BERT (codertimo)

- Reference: https://github.com/codertimo/BERT-pytorch

1. Vocab 저장 방식

```
class TorchVocab(object):
    """Defines a vocabulary object that will be used to numericalize a field.
    Attributes:
        freqs: A collections.Counter object holding the frequencies of tokens
            in the data used to build the Vocab.
        stoi: A collections.defaultdict instance mapping token strings to
            numerical identifiers.
        itos: A list of token strings indexed by their numerical identifiers.
    """

    def __init__(self, counter, max_size=None, min_freq=1, specials=['<pad>', '<oov
                 vectors=None, unk_init=None, vectors_cache=None):
        """Create a Vocab object from a collections.Counter.
        Arguments:
            counter: collections.Counter object holding the frequencies of
                each value found in the data.
            max_size: The maximum size of the vocabulary, or None for no
                maximum. Default: None.
            min_freq: The minimum frequency needed to include a token in the
                vocabulary. Values less than 1 will be set to 1. Default: 1.
            specials: The list of special tokens (e.g., padding or eos) that
                will be prepended to the vocabulary in addition to an <unk>
                token. Default: ['<pad>']
            vectors: One of either the available pretrained vectors
                or custom pretrained vectors (see Vocab.load_vectors);
                or a list of aforementioned vectors
            unk_init (callback): by default, initialize out-of-vocabulary word vect
                to zero vectors; can be any function that takes in a Tensor and
                returns a Tensor of the same size. Default: torch.Tensor.zero_
            vectors_cache: directory for cached vectors. Default: '.vector_cache'
        """
        self.freqs = counter
        counter = counter.copy()
        min_freq = max(min_freq, 1)

        self.itos = list(specials)
        # frequencies of special tokens are not counted when building vocabulary
        # in frequency order
        for tok in specials:
            del counter[tok]

        max_size = None if max_size is None else max_size + len(self.itos)

        # sort by frequency, then alphabetically
        words_and_frequencies = sorted(counter.items(), key=lambda tup: tup[0])
```

```
        words_and_frequencies.sort(key=lambda tup: tup[1], reverse=True)

        for word, freq in words_and_frequencies:
            if freq < min_freq or len(self.itos) == max_size:
                break
            self.itos.append(word)

        # stoi is simply a reverse dict for itos
        self.stoi = {tok: i for i, tok in enumerate(self.itos)}

        self.vectors = None
        if vectors is not None:
            self.load_vectors(vectors, unk_init=unk_init, cache=vectors_cache)
        else:
            assert unk_init is None and vectors_cache is None

    def __eq__(self, other):
        if self.freqs != other.freqs:
            return False
        if self.stoi != other.stoi:
            return False
        if self.itos != other.itos:
            return False
        if self.vectors != other.vectors:
            return False
        return True


    def __len__(self):
        return len(self.itos)

    def vocab_rerank(self):
        self.stoi = {word: i for i, word in enumerate(self.itos)}

    def extend(self, v, sort=False):
        words = sorted(v.itos) if sort else v.itos
        for w in words:
            if w not in self.stoi:
                self.itos.append(w)
                self.stoi[w] = len(self.itos) - 1


class Vocab(TorchVocab):
    def __init__(self, counter, max_size=None, min_freq=1):
        self.pad_index = 0
        self.unk_index = 1
        self.eos_index = 2
        self.sos_index = 3
        self.mask_index = 4
        super().__init__(counter, specials=["<pad>", "<unk>", "<eos>", "<sos>", "<m
                         max_size=max_size, min_freq=min_freq)

    def to_seq(self, sentece, seq_len, with_eos=False, with_sos=False) -> list:
        pass

    def from_seq(self, seq, join=False, with_pad=False):
```

```
        pass

    @staticmethod
    def load_vocab(vocab_path: str) -> 'Vocab':
        with open(vocab_path, "rb") as f:
            return pickle.load(f)

    def save_vocab(self, vocab_path):
        with open(vocab_path, "wb") as f:
            pickle.dump(self, f)


# Building Vocab with text files
class WordVocab(Vocab):
    def __init__(self, texts, max_size=None, min_freq=1):
        print("Building Vocab")
        counter = Counter()
        for line in tqdm.tqdm(texts):
            if isinstance(line, list):
                words = line
            else:
                words = line.replace("\n", "").replace("\t", "").split()

            for word in words:
                counter[word] += 1
        super().__init__(counter, max_size=max_size, min_freq=min_freq)

    def to_seq(self, sentence, seq_len=None, with_eos=False, with_sos=False, with_l
        if isinstance(sentence, str):
            sentence = sentence.split()

        seq = [self.stoi.get(word, self.unk_index) for word in sentence]

        if with_eos:
            seq += [self.eos_index]  # this would be index 1
        if with_sos:
            seq = [self.sos_index] + seq

        origin_seq_len = len(seq)

        if seq_len is None:
            pass
        elif len(seq) <= seq_len:
            seq += [self.pad_index for _ in range(seq_len - len(seq))]
        else:
            seq = seq[:seq_len]

        return (seq, origin_seq_len) if with_len else seq

    def from_seq(self, seq, join=False, with_pad=False):
        words = [self.itos[idx]
                 if idx < len(self.itos)
                 else "<%d>" % idx
                 for idx in seq
                 if not with_pad or idx != self.pad_index]
```

```
        return " ".join(words) if join else words

    @staticmethod
    def load_vocab(vocab_path: str) -> 'WordVocab':
        with open(vocab_path, "rb") as f:
            return pickle.load(f)


def build():
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument("-c", "--corpus_path", required=True, type=str)
    parser.add_argument("-o", "--output_path", required=True, type=str)
    parser.add_argument("-s", "--vocab_size", type=int, default=None)
    parser.add_argument("-e", "--encoding", type=str, default="utf-8")
    parser.add_argument("-m", "--min_freq", type=int, default=1)
    args = parser.parse_args()

    with open(args.corpus_path, "r", encoding=args.encoding) as f:
        vocab = WordVocab(f, max_size=args.vocab_size, min_freq=args.min_freq)

    print("VOCAB SIZE:", len(vocab))
    vocab.save_vocab(args.output_path)
```

