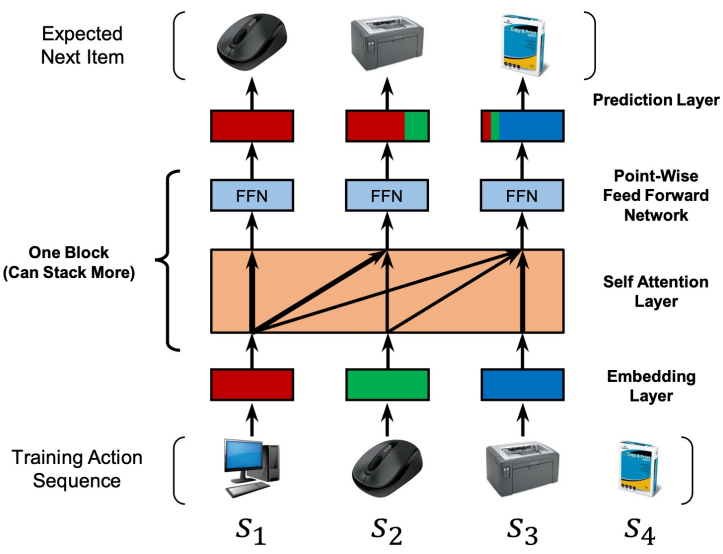
Self-Attentive Sequential Recommendation

Link : https://arxiv.org/pdf/1808.09781.pdf

작성자 : 이경찬



III. Methodology

In the setting of sequential recommendation, we are given a user's action sequence $\mathcal{S}^u = (\mathcal{S}_1^u, \mathcal{S}_2^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|}^u)$, and seek to predict the next item. During the training process, at time step t, the model predicts the next item depending on the previous t items. As shown in Figure 1, it will be convenient to think of the model's input as $(\mathcal{S}_1^u, \mathcal{S}_2^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|-1}^u)$ and its expected output as a 'shifted' version of the same sequence: $(\mathcal{S}_2^u, \mathcal{S}_3^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|}^u)$. In this section, we describe how we build a sequential recommendation model via an embedding layer, several self-attention blocks, and a prediction layer. We also analyze its complexity and further discuss how SASRec differs from related models. Our notation is summarized in Table I.

어느 유저 u의 session을 $\mathcal{S}^u$라고 한다면, $\mathcal{S}^u$는 다음과 같다:

$$\mathcal{S}^u = (\underbrace{\mathcal{S}_1^u, \mathcal{S}_2^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|-1}^u}_{\text{input}}, \overbrace{\mathcal{S}_{|\mathcal{S}^u|}^u}^{\text{output}})$$

시간 t일 때, 모델은 이전 t개의 아이템에 의존하여 다음 아이템을 예측한다.

A. Embedding Layer

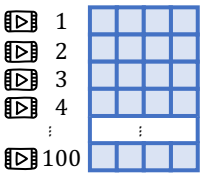
We transform the training sequence $(\mathcal{S}_1^u, \mathcal{S}_2^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|-1}^u)$ into a fixed-length sequence $s = (s_1, s_2, \ldots, s_n)$, where n represents the maximum length that our model can handle. If the sequence length is greater than n, we consider the most recent n actions. If the sequence length is less than n, we repeatedly add a 'padding' item to the left until the length is n. We create an item embedding matrix $\mathbf{M} \in \mathbb{R}^{|\mathcal{I}| \times d}$ where d is the latent dimensionality, and retrieve the input embedding matrix $\mathbf{E} \in \mathbb{R}^{n \times d}$, where $\mathbf{E}_i = \mathbf{M}_{s_i}$. A constant zero vector $\mathbf{0}$ is used as the embedding for the padding item.

모델이 다룰 수 있는 max length를 s라고 하자. 그럼 모든 input은 모두 s개의 시퀀스로 바뀌어야한다!

$$s = (s_1, s_2, \ldots, s_n)$$

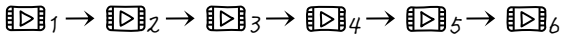
s개보다 길다면 최근 n개만 사용하고, 짧다면 왼쪽을 0으로 패딩한다!

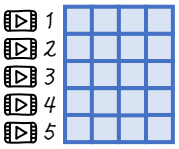
아이템 수 $|\mathcal{I}| = 100$,
embedding dimension=4 라고 한다면



Item embedding matrix M

input 아이템에 대해서 input embedding E를 만든다. 만약 input 아이템이 아래처럼 1→2→3→4→5→6 이라면 E는 5x4가 된다.





Input embedding matrix E

Positional Embedding: As we will see in the next section, since the self-attention model doesn't include any recurrent or convolutional module, it is not aware of the positions of previous items. Hence we inject a learnable position embedding $\mathbf{P} \in \mathbb{R}^{n \times d}$ into the input embedding:

$$\widehat{\mathbf{E}} = \begin{bmatrix} \mathbf{M}_{s_1} + \mathbf{P}_1 \\ \mathbf{M}_{s_2} + \mathbf{P}_2 \\ \cdots \\ \mathbf{M}_{s_n} + \mathbf{P}_n \end{bmatrix} \tag{1}$$

We also tried the fixed position embedding as used in [3], but found that this led to worse performance in our case. We analyze the effect of the position embedding quantitatively and qualitatively in our experiments.

Attention is all you need 논문처럼 fixed position embedding을 사용했더니 성능이 더 안좋았다고 한다.

B. *Self-Attention Block*

The scaled dot-product attention [3] is defined as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}, \qquad (2)$$

where $\mathbf{Q}$ represents the queries, $\mathbf{K}$ the keys and $\mathbf{V}$ the values (each row represents an item). Intuitively, the attention layer calculates a weighted sum of all values, where the weight between query i and value j relates to the interaction between query i and key j. The scale factor $\sqrt{d}$ is to avoid overly large values of the inner product, especially when the dimensionality is high.

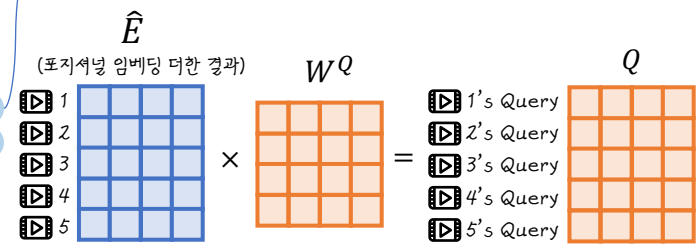
attention layer는 value의 weighted sum을 계산한다. 이 때의 weight는 query i와 key j간의 interaction과 관련있다. $\sqrt{d}$라는 scale factor의 역할은 dimension이 커졌을 때 내적값이 너무 커지지 않게 하기 위함이다!

