size

[[1, 0, 0, 0, ⋯ , 0, 0, 0],
[1, 1, 0, 0, ⋯ , 0, 0, 0],
[1, 1, 1, 0, ⋯ , 0, 0, 0],
[1, 1, 1, 1, .. , 0, 0, 0],
⋮
[1, 1, 1, 1, .. , 0, 0, 0]]

Return ⋯


② Build Transformer.
Pipeline : (let's say N=2 : # of blocks)
Output Probabilities
CrossEntropy (output, target)
Softmax (x)
Linear (x)
: (Batch, seq_len, d_model)
→ (Batch, seq_len, tgt_vocab_size)
x= LayerNorm (x+x')
Add & Norm
Feed Forward
(for non-linearity)
Feed Forward
Multi-Head Attention
Value Key
Query
LayerNorm (x+x')
x': Concat all x'
Masked Multi-Head Attention
Causal Masking (x)
e.g.
1 0 0 ... 0 0
1 1 0 ... 0 0
1 1 1 ... 0 0
1 1 1 ... 1 0
encoder_output (B, Seq_len, d_model)
x (B, S, d_model)
Q: (B, Seq_len, d_model // h)
K: (B, Seq_len, d_model//h)
V: (B, Seq_len, d_model//h)
W: (B, Seq_len, Seq_len //h)
x': (B, Seq_len, d_model//h)
x': Concat all x'
x (B, Seq_len, d_model)
(B, Seq_len, d_model)
Q: (B, Seq_len, d_model // h)
K: (B, Seq_len, d_model//h)
V: (B, Seq_len, d_model//h)
W: (B, Seq_len, Seq_len //h)
x': (B, Seq_len, d_model//h)
x': Concat all x' (B, Seq_len, d_model)
Mask (x)
: Pad token only
Input Embedding
Output Embedding
Positional Encoding
: input data to block
(Batch, Seq_len, d_model)
nn.Embedding
: Size (Batch, Seq_len, d_model)
InputEmbedding (src_tokenizer.get_vocab_size(), d_model)
InputEmbedding (tgt_tokenizer.get_vocab_size(), d_model)
Encoder_input
Encoder_mask
Size: (Batch, Seq_len)
Size: (Batch, 1, Seq_len)
Decoder_input
Decoder_mask
Size: (Batch, Seq_len)
Size: (Batch, Seq_len, Seq_len)
src_tokenizer
tgt_tokenizer
src_input
tgt_input
Raw dataset
target