vocab을 객체로 만들어 저장 및 불러와서 사용, vocab의 len() 혹은 == 연산자 사용을 위해 magic method 정의해서 사용

- `__eq__` & `__len__`

```
def __eq__(self, other):
    if self.freqs != other.freqs:
        return False
    if self.stoi != other.stoi:
        return False
    if self.itos != other.itos:
        return False
    if self.vectors != other.vectors:
        return False
    return True


def __len__(self):
    return len(self.itos)
```

Properties

- 크게 3개의 Class로 만들어서 vocab 구성

- `TorchVocab`
- word frequency counter를 사용.
- `min_freq` 설정 맞춰 vocab설정 후 `stoi` & `itos` 설정.
- `__len__` , `__eq__` 사용
- `load_vector()` 의 경우 구조상 `Vocab` Class에 구현되어야 함. pre-trained vector setup하는 부분 여기선 vector 변수 존재하지만 해당 부분 구현 x.
- `Vocab`
- `TorchVocab` 상속
- 기본적인 special token index 설정
- `load_vocab()` (static) 및 `save_vocab()` 구현
- `WordVocab`
- 실질적 사용하는 Vocab 객체, `Vocab` 상속.
- text corpus 불러와서 counter 객체 만듬
- idx <=> text 함수구현
- `build()`
- vocab 처음 구축시 사용 함수

기타

- `Object` 상속 의미
  - Python 3.x:
    class MyClass(object): new-style 클래스
    class MyClass: new-style 클래스 (명시하지 않아도 자동으로 상속)
  - Python 2.x:
    class MyClass(object): new-style 클래스
    class MyClass: >>>>OLD-STYLE 클래스<<<<

2. On Memory / Out Memory

```
with open(corpus_path, "r", encoding=encoding) as f:
    if self.corpus_lines is None and not on_memory:
        for _ in tqdm.tqdm(f, desc="Loading Dataset", total=corpus_lines):
            self.corpus_lines += 1

    if on_memory:
        self.lines = [line[:-1].split("\t")
                      for line in tqdm.tqdm(f, desc="Loading Dataset", total=corpus
        self.corpus_lines = len(self.lines)

if not on_memory:
    self.file = open(corpus_path, "r", encoding=encoding)
    self.random_file = open(corpus_path, "r", encoding=encoding)

    for _ in range(random.randint(self.corpus_lines if self.corpus_lines < 1000 els
        self.random_file.__next__()
```

