LANGUAGE TRANSLATION WITH

NN.TRANSFORMER & TORCH TEXT

Data Sourcing & Processing

- how to use torchtext's inbuilt datasets
- tokenize a raw text sentence
- build vocabulary
- numericalize tokens into tensor

```
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torchtext.datasets import Multi30k
from typing import Iterable, List
```

```
SRC_LANGUAGE = 'de'
TGT_LANGUAGE = 'en'

# Place-holders
token_transform = {}
vocab_transform = {}

# Create source & target language tokenizer.
# Make sure to install the dependencies.

# pip install -U spacy
# python -m spacy download en_core_web_sm
# python -m spacy download de_core_news_sm
token_transform[SRC_LANGUAGE] = get_tokenizer('spacy', language='de_core_news_sm')
token_transform[TGT_LANGUAGE] = get_tokenizer('spacy', language='en_core_web_sm')
```

```
# helper function to yield list of tokens
def yield_tokens(data_iter: Iterable, language: str) -> List[str]:
    language_index = {SRC_LANGUAGE: 0, TRT_LANGUAGE: 1}

    for data_sample in data_iter:
        yield token_transform[language](data_sample[language_index[language]])
```

get_tokenizer() 함수

SRC / TGT의 data_sample을 tokenizing.

Language index

data_sample[0] or data_sample[1]

```
# Define special symbols and indices
UNK_IDX, PAD_IDX, BOS_IDX, EOS_IDX = 0, 1, 2, 3
special_symbols = ['<unk>', '<pad>', '<bos>', '<EOS>']

for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    # Training data Iterator
    train_iter = Multi30k (split = 'train', language_pair = (SRC_LANGUAGE, TGT_LANGUAGE))
    # Create torchtext's Vocab object
    vocab_transform[ln] = build_vocab_from_iterator (yield_tokens (train_iter, ln),
                                                     min_freq = 1,
                                                     specials = special_symbols,
                                                     special_first = True)

# Set UNK_IDX as the default index.
# This index is returned when the token is not found.
# If not set, it throws RuntimeError when the queried token is not found in the vocabulary.
for ln in [SRC_LANGUAGE, TGT_LANGUAGE]:
    vocab_transform[ln].set_default_index(UNK_IDX)
```

Seq2Seq Network using Transformer

① embedding layer

- tensor of input indices → tensor of input embeddings
- These embeddings are further augmented with positional encodings.