Self-Attention layer: In NLP tasks such as machine translation, attention mechanisms are typically used with $\mathbf{K} = \mathbf{V}$ (e.g. using an RNN encoder-decoder for translation: the encoder's hidden states are keys and values, and the decoder's hidden states are queries) [28]. Recently, a self-attention method was proposed which uses the same objects as queries, keys, and values [3]. In our case, the self-attention operation takes the embedding $\widehat{\mathbf{E}}$ as input, converts it to three matrices through linear projections, and feeds them into an attention layer:

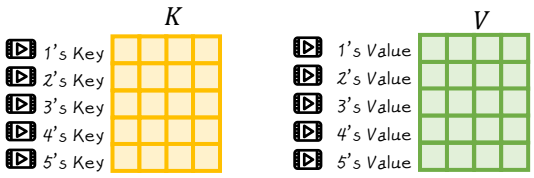
$$\mathbf{S} = \text{SA}(\widehat{\mathbf{E}}) = \text{Attention}(\widehat{\mathbf{E}}\mathbf{W}^Q, \widehat{\mathbf{E}}\mathbf{W}^K, \widehat{\mathbf{E}}\mathbf{W}^V), \qquad (3)$$

where the projection matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d\times d}$. The projections make the model more flexible. For example, the model can learn asymmetric interactions (i.e., <query i, key j> and <query j, key i> can have different interactions).

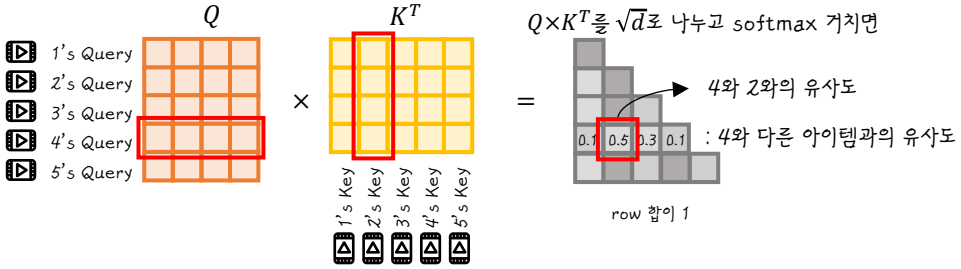
Attention is all you need에서 사용한 self-attention은 query, key, value를 모두 같다.

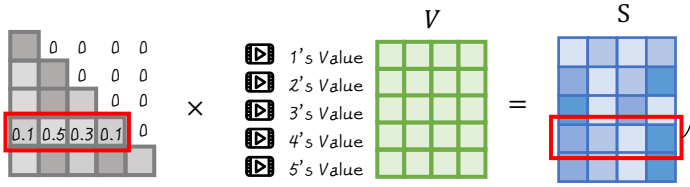


같은 식으로 W^K, W^V와 연산하면 K, V를 얻는다



projection matrices W^Q, W^K, W^V를 왜 쓸까?
바로 모델이 asymmetric interaction을 학습하여 flexible해지기 때문이다. 예를 들어 projection matrix가 없이 아이템 임베딩끼리만 attention을 계산한다면 i와 j의 내적은 어느 상황에서나 똑같겠지만, projection matrix가 있다면 i가 쿼리일 때, j가 쿼리일 때 처럼 다른 벡터간의 내적이 계산되어 상황마다 내적값이 달라진다.





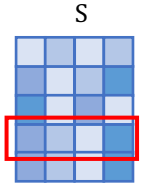
1, 2, 3, 4의 임베딩을 weighted sum해서 만들어짐
0.1*1의 value + 0.5*2의 value + 0.3*3의 value + 0.1*4의 value
아이템 2의 성분이 50%로 많이 포함되어 있다.

→ 이 벡터와 아이템 5와의 내적이 커지도록 학습됨!

인과관계

Causality: Due to the nature of sequences, the model should consider only the first t items when predicting the $(t+1)$-st item. However, the t-th output of the self-attention layer ($\mathbf{S}_t$) contains embeddings of subsequent items, which makes the model ill-posed. Hence, we modify the attention by forbidding all links between $\mathbf{Q}_i$ and $\mathbf{K}_j$ $(j > i)$.

모델은 t+1번째 아이템을 예측하기 위해서 오직 앞에 놓인 t개의 아이템들을 고려해야만 한다. 모델이 해서는 안되는 행동(?)은 t+1번째 아이템을 예측하기 위해 t+2번째 아이템을 고려대상에 포함시키는 것이다.



1~4의 아이템만 보고 아이템 5을 예측하도록 하기 위해 QK^T에서 j>i인 부분은 0으로 처리한다.

Point-Wise Feed-Forward Network: Though the self-attention is able to aggregate all previous items' embeddings with adaptive weights, ultimately it is still a linear model. To endow the model with nonlinearity and to consider interactions between different latent dimensions, we apply a point-wise two-layer feed-forward network to all $\mathbf{S}_i$ identically (sharing parameters):

$$\mathbf{F}_i = \text{FFN}(\mathbf{S}_i) = \text{ReLU}(\mathbf{S}_i\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \quad (4)$$

where $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}$ are $d \times d$ matrices and $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}$ are d-dimensional vectors. Note that there is no interaction between $\mathbf{S}_i$ and $\mathbf{S}_j$ $(i \neq j)$, meaning that we still prevent information leaks (from back to front).

Self-Attention은 결국 linear 모델이다. 비선형성을 부여하기 위해 2개의 feed-forward network를 적용한다.

S의 각 행인 S_i끼리의 interaction은 없도록 한다! 위에서 말했던 information leakage때문이다. 다만, feed-forward network의 파라미터는 공유한다.