데이터 역시 vocab처럼 `Dataset` 객체화 시켜서 `__getitem__()` 사용해서 데이터 불러오는데, 그 중에서 On memory 변수 사용해서 한번에 텍스트를 다 읽거나, 아니면 open만 시켜둔 후 이후 하나씩 불러와서 읽음

3. Masking & Next Sentence

```
def random_word(self, sentence):
    tokens = sentence.split()
    output_label = []

    for i, token in enumerate(tokens):
        prob = random.random()
        if prob < 0.15:
            prob /= 0.15

            # 80% randomly change token to mask token
            if prob < 0.8:
                tokens[i] = self.vocab.mask_index

            # 10% randomly change token to random token
            elif prob < 0.9:
                tokens[i] = random.randrange(len(self.vocab))

            # 10% randomly change token to current token
            else:
                tokens[i] = self.vocab.stoi.get(token, self.vocab.unk_index)

            output_label.append(self.vocab.stoi.get(token, self.vocab.unk_index))

        else:
            tokens[i] = self.vocab.stoi.get(token, self.vocab.unk_index)
            output_label.append(0)

    return tokens, output_label

def random_sent(self, index):
    t1, t2 = self.get_corpus_line(index)

    # output_text, label(isNotNext:0, isNext:1)
    if random.random() > 0.5:
        return t1, t2, 1
    else:
        return t1, self.get_random_line(), 0
```

- token의 15%는 UNK처리함.(논문에서 해당 technique 못찾음)
- 이후 80%,10%,10%는 처리

- next sentence의 경우 절반은 next로 나머지는 다른 임의의 sentence로

4. Model

BERT의 모델구조는 다음과 같이 짜여져있다.

- `TransformerBlock`
- `BERT`
- `BERTLM` (BERT Language Model)

`BERT` Class 에서 TransformerBlock을 n 번 반복시켜 전체 모델을 만든다.
`BERTLM` Class는 `BERT` 객체를 생성 후 2가지 task인 next sentence/predict masked token을 진행한다.

transformer.py

```
import torch.nn as nn

from .attention import MultiHeadedAttention
from .utils import SublayerConnection, PositionwiseFeedForward


class TransformerBlock(nn.Module):
    """
    Bidirectional Encoder = Transformer (self-attention)
    Transformer = MultiHead_Attention + Feed_Forward with sublayer connection
    """

    def __init__(self, hidden, attn_heads, feed_forward_hidden, dropout):
        """
        :param hidden: hidden size of transformer
        :param attn_heads: head sizes of multi-head attention
        :param feed_forward_hidden: feed_forward_hidden, usually 4*hidden_size
        :param dropout: dropout rate
        """

        super().__init__()
        self.attention = MultiHeadedAttention(h=attn_heads, d_model=hidden)
        self.feed_forward = PositionwiseFeedForward(d_model=hidden, d_ff=feed_forwa
        self.input_sublayer = SublayerConnection(size=hidden, dropout=dropout)
        self.output_sublayer = SublayerConnection(size=hidden, dropout=dropout)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x, mask):
        x = self.input_sublayer(x, lambda _x: self.attention.forward(_x, _x, _x, ma
        x = self.output_sublayer(x, self.feed_forward)
        return self.dropout(x)
```

bert.py

```
import torch.nn as nn

from .transformer import TransformerBlock
from .embedding import BERTEmbedding


class BERT(nn.Module):
    """
    BERT model : Bidirectional Encoder Representations from Transformers.
    """

    def __init__(self, vocab_size, hidden=768, n_layers=12, attn_heads=12, dropout=
        """
        :param vocab_size: vocab_size of total words
        :param hidden: BERT model hidden size
        :param n_layers: numbers of Transformer blocks(layers)
        :param attn_heads: number of attention heads
        :param dropout: dropout rate
        """

        super().__init__()
        self.hidden = hidden
        self.n_layers = n_layers
        self.attn_heads = attn_heads

        # paper noted they used 4*hidden_size for ff_network_hidden_size
        self.feed_forward_hidden = hidden * 4

        # embedding for BERT, sum of positional, segment, token embeddings
        self.embedding = BERTEmbedding(vocab_size=vocab_size, embed_size=hidden)

        # multi-layers transformer blocks, deep network
        self.transformer_blocks = nn.ModuleList(
            [TransformerBlock(hidden, attn_heads, hidden * 4, dropout) for _ in ran

    def forward(self, x, segment_info):
        # attention masking for padded token
        # torch.ByteTensor([batch_size, 1, seq_len, seq_len)
        mask = (x > 0).unsqueeze(1).repeat(1, x.size(1), 1).unsqueeze(1)

        # embedding the indexed sequence to sequence of vectors
        x = self.embedding(x, segment_info)

        # running over multiple transformer blocks
        for transformer in self.transformer_blocks:
            x = transformer.forward(x, mask)

        return x
```