② actual Transformer model

③ linear layer

- give un-normalized probabilities for each token in the target language

```
from torch import Tensor
import torch
import torch.nn as nn
from torch.nn import Transformer
import math
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# helper module that adds positional encoding to the token embedding
# to introduce a notion of word order.

class PositionalEncoding(nn.Module):
    def __init__(self,
                 emb_size: int,
                 dropout: float,
                 maxlen: int = 5000):
        super(PositionalEncoding, self).__init__()
        den = torch.exp(-torch.arange(0, emb_size, 2) * math.log(10000) / emb_size)
        pos = torch.arange(0, maxlen).reshape(maxlen, 1)
        pos_embedding = torch.zeros((maxlen, emb_size))
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        pos_embedding[:, 1::2] = torch.cos(pos * den)
        pos_embedding = pos_embedding.unsqueeze(-2)   → 뒤에서 두번째에 차원 추가 (세로로 펴짐 해결)

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, token_embedding: Tensor):
        return self.dropout(token_embedding + self.pos_embedding[:token_embedding.size(0), :])
```

```
# helper Module to convert tensor of input indices
# into corresponding tensor of token embeddings

class TokenEmbedding (nn.Module):
    def __init__ (self, vocab_size : int, emb_size):
        super (TokenEmbedding, self). __init__()
        self. embedding = nn.Embedding (vocab_size, emb_size)
        self. emb_size = emb_size

    def forward (self, token : Tensor):
        return self. embedding (tokens. long()) * math. sqrt (self. emb_size)


# Seq2Seq Network
class Seq2SeqTransformer (nn.Module):
    def __init__ (self,
                  num_encoder_layers : int,
                  num_decoder_layers : int,
                  emb_size : int,
                  nhead : int,
                  src_vocab_size : int,
                  tgt_vocab_size : int,
                  dim_feedforward : int = 512,
                  dropout : float = 0.1):
        super (Seq2SeqTransformer, self). __init__()
        self. transformer = Transformer ( d_model = emb_size,
                                          nhead = nhead,
                                          num_encoder_layers = num_encoder_layers,
                                          num_decoder_layers = num_decoder_layers,
                                          dim_feedforward = dim_feedforward,
                                          dropout = dropout )
        self. generator = nn. Linear (emb.size, tgt_vocab_size )
        self. src_tok_emb = TokenEmbedding ( src_vocab_size, emb_size )
        self. tgt_tok_emb = TokenEmbedding ( tgt_vocab_size, emb_size )
        self. positional_encoding = Positional Encoding ( emb_size, dropout = dropout )
```

```
def forward (self,
                src : Tensor,
                tgt : Tensor,
                src_mask : Tensor,
                tgt_mask : Tensor,
                src_padding_mask = Tensor,
                tgt_padding_mask = Tensor,
                memory_key_padding_mask : Tensor):
    src_emb = self.positional_encoding (self.src_tok_emb (src))
    tgt_emb = self.positional_encoding (self.tgt_tok.emb (tgt))
    outs= self.transformer (src_emb, tgt_emb, src_mask, tgt_mask, None,
                            src_padding_mask, tgt_padding_mask,
                            memory_key_padding_mask)
    return self.generator (outs)

def encode (self, src: Tensor, src_mask: Tensor):
    return self.transformer.encoder (self.positional_encoding (self.src_tok_emb (src)), src_mask)

def decode (self, tgt: Tensor, memory: Tensor, tgt_mask : Tensor):
    return self.transformer.decoder (self.positional_encoding (self.tgt_tok_emb (tgt)), memory, tgt_mask)
```

During training, we need :

① a subsequent word mask that will prevent model to look into the future words when making predictions.

② masks to hide source & target padding tokens.

```
def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz, sz), device=DEVICE)) == 1).transpose(0,1)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask ==1, float(0,0))
    return mask

def create_mask(src, tgt):
    src_seq_len = src.shape[0]
    tgt_seq_len = tgt.shape[0]

    tgt_mask = generate_square_subsequent_mask(tgt_seq_len)
    src_mask = torch.zeros((src_seq_len, src_seq_len), device=DEVICE).type(torch.bool)

    src_padding_mask = (src == PAD_IDX).transpose(0,1)
    tgt_padding_mask = (tgt == PAD_IDX).transpose(0,1)
    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

Define parameters.

```
torch.manual_seed(0)

SRC_VOCAB_SIZE = len(vocab_transform[SRC_LANGUAGE])
TGT_VOCAB_SIZE = len(vocab_transform[TGT_LANGUAGE])
EMB_SIZE = 512
NHEAD = 8
FFN_HID_DIM = 512
BATCH_SIZE = 128
NUM_ENCODER_LAYERS = 3
NUM_DECODER_LAYERS = 3

transformer = Seq2SeqTransformer(NUM_ENCODER_LAYERS,
                                 NUM_DECODER_LARERS,
                                 EMB_SIZE,
                                 NHEAD,
                                 SRC_VOCAB_SIZE,
                                 TGT_VOCAB_SIZE,
                                 FFN_HID_DIM)
for p in transformer.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)

transformer = transformer.to(DEVICE)

loss_fn = torch.nn.CrossEntropyLoss(ignore_index = PAD_IDX)

optimizer = torch.optim.Adam(transformer.parameters(),
                             lr = 0.0001,
                             betas = (0.9, 0.98),
                             eps = 1e-9)
```