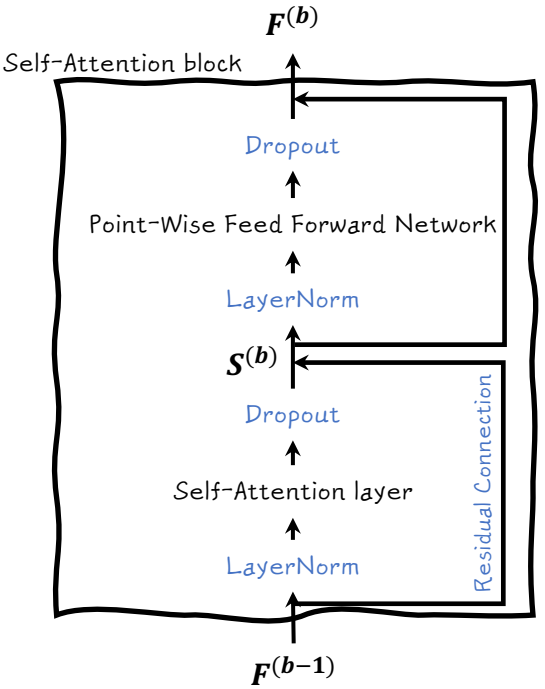
C. Stacking Self-Attention Blocks

Self-Attention block을 여러개 쌓는다

After the first self-attention block, $\mathbf{F}_i$ essentially aggregates all previous items' embeddings (i.e., $\widehat{\mathbf{E}}_j, j \leq i$). However, it might be useful to learn more complex item transitions via another self-attention block based on $\mathbf{F}$. Specifically, we stack the self-attention block (i.e., a self-attention layer and a feed-forward network), and the b-th $(b > 1)$ block is defined as:

$$\begin{aligned} \mathbf{S}^{(b)} &= \text{SA}(\mathbf{F}^{(b-1)}), \\ \mathbf{F}_i^{(b)} &= \text{FFN}(\mathbf{S}_i^{(b)}), \quad \forall i \in \{1, 2, \ldots, n\}, \end{aligned} \quad (5)$$

self-attention(+FNN)을 거치고 난 후, 더욱 복잡한 정보를 얻기 위해서는 한 번으론 부족하다. self-attention 블록을 여러 개 쌓는다.



and the 1-st block is defined as $\mathbf{S}^{(1)} = \mathbf{S}$ and $\mathbf{F}^{(1)} = \mathbf{F}$.

However, when the network goes deeper, several problems become exacerbated: 1) the increased model capacity leads to overfitting; 2) the training process becomes unstable (due to vanishing gradients etc.); and 3) models with more parameters often require more training time. Inspired by [3], We perform the following operations to alleviate these problems:

$$g(x) = x + \text{Dropout}(g(\text{LayerNorm}(x))),$$

where $g(x)$ represents the self attention layer or the feed-forward network. That is to say, for layer g in each block, we apply layer normalization on the input x before feeding into g, apply dropout on g's output, and add the input x to the final output. We introduce these operations below.

네트워크가 깊어지면 다음과 같은 문제점이 생긴다.

1) model capacity가 높아지면 오버피팅 가능성이 커진다.
2) vanishing gradient때문에 학습이 불안정해진다.
3) 학습 시간이 길어진다.

그래서 Dropout과 Layer Normalization, Residual connection을 적용시킴

Residual Connections: In some cases, multi-layer neural networks have demonstrated the ability to learn meaningful features hierarchically [34]. However, simply adding more layers did not easily correspond to better performance until residual networks were proposed [35]. The core idea behind residual networks is to propagate low-layer features to higher layers by residual connection. Hence, if low-layer features are useful, the model can easily propagate them to the final layer. Similarly, we assume residual connections are also useful in our case. For example, existing sequential recommendation methods have shown that the last visited item plays a key role on predicting the next item [1], [19], [21]. However, after several self-attention blocks, the embedding of the last visited item is entangled with all previous items; adding residual connections to propagate the last visited item's embedding to the final layer would make it much easier for the model to leverage low-layer information.

Residual network가 제안되기 전까진 단순히 Layer를 더한다고 성능이 무조건 향상되진 않았었다

low-layer feature를 high-layer feature에 더하면, low-layer feature가 유용한 경우 모델은 쉽게 예측할 수 있다.

마지막 아이템이 중요한 것은 이미 많이 알려져있지만, 여러 self-attention block을 거치고 나면 마지막 아이템 임베딩에는 이전 아이템들 정보가 많이 섞여있다. 그래서 Residual connection을 통해 low layer 정보를 주면 모델이 더 쉽게 예측할 수 있다.

Layer Normalization: Layer normalization is used to normalize the inputs across features (i.e., zero-mean and unit-variance), which is beneficial for stabilizing and accelerating neural network training [36]. Unlike batch normalization [37], the statistics used in layer normalization are independent of other samples in the same batch. Specifically, assuming the input is a vector $\mathbf{x}$ which contains all features of a sample, the operation is defined as:

[36]Layer Normalization

$$\text{LayerNorm}(\mathbf{x}) = \boldsymbol{\alpha} \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \boldsymbol{\beta},$$

where $\odot$ is an element-wise product (i.e., the Hadamard product), μ and σ are the mean and variance of $\mathbf{x}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are learned scaling factors and bias terms.

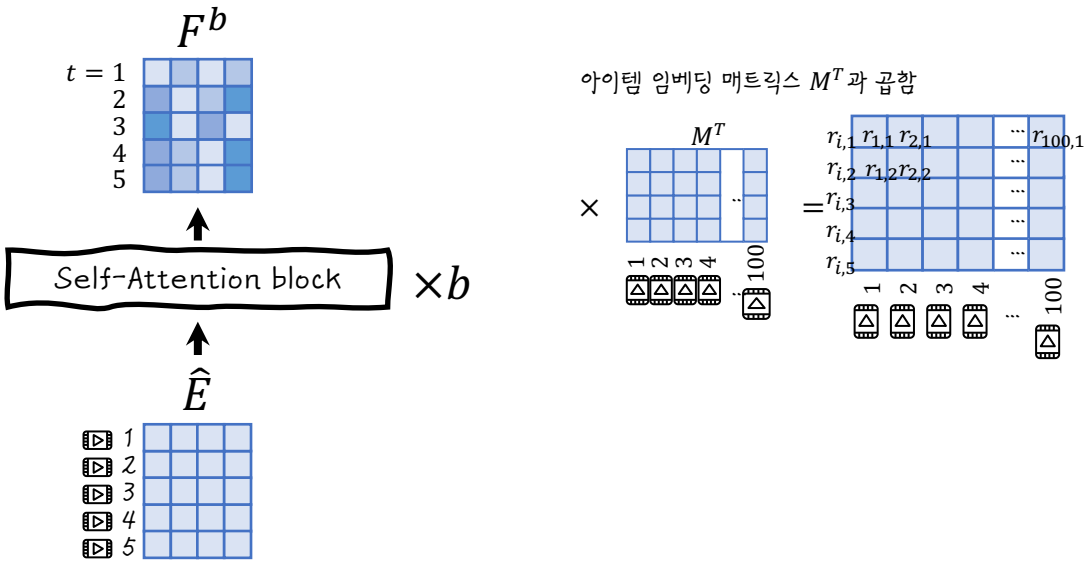
Dropout: To alleviate overfitting problems in deep neural networks, 'Dropout' regularization techniques have been shown to be effective in various neural network architectures [38]. The idea of dropout is simple: randomly 'turn off' neurons with probability p during training, and use all neurons when testing. Further analysis points out that dropout can be viewed as a form of ensemble learning which considers an enormous number of models (exponential in the number of neurons and input features) that share parameters [39]. We also apply a dropout layer on the embedding $\widehat{\mathbf{E}}$.