language_model.py

```
import torch.nn as nn

from .bert import BERT


class BERTLM(nn.Module):
    """
    BERT Language Model
    Next Sentence Prediction Model + Masked Language Model
    """

    def __init__(self, bert: BERT, vocab_size):
        """
        :param bert: BERT model which should be trained
        :param vocab_size: total vocab size for masked_lm
        """

        super().__init__()
        self.bert = bert
        self.next_sentence = NextSentencePrediction(self.bert.hidden)
        self.mask_lm = MaskedLanguageModel(self.bert.hidden, vocab_size)

    def forward(self, x, segment_label):
        x = self.bert(x, segment_label)
        return self.next_sentence(x), self.mask_lm(x)


class NextSentencePrediction(nn.Module):
    """
    2-class classification model : is_next, is_not_next
    """

    def __init__(self, hidden):
        """
        :param hidden: BERT model output size
        """
        super().__init__()
        self.linear = nn.Linear(hidden, 2)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self, x):
        return self.softmax(self.linear(x[:, 0]))


class MaskedLanguageModel(nn.Module):
    """
    predicting origin token from masked input sequence
    n-class classification problem, n-class = vocab_size
    """

    def __init__(self, hidden, vocab_size):
        """
```

```
        :param hidden: output size of BERT model
        :param vocab_size: total vocab size
        """
        super().__init__()
        self.linear = nn.Linear(hidden, vocab_size)
        self.softmax = nn.LogSoftmax(dim=-1)

    def forward(self, x):
        return self.softmax(self.linear(x))
```

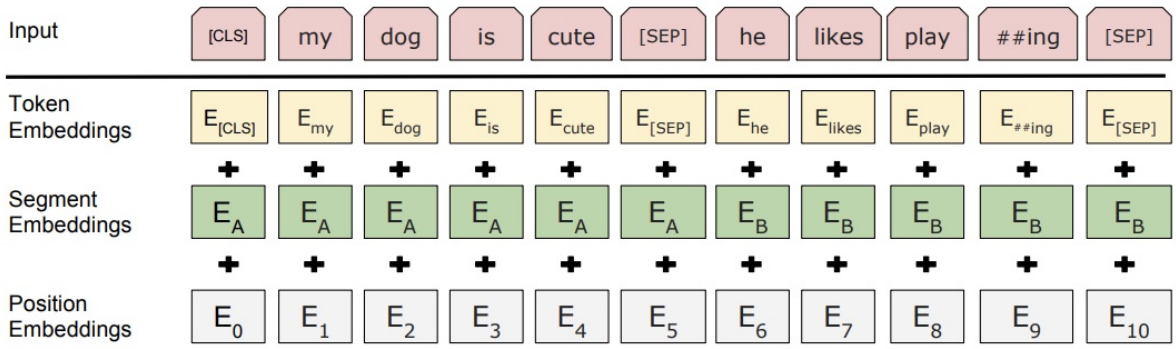
5. BERT embedding

앞서 BERT 모델 부분을 보면 embedding과 TransformerBlock이 구분되어있는것을 볼 수 있다.

```
# embedding for BERT, sum of positional, segment, token embeddings
self.embedding = BERTEmbedding(vocab_size=vocab_size, embed_size=hidden)

# multi-layers transformer blocks, deep network
self.transformer_blocks = nn.ModuleList(
    [TransformerBlock(hidden, attn_heads, hidden * 4, dropout) for _ in range(n_lay
```

먼저 BERTEmbedding 모듈은 다음 그림과 같이 Token embedding & Segment Embeddings & Position Embeddings 부분을 구현하는 부분이다.



Embedding은 총 5개의 py파일로 구성되어있다.

- `__init__.py`
  - init하는 부분, `bert.py`의 `BERTEmbedding` 클래스를 임포트
- `bert.py`
  - 3개의 임베딩 기법은 하나로 합치는 부분
- `position.py`
  - position embedding 구현

- `segment.py`
  - segment embedding 구현
- `token.py`
  - token embedding 구현

position.py

position encoding은 기존의 transformer paper에 나왔던 trigonometric function과 동일합니다.

$$
\begin{matrix}
PE_{(pos, 2i)}=\sin(pos/10000^{2i/d_\text{model}})\
PE_{(pos, 2i+1)}=\cos(pos/10000^{2i/d_\text{model}})
\end{matrix}
$$

구현은 다음과 같이 했습니다.

```
pe = torch.zeros(max_len, d_model).float()
pe.require_grad = False # 학습 안되게


# pos => position / div_term => 10000^(2i/dmodel)
position = torch.arange(0, max_len).float().unsqueeze(1)
div_term = (torch.arange(0, d_model, 2).float() * -(math.log(10000.0) / d_model)).e

pe[:, 0::2] = torch.sin(position * div_term)
pe[:, 1::2] = torch.cos(position * div_term)

pe = pe.unsqueeze(0)
self.register_buffer('pe', pe)
```

근데 이부분이 애매한것은 논문을 보면 position encoding에 대한 다른 말 없이 다음과 같이 작성되어있습니다.

> We use learned positional embeddings with supported sequence lengths up to 512 tokens.

정확히 어떤 position encoding 기법을 썻는지 나와있지 않고 중요한 것은 learned로 나와있습니다.
google이 구현한 position encoding 부분을 보면 다음과 같습니다.

```
full_position_embeddings = tf.get_variable(
    name=position_embedding_name,
```

```
        shape=[max_position_embeddings, width],
        initializer=create_initializer(initializer_range))
```

google의 모델은 간단하게 random하게 initialize한 후 모델과 함께 학습하도록 만들어져있습니다.

segment.py

```
class SegmentEmbedding(nn.Embedding):
    def __init__(self, embed_size=512):
        super().__init__(3, embed_size, padding_idx=0)
```

nn.Embedding을 상속하고 총 vocab이 3개인 임베딩 메트릭스를 만들었습니다. 3인 이유는 seg A / seg B / pad 입니다.

token.py

```
class TokenEmbedding(nn.Embedding):
    def __init__(self, vocab_size, embed_size=512):
        super().__init__(vocab_size, embed_size, padding_idx=0)
```

마지막으로 token 임베딩은 다음과 같이 vocab_size에 맞게 구현되도록 만들었습니다.

BERTEmbedding

위 3개의 embedding을 하나로 합쳐 다음과 같이 구현했습니다.

```
class BERTEmbedding(nn.Module):
    """
    BERT Embedding which is consisted with under features
        1. TokenEmbedding : normal embedding matrix
        2. PositionalEmbedding : adding positional information using sin, cos
        2. SegmentEmbedding : adding sentence segment info, (sent_A:1, sent_B:2)
        sum of all these features are output of BERTEmbedding
    """

    def __init__(self, vocab_size, embed_size, dropout=0.1):
        """
        :param vocab_size: total vocab size
        :param embed_size: embedding size of token embedding
        :param dropout: dropout rate
        """
        super().__init__()
        self.token = TokenEmbedding(vocab_size=vocab_size, embed_size=embed_size)
```