D. Prediction Layer

After b self-attention blocks that adaptively and hierarchically extract information of previously consumed items, we predict the next item (given the first t items) based on $\mathbf{F}_t^{(b)}$. Specifically, we adopt an MF layer to predict the relevance of item i:

$$r_{i,t} = \mathbf{F}_t^{(b)} \mathbf{N}_i^T,$$

where $r_{i,t}$ is the relevance of item i being the next item given the first t items (i.e., $s_1, s_2, \ldots, s_t$), and $\mathbf{N} \in \mathbb{R}^{|\mathcal{I}| \times d}$ is an item embedding matrix. Hence, a high interaction score $r_{i,t}$ means a high relevance, and we can generate recommendations by ranking the scores.



Shared Item Embedding: To reduce the model size and alleviate overfitting, we consider another scheme which only uses a single item embedding $\mathbf{M}$:

$$r_{i,t} = \mathbf{F}_t^{(b)} \mathbf{M}_i^T. \tag{6}$$

Note that $\mathbf{F}_t^{(b)}$ can be represented as a function depending on the item embedding $\mathbf{M}$: $\mathbf{F}_t^{(b)} = f(\mathbf{M}_{s_1}, \mathbf{M}_{s_2}, \ldots, \mathbf{M}_{s_t})$. A potential issue of using homogeneous item embeddings is that their inner products cannot represent asymmetric item transitions (e.g. item i is frequently bought after j, but not vise versa), and thus existing methods like FPMC tend to use heterogeneous item embeddings. However, our model doesn't have this issue since it learns a nonlinear transformation. For example, the feed forward network can easily achieve the asymmetry with the same item embedding: $\text{FFN}(\mathbf{M}_i)\mathbf{M}_j^T \neq \text{FFN}(\mathbf{M}_j)\mathbf{M}_i^T$. Empirically, using a shared item embedding significantly improves the performance of our model.

Explicit User Modeling: To provide personalized recommendations, existing methods often take one of two approaches: 1) learn an *explicit* user embedding representing user preferences (e.g. MF [40], FPMC [1] and Caser [22]); 2) consider the user's previous actions, and induce an *implicit* user embedding from embeddings of visited items (e.g. FSIM [8], Fossil [21], GRU4Rec [2]). Our method falls into the latter category, as we generate an embedding $\mathbf{F}_n^{(b)}$ by considering all actions of a user. However, we can also insert an explicit user embedding at the last layer, for example via addition: $r_{u,i,t} = (\mathbf{U}_u + \mathbf{F}_t^{(b)})\mathbf{M}_i^T$ where $\mathbf{U}$ is a user embedding matrix. However, we empirically find that adding an explicit user embedding doesn't improve performance (presumably because the model already considers all of the user's actions).

E. Network Training

Recall that we convert each user sequence (excluding the last action) $(\mathcal{S}_1^u, \mathcal{S}_2^u, \ldots, \mathcal{S}_{|\mathcal{S}^u|-1}^u)$ to a fixed length sequence $s = \{s_1, s_2, \ldots, s_n\}$ via truncation or padding items. We define o_t as the expected output at time step t:

$$o_t = \begin{cases} \texttt{<pad>} & \text{if } s_t \text{ is a padding item} \\ s_{t+1} & 1 \le t < n \\ \mathcal{S}_{|\mathcal{S}^u|}^u & t = n \end{cases},$$

where <pad> indicates a padding item. Our model takes a sequence s as input, the corresponding sequence o as expected output, and we adopt the binary cross entropy loss as the objective function:

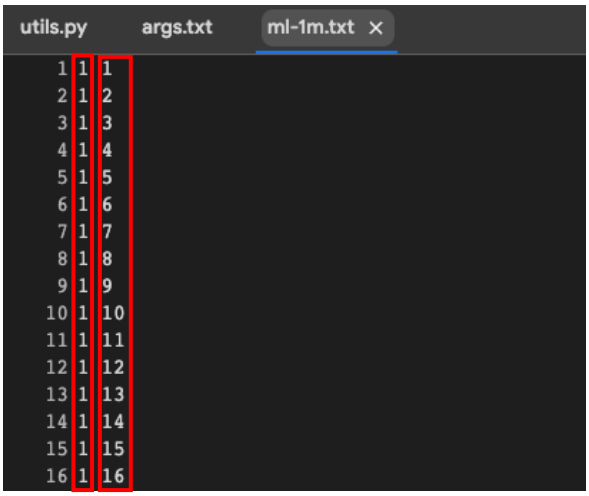
$$-\sum_{\mathcal{S}^u \in \mathcal{S}} \sum_{t \in [1,2,\ldots,n]} \left[\log(\sigma(r_{o_t,t})) + \sum_{j \notin \mathcal{S}^u} \log(1 - \sigma(r_{j,t})) \right].$$

Note that we ignore the terms where $o_t =$ <pad>.

The network is optimized by the *Adam* optimizer [41], which is a variant of Stochastic Gradient Descent (SGD) with adaptive moment estimation. In each epoch, we randomly generate one negative item j for each time step in each sequence. More implementation details are described later.

전처리방식 이해

ml-1m.txt



유저 아이템

User라는 defaultdict(list)에 한 줄씩 읽으며 append

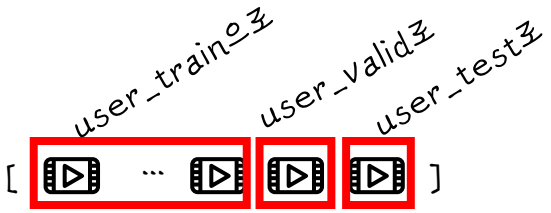
```
User[u].append(i)
```

길이가 1, 2인 시퀀스는

user_train[1] = [ ] 식으로 넣고

user_valid[1] = [], user_test[1] = [] 식으로 빈 리스트로.

길이가 3 이상인 시퀀스는



WarpSampler 이해

```
from multiprocessing import Process, Queue
```

```
class WarpSampler(object):
    def __init__(self, User, usernum, itemnum, batch_size=64, maxlen=10, n_workers=1):
        self.result_queue = Queue(maxsize=n_workers * 10)
        self.processors = []
        for i in range(n_workers):
            self.processors.append(
                Process(target=sample_function, args=(User,
                                                      usernum,
                                                      itemnum,
                                                      batch_size,
                                                      maxlen,
                                                      self.result_queue,
                                                      np.random.randint(2e9)
                                                      )))
            self.processors[-1].daemon = True
            self.processors[-1].start()

    def next_batch(self):
        return self.result_queue.get()

    def close(self):
        for p in self.processors:
            p.terminate()
            p.join()
```

result_queue에 들어가는 항목의 수 상한

worker 수

WarpSampler.processors = [Process , Process , ··· , Process]

Process : user_train에 대해 sample_function() 실행

sample_function() 한 번 실행하면 self.result_queue에 배치가 쌓임

process.start() : 프로세스의 활동을 시작합니다. 이것은 프로세스 객체 당 최대 한 번 호출되어야 합니다.

process.join() : 작업자 프로세스가 종료될 때까지 기다립니다. join() 호출 전에 반드시 close() 나 terminate()를 호출해야합니다 .

```
def sample_function(user_train, usernum, itemnum, batch_size, maxlen, result_queue, SEED):
    def sample():
        user = np.random.randint(1, usernum + 1)
        while len(user_train[user]) <= 1:
            user = np.random.randint(1, usernum + 1)

        # ex : user_train[user] = [1, 2, 3, 4, 5], max_len = 50
        seq = np.zeros([maxlen], dtype=np.int32)
        pos = np.zeros([maxlen], dtype=np.int32)
        neg = np.zeros([maxlen], dtype=np.int32)
        nxt = user_train[user][-1]  # 마지막 아이템부터 시작 ex) 5
        idx = maxlen - 1  # 끝자리

        ts = set(user_train[user])  # True items. =positive items
        for i in reversed(user_train[user][:-1]):  # 마지막 아이템 빼고 거꾸로 돌면서
            seq[idx] = i  # 맨 뒤부터 채움 ex) seq : [0, 0, ..45개 0.., 0, 0, 4]
            pos[idx] = nxt  # 마지막 아이템을 맨 뒤부터 채움 pos : [0, 0, ..45개 0.., 0, 5]
            if nxt != 0:
                neg[idx] = random_neq(1, itemnum + 1, ts)  # neg : [0, 0, ..45개 0.., 0, 2832]
            nxt = i  # nxt : 4
            idx -= 1  # idx : 48
            if idx == -1:
                break
        return (user, seq, pos, neg)  # seq : [0, 0, ..., 0, 0, 1, 2, 3, 4] -> 길이 50

    np.random.seed(SEED)
    while True:
        one_batch = []
        for i in range(batch_size):
            one_batch.append(sample())

        result_queue.put(zip(*one_batch))
```

sample_function

임의의 유저 뽑음 → (user, seq, pos, neg) 튜플 생성

ex) user 3이 뽑혔고 item sequence가 1, 2, 3, 4, 5라면
- user : 3
- seq : [0, 0, ···, 0, 0, 1, 2, 3, 4] → 수 : max_len
- pos : [0, 0, ···, 0, 0, 2, 3, 4, 5]
- neg : [0, 0, ···, 0, 0, 1251, 324, 2352, 6123]

배치사이즈 128

one_batch = [(user, seq, pos, neg), (), ···, ()]

sampler.result_queue.put(zip(*one_batch))

torch.nn.BCEWithLogitsLoss() 이해

BCEWITHLOGITSLOSS

CLASS torch.nn.BCEWithLogitsLoss(*weight=None, size_average=None, reduce=None, reduction='mean', pos_weight=None*) [SOURCE]

Sigmoid Layer와 BCELoss를 하나로 합쳐놓은 클래스!

This loss combines a *Sigmoid* layer and the *BCELoss* in one single class. This version is more numerically stable than using a plain *Sigmoid* followed by a *BCELoss* as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top,$$

$$l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets `t[i]` should be numbers between 0 and 1.

It's possible to trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$\ell_c(x, y) = L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top,$$

$$l_{n,c} = -w_{n,c} \left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c})) \right],$$

where c is the class number ($c > 1$ for multi-label binary classification, $c = 1$ for single-label binary classification), n is the number of the sample in the batch and p_c is the weight of the positive answer for the class c.

$p_c > 1$ increases the recall, $p_c < 1$ increases the precision.

For example, if a dataset contains 100 positive and 300 negative examples of a single class, then *pos_weight* for the class should be equal to $\frac{300}{100} = 3$. The loss would act as if the dataset contains $3 \times 100 = 300$ positive examples.

논문에서의 목적 함수와 비교해보자. 논문에서는 아래처럼,

$$-\sum_{\mathcal{S}^u \in \mathcal{S}} \sum_{t \in [1,2,\dots,n]} \left[\log(\sigma(r_{o_t,t})) + \sum_{j \notin \mathcal{S}^u} \log(1 - \sigma(r_{j,t})) \right]$$

Binary Cross Entropy를 쓴다고 했었다.