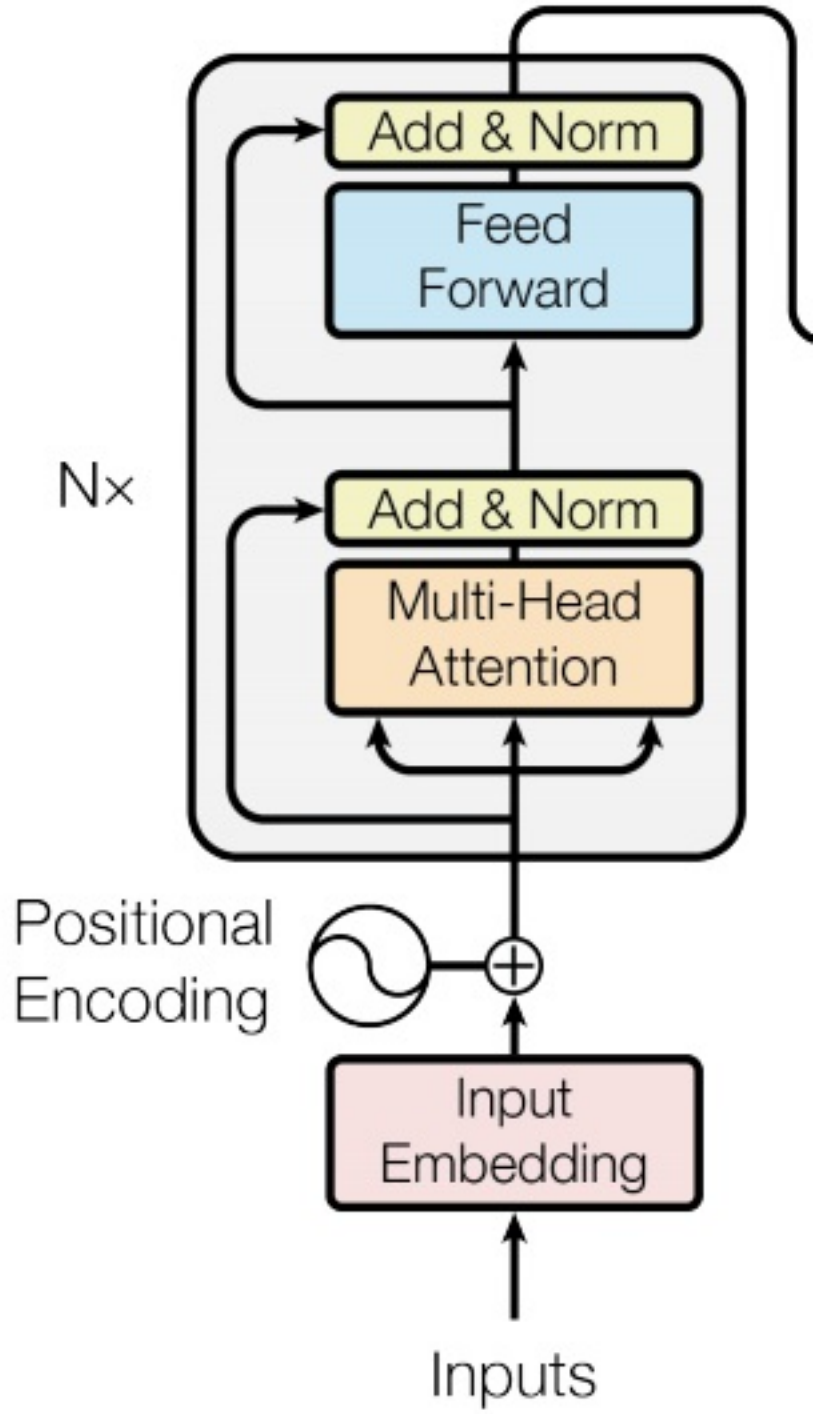
```
        self.position = PositionalEmbedding(d_model=self.token.embedding_dim)
        self.segment = SegmentEmbedding(embed_size=self.token.embedding_dim)
        self.dropout = nn.Dropout(p=dropout)
        self.embed_size = embed_size

    def forward(self, sequence, segment_label):
        x = self.token(sequence) + self.position(sequence) + self.segment(segment_l
        return self.dropout(x)
```

참고로 여기서는 마지막에 dropout만 했지만 google의 코드에서는 layer_norm + dropout을 했습니다.

6. Transformer Model

Transformer모델 부분을 위해서 총 4개의 모듈이 사용되었습니다.




```
self.attention = MultiHeadedAttention(h=attn_heads, d_model=hidden)
self.feed_forward = PositionwiseFeedForward(d_model=hidden, d_ff=feed_forward_hidde
self.input_sublayer = SublayerConnection(size=hidden, dropout=dropout)
self.output_sublayer = SublayerConnection(size=hidden, dropout=dropout)
self.dropout = nn.Dropout(p=dropout)
```

사용한 모듈 중 `Dropout`을 제외하고 구현한 모듈은 다음과 같습니다.

- `MultiHeadAttention`
- `SublayerConnection`
- `PositionwiseFeedForward`

6-1. `LayerNorm` & `SublayerConnection`

우선 sublayer를 먼저 살펴보면 sublayer는 세부적으로 `layernorm`, `sublayer` 모듈로 이루어져 있습니다.

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2


class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

layernorm의 경우 다음의 수식을 만족하도록 만들어줍니다.

$$
\frac{a(x-\mu)}{\sigma+\epsilon}+b
$$

여기서 a,b는 학습되는 상수이고 각각 $1,0$으로 초기화합니다. μ, ϵ의 경우에는 평균, 표준

편차입니다.
그리고 평균과 표준편차를 구할때는 layer에 대해서 구해야 하기 때문에 마지막 차원(-1)에 대해서 구합니다.