E. Network Training

Recall that we convert each user sequence (excluding the last action) $(\mathcal{S}^u_1, \mathcal{S}^u_2, \dots, \mathcal{S}^u_{|\mathcal{S}^u|-1})$ to a fixed length sequence $s = \{s_1, s_2, \dots, s_n\}$ via truncation or padding items. We define o_t as the expected output at time step t:

$$o_t = \begin{cases} \texttt{<pad>} & \text{if } s_t \text{ is a padding item} \\ s_{t+1} & 1 \le t < n \\ \mathcal{S}^u_{|\mathcal{S}^u|} & t = n \end{cases},$$

where `<pad>` indicates a padding item. Our model takes a sequence s as input, the corresponding sequence o as expected

torch.nn.init.xavier_normal_() 이해

원논문 **Understanding the difficulty of training deep feedforward neural networks**

torch.nn.init.xavier_normal_(*tensor*, *gain=1.0*) [SOURCE]

Fills the input *Tensor* with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks* - Glorot, X. & Bengio, Y. (2010), using a normal distribution. The resulting tensor will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where

$$\text{std} = \text{gain} \times \sqrt{\frac{2}{\text{fan\_in} + \text{fan\_out}}}$$

fan_in : 이전 layer(input)의 노드 수
fan_out : 다음 layer의 노드 수

Also known as Glorot initialization.

Parameters:

- **tensor** (*Tensor*) – an n-dimensional *torch.Tensor*
- **gain** (*float*) – an optional scaling factor

Return type:

Tensor

Examples

```
>>> w = torch.empty(3, 5)
>>> nn.init.xavier_normal_(w)
```

텐서

```
for name, param in model.named_parameters():
    try:
        torch.nn.init.xavier_normal_(param.data)
        print(f"{name:50} : {param.shape}")
    except:
        pass # just ignore those failed init layers
        print(f"{name:50} : {param.shape}\t<-- failed")
```

```
item_emb.weight                                    : torch.Size([3417, 50])
pos_emb.weight                                     : torch.Size([50, 50])
attention_layernorms.0.weight                      : torch.Size([50])   <-- failed
attention_layernorms.0.bias                        : torch.Size([50])   <-- failed
attention_layernorms.1.weight                      : torch.Size([50])   <-- failed
attention_layernorms.1.bias                        : torch.Size([50])   <-- failed
attention_layers.0.in_proj_weight                  : torch.Size([150, 50])
attention_layers.0.in_proj_bias                    : torch.Size([150])  <-- failed
attention_layers.0.out_proj.weight                 : torch.Size([50, 50])
attention_layers.0.out_proj.bias                   : torch.Size([50])   <-- failed
attention_layers.1.in_proj_weight                  : torch.Size([150, 50])
attention_layers.1.in_proj_bias                    : torch.Size([150])  <-- failed
attention_layers.1.out_proj.weight                 : torch.Size([50, 50])
attention_layers.1.out_proj.bias                   : torch.Size([50])   <-- failed
forward_layernorms.0.weight                        : torch.Size([50])   <-- failed
forward_layernorms.0.bias                          : torch.Size([50])   <-- failed
forward_layernorms.1.weight                        : torch.Size([50])   <-- failed
forward_layernorms.1.bias                          : torch.Size([50])   <-- failed
forward_layers.0.conv1.weight                      : torch.Size([50, 50, 1])
forward_layers.0.conv1.bias                        : torch.Size([50])   <-- failed
forward_layers.0.conv2.weight                      : torch.Size([50, 50, 1])
forward_layers.0.conv2.bias                        : torch.Size([50])   <-- failed
forward_layers.1.conv1.weight                      : torch.Size([50, 50, 1])
forward_layers.1.conv1.bias                        : torch.Size([50])   <-- failed
forward_layers.1.conv2.weight                      : torch.Size([50, 50, 1])
forward_layers.1.conv2.bias                        : torch.Size([50])   <-- failed
last_layernorm.weight                              : torch.Size([50])   <-- failed
last_layernorm.bias                                : torch.Size([50])   <-- failed
```

2차원 이상의 텐서만 xavier normalization이 가능하다.

torch.nn.MultiheadAttention 모듈 및 SASRec의 forward에 대한 이해

SASRec의 __init__ 중

2

```
for _ in range(args.num_blocks):
    new_attn_layernorm = torch.nn.LayerNorm(args.hidden_units, eps=1e-8)
    self.attention_layernorms.append(new_attn_layernorm)

    new_attn_layer = torch.nn.MultiheadAttention(args.hidden_units,
                                                 args.num_heads,
                                                 args.dropout_rate)
    self.attention_layers.append(new_attn_layer)
```

arguments
hidden_units : 50
num_heads : 1
dropout_rate : 0.5

torch.nn.ModuleList()

```
SASRec(
  (item_emb): Embedding(3417, 50, padding_idx=0)
  (pos_emb): Embedding(50, 50)
  (emb_dropout): Dropout(p=0.5, inplace=False)
  (attention_layernorms): ModuleList(
    (0-1): 2 x LayerNorm((50,), eps=1e-08, elementwise_affine=True)
  )
  (attention_layers): ModuleList(
    (0-1): 2 x MultiheadAttention(
      (out_proj): NonDynamicallyQuantizableLinear(in_features=50, out_features=50, bias=True)
    )
  )
  (forward_layernorms): ModuleList(
    (0-1): 2 x LayerNorm((50,), eps=1e-08, elementwise_affine=True)
  )
  (forward_layers): ModuleList(
    (0-1): 2 x PointWiseFeedForward(
      (conv1): Conv1d(50, 50, kernel_size=(1,), stride=(1,))
      (dropout1): Dropout(p=0.5, inplace=False)
      (relu): ReLU()
      (conv2): Conv1d(50, 50, kernel_size=(1,), stride=(1,))
      (dropout2): Dropout(p=0.5, inplace=False)
    )
  )
  (last_layernorm): LayerNorm((50,), eps=1e-08, elementwise_affine=True)
)
```

[torch.nn.MultiheadAttention](#)의 설명은 다음과 같다

MULTIHEADATTENTION

CLASS torch.nn.MultiheadAttention(*embed_dim*, *num_heads*, *dropout=0.0*, *bias=True*, *add_bias_kv=False*, *add_zero_attn=False*, *kdim=None*, *vdim=None*, *batch_first=False*, *device=None*, *dtype=None*) [SOURCE]

Allows the model to jointly attend to information from different representation subspaces as described in the paper: Attention Is All You Need.

Multi-Head Attention is defined as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \dots, head_h)W^O$$

Multi-head attention은 여러 head를 concat한 후 W^O를 곱한다

where $head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

`forward()` will use the optimized implementations of `scaled_dot_product_attention()`.

scaled dot product attention을 계산한다.

MultiheadAttention의 __init__ 중

```python
def __init__(self, embed_dim, num_heads, dropout=0., bias=True, add_bias_kv=False, add_zero_attn=False,
             kdim=None, vdim=None, batch_first=False, device=None, dtype=None) -> None:
    factory_kwargs = {'device': device, 'dtype': dtype}
    super().__init__()
    self.embed_dim = embed_dim
    self.kdim = kdim if kdim is not None else embed_dim
    self.vdim = vdim if vdim is not None else embed_dim
    self._qkv_same_embed_dim = self.kdim == embed_dim and self.vdim == embed_dim

    self.num_heads = num_heads
    self.dropout = dropout
    self.batch_first = batch_first
    self.head_dim = embed_dim // num_heads
    assert self.head_dim * num_heads == self.embed_dim, "embed_dim must be divisible by num_heads"

    if not self._qkv_same_embed_dim:
        self.q_proj_weight = Parameter(torch.empty((embed_dim, embed_dim), **factory_kwargs))
        self.k_proj_weight = Parameter(torch.empty((embed_dim, self.kdim), **factory_kwargs))
        self.v_proj_weight = Parameter(torch.empty((embed_dim, self.vdim), **factory_kwargs))
        self.register_parameter('in_proj_weight', None)
    else:
        self.in_proj_weight = Parameter(torch.empty((3 * embed_dim, embed_dim), **factory_kwargs))
        self.register_parameter('q_proj_weight', None)
        self.register_parameter('k_proj_weight', None)
        self.register_parameter('v_proj_weight', None)

    if bias:
        self.in_proj_bias = Parameter(torch.empty(3 * embed_dim, **factory_kwargs))
    else:
        self.register_parameter('in_proj_bias', None)
    self.out_proj = NonDynamicallyQuantizableLinear(embed_dim, embed_dim, bias=bias, **factory_kwargs)

    if add_bias_kv:
        self.bias_k = Parameter(torch.empty((1, 1, embed_dim), **factory_kwargs))
        self.bias_v = Parameter(torch.empty((1, 1, embed_dim), **factory_kwargs))
    else:
        self.bias_k = self.bias_v = None

    self.add_zero_attn = add_zero_attn

    self._reset_parameters()
```

(if not self._qkv_same_embed_dim:) key, query, value가 모두 같은 차원일 때만 보자. 그럼 건너뛰고

(if bias:) bias는 디폴트 True이므로 self.in_proj_bias를 갖는다.

(self.bias_k = self.bias_v = None) add_bias_kv는 디폴트 False이므로 key, value의 bias를 따로 두지 않는다

torch.nn.MultiheadAttention의 소스코드(torch github은 https://github.com/pytorch/pytorch)를 들여다보면

Key, Query, Value 차원이 모두 같다고 하고 보자. 그럼 _qkv_same_embed_dim은 True.

head_dim이라는 head의 차원은 embed_dim // num_heads로 한다. 우리는 embed_dim=50, num_heads=1이므로 head_dim=2가 되겠다.

self. register_parameter(name, param) 메소드는 MultiheadAttention이 상속받은 nn.Module의 메소드로서, parameter를 name을 이용해 attribute로 접근할 수 있게 한다.

'q_proj_weight'라는 파라미터 안쓰겠다는 거다

torch.empty(size)에서 size는 integer의 시퀀스로서, size형태의 텐서를 반환하며 값들은 초기화되어 있지 않다.

Parmeter는 Tensor의 서브클래스이다. Parameter는 Module과 같이 쓰일 때 특별한 property를 갖게 된다. Parameter가 Module의 attribute로 할당되면, Module의 parameter 리스트에 자동으로 리스팅되며, parameter()라는 iterator에서 등장한다.

또, self.out_proj라는 50차원 -> 50차원의 Linear 계층을 갖게 된다. (NonDynamicallyQuant..클래스는 Linear의 서브클래스다)

self._reset_parameters()는 self.in_proj_weight를 xavier_uniform으로 초기화하고, self.in_proj_bias, self.out_proj의 bias를 0으로 초기화한다.

```
1 model.attention_layers[0].in_proj_bias

Parameter containing:
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0.], requires_grad=True)
```

```
1 model.attention_layers[0].out_proj.bias

Parameter containing:
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0.], requires_grad=True)
```

아무튼, MultiheadAttention은 self.in_proj_weight라는 Parameter, 즉 shape은 (150, 50)인 이라는 텐서를 인스턴스로 갖게 된다.

```
[47]  1 model.attention_layers[0].in_proj_weight.shape

torch.Size([150, 50])
```

```
[46]  1 model.attention_layers[0].in_proj_weight

Parameter containing:
tensor([[-0.1077,  0.0398,  0.1361,  ...,  0.1186,  0.0837, -0.0804],
        [-0.1213, -0.1711, -0.0530,  ..., -0.1129,  0.1201, -0.1099],
        [ 0.0654,  0.0287,  0.0755,  ...,  0.0036,  0.1394,  0.0496],
        ...,
        [ 0.0004, -0.0212,  0.1702,  ...,  0.1018, -0.1508, -0.0862],
        [ 0.0287,  0.1489,  0.1348,  ...,  0.0787, -0.0072, -0.0654],
        [-0.0077, -0.0908, -0.1179,  ..., -0.1024, -0.0060,  0.0925]],
       requires_grad=True)
```

```
1 model.attention_layers[0].out_proj

NonDynamicallyQuantizableLinear(in_features=50, out_features=50, bias=True)
```

MultiheadAttention의 forward

```
def forward(
        self,
        query: Tensor,
        key: Tensor,
        value: Tensor,
        key_padding_mask: Optional[Tensor] = None,
        need_weights: bool = True,
        attn_mask: Optional[Tensor] = None,
        average_attn_weights: bool = True,
        is_causal: bool = False) -> Tuple[Tensor, Optional[Tensor]]:
```

query, key, value를 받는다.