이후 sublayer에서 x에 normalize한 후 특정 layer(`sublayer`)를 거친 후 기존의 x와 더해지도록 합니다.
여기서의 `sublayer`는 적용할 때 feedforward 혹은 attention이 됩니다.

6-2 `PositionwiseFeedForward`

FeedForwar의 경우 추가적으로 활성화 함수이 `gelu`를 따로 구현해줍니다.
(gelu paper: https://arxiv.org/pdf/1606.08415.pdf)

```
class GELU(nn.Module):
    """
    Paper Section 3.4, last paragraph notice that BERT used the GELU instead of REL
    """

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi)
                            * (x + 0.044715 * torch.pow(x, 3))))
```

paper에 나온 gelu의 수식은 다음과 같습니다.

$$
0.5x(1+\tanh[\sqrt{2/\pi}(x+0.44715x^3)])
$$

gelu함수를 사용한 ff는 다음과 같습니다.

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = GELU()

    def forward(self, x):
        return self.w_2(self.dropout(self.activation(self.w_1(x))))
```

두개의 Dense Layer를 만들어 사용합니다.

6-3 `MultiHeadAttention`

마지막 `MultiHeadAttention`의 경우 2개의 모듈로 구성됩니다.

- `single.py`
- `multi_head.py`

single은 기존의 scaled dot-product attention이다.

```
class Attention(nn.Module):
    """
    Compute 'Scaled Dot Product Attention
    """

    def forward(self, query, key, value, mask=None, dropout=None):
        scores = torch.matmul(query, key.transpose(-2, -1)) \
                 / math.sqrt(query.size(-1))

        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        p_attn = F.softmax(scores, dim=-1)

        if dropout is not None:
            p_attn = dropout(p_attn)

        return torch.matmul(p_attn, value), p_attn
```

우선 qeury와 key를 transpose한것을 matrix multiply를 한 이후 qeury의 dimension의 sqrt값으로 나눠준다.

이후 결과값인 score에 masking을 하는 경우 10^-9으로 masking을 하고 softmax를 취해준다. 이 결과값에 value를 곱한 후 return한다.

이후 이 attention을 사용한 mutliheadattention은 다음과같다.

```
class MultiHeadedAttention(nn.Module):
    """
    Take in model size and number of heads.
    """

    def __init__(self, h, d_model, dropout=0.1):
        super().__init__()
        assert d_model % h == 0

        # We assume d_v always equals d_k
        self.d_k = d_model // h
```

```
        self.h = h

        self.linear_layers = nn.ModuleList([nn.Linear(d_model, d_model) for _ in ra
        self.output_linear = nn.Linear(d_model, d_model)
        self.attention = Attention()

        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [l(x).view(batch_size, -1, self.h, self.d_k).transpose(
                             for l, x in zip(self.linear_layers, (query, key, value

        # 2) Apply attention on all the projected vectors in batch.
        x, attn = self.attention(query, key, value, mask=mask, dropout=self.dropout

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous().view(batch_size, -1, self.h * self.d_k)
```

우선 3개의 Dense layer를 준비한다. 이 layer는 input을 각각 key, query, value로 만들어주는 layer이다. 그리고 output layer의 경우 최종 출력을 뽑기위한 layer로 dense layer를 하나더 준비한다.

그다음 forward함수에서 각각을 해당 dense layer를 적용시킨 후 `view` 함수를 이용해 기존의 `[bs, len, d_model]` 이였던 차원을 `[bs, len, h, d_k]` 로 만들어 준후 마지막으로 1,2 axis를 transpose해서 최종적으로 `[bs, h, len, d_k]` 로 만들어준다. 그 다음 각각의 값들을 사용해서 attention을 적용시킨 후 다시 마지막에 다시 역으로 `transpose` 함수, `contiguous` 함수, `view` 함수를 사용해 `[bs, len, d_model]` shape으로 맞춰준다.

이렇게 만들어진 모델을 사용해서 2개의 task를 학습하면서 pre-train을 진행하면 된다.
(학습 부분은 `trainer/pretrain.py` 에 구현되어 있지만, 사용하는 framework에 따라 매우 다른 부분이므로 다루지 않음)