잠깐! 그럼 SASRec의 MultiheadAttention에는 forward의 input이 뭘까?

```
def forward(self, user_ids, log_seqs, pos_seqs, neg_seqs):  # for training
    log_feats = self.log2feats(log_seqs)  # user_ids hasn't been used yet
```

log_seqs는 일단 배치고, 한 example은 user가 사용한 아이템이 1, 2, 3, 4, 5라면 [0, 0, ⋯, 0, 0, 1, 2, 3, 4]처럼 패딩된 아이템 시퀀스다.

```
def log2feats(self, log_seqs):
    seqs = self.item_emb(torch.LongTensor(log_seqs).to(self.dev))
    seqs *= self.item_emb.embedding_dim ** 0.5
    positions = np.tile(np.array(range(log_seqs.shape[1])), [log_seqs.shape[0], 1])
    seqs += self.pos_emb(torch.LongTensor(positions).to(self.dev))
    seqs = self.emb_dropout(seqs)

    timeline_mask = torch.BoolTensor(log_seqs == 0).to(self.dev)
    seqs *= ~timeline_mask.unsqueeze(-1)  # broadcast in last dim
50
    tl = seqs.shape[1]  # time dim len for enforce causality
    attention_mask = ~torch.tril(torch.ones((tl, tl), dtype=torch.bool, device=self.dev))

    for i in range(len(self.attention_layers)):
        seqs = torch.transpose(seqs, 0, 1)
        Q = self.attention_layernorms[i](seqs)
        mha_outputs, _ = self.attention_layers[i](Q, seqs, seqs,
                                                  attn_mask=attention_mask)
        # key_padding_mask=timeline_mask
        # need_weights=False) this arg do not work?
        seqs = Q + mha_outputs
        seqs = torch.transpose(seqs, 0, 1)

        seqs = self.forward_layernorms[i](seqs)
        seqs = self.forward_layers[i](seqs)
        seqs *= ~timeline_mask.unsqueeze(-1)

    log_feats = self.last_layernorm(seqs)  # (U, T, C) -> (U, -1, C)

    return log_feats
```

아이템 임베딩을 뽑는다

50에 루트 씌우니까 7정도. 이를 위 임베딩에 곱함

positions

```
1 positions

array([[ 0,  1,  2, ..., 47, 48, 49],
       [ 0,  1,  2, ..., 47, 48, 49],
       [ 0,  1,  2, ..., 47, 48, 49],
       ...,
       [ 0,  1,  2, ..., 47, 48, 49],
       [ 0,  1,  2, ..., 47, 48, 49],
       [ 0,  1,  2, ..., 47, 48, 49]])

[57] 1 positions.shape

(128, 50)
```

positional embedding 더하고 dropout

시퀀스에서 0인 아이템 자리는 True. 위의 예를 보면 timeline_mask = [True, True, ⋯ , True, False, False, False, False, False]

```
[70] 1 ~timeline_mask

tensor([[False, False, False, ..., True, True, True],
        [False, False, False, ..., True, True, True],
        [ True,  True,  True, ..., True, True, True],
        ...,
        [ True,  True,  True, ..., True, True, True],
        [ True,  True,  True, ..., True, True, True],
        [False, False, False, ..., True, True, True]])

1 ~timeline_mask.unsqueeze(-1)

tensor([[[False],
         [False],
         [False],
         ...,
         [ True],
         [ True],
         [ True]],

        [[False],
         [False],
         [False],
         ...,
         [ True],
         [ True],
         [ True]],

        [[ True],
         [ True],
         [ True],
         ...,
         [ True],
         [ True],
         [ True]],

        ...,

        [[ True],
```

한 example에 대한 seqs는 50 by 50 (max_seq_len by embed_dim)일텐데, 위의 예시로 보면 0~45 row는 패딩아이템 0의 임베딩임. 이를 다 0으로 만들어버려!

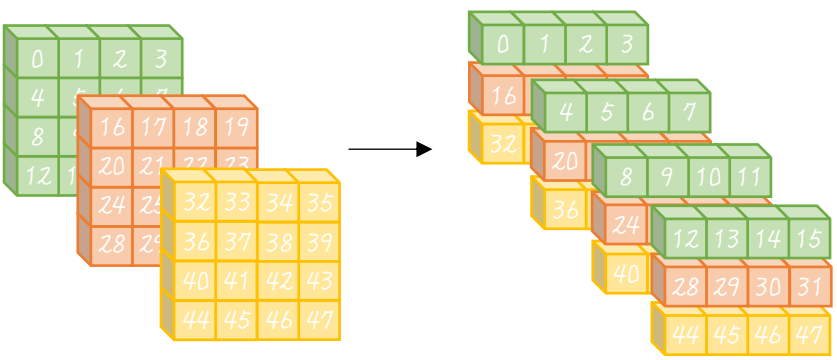
앞의 46개 패딩아이템은 0을 곱함. False를 곱함.

casuality(인과관계)라는 단어가 논문에서와 마찬가지로 또 나왔다. 시간상 뒤에 쿨직된 아이템이 앞에 쿨직된 아이템을 예측하는데 쓰이지 못하도록 한다

attention_mask : torch.tril은 lower triangular matrix를 반환한다. low에(대각포함) 1이 있는 행결을 ~로 반대로 뒤집으니까, 대각빼고 upper 부분에 1인 행렬이 attention_mask가 된다.

LayerNormalization을 거치게 하여 Query를 얻는다! Q's shape : (50, 128, 50)

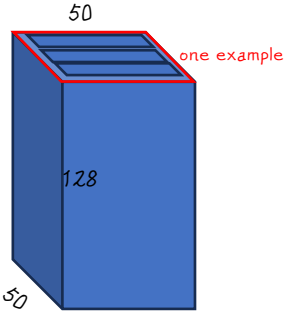
torch.transpose(tensor, dim1, dim2)는 tensor의 dim1차원과 dim2차원을 바꾼다.



seqs → torch.transpose(seqs, 0, 1)

```
[97] 1 seqs[0]   timeline mask를 곱하기 전후를 비교해보자.

tensor([[ 0.1416, -0.0450, -0.3449,  ...,  0.0411, -0.2848, -0.0840],
        [ 0.1416, -0.0450, -0.3449,  ...,  0.0411, -0.2848, -0.0840],
        [ 0.1416, -0.0450, -0.3449,  ...,  0.0411, -0.2848, -0.0840],
        ...,
        [-0.1515,  0.0820, -0.1521,  ...,  0.0450, -0.2075, -0.1690],
        [ 0.0361,  0.2827,  0.1651,  ..., -0.0030, -0.0713,  0.1172],
        [ 0.0227, -0.0636, -0.0312,  ..., -0.0462, -0.3081,  0.1393]],
       grad_fn=<SelectBackward0>)

[98] 1 seqs *= ~timeline_mask.unsqueeze(-1)  # broadcast in last dim

1 seqs[0]

tensor([[ 0.0000, -0.0000, -0.0000,  ...,  0.0000, -0.0000, -0.0000],
        [ 0.0000, -0.0000, -0.0000,  ...,  0.0000, -0.0000, -0.0000],
        [ 0.0000, -0.0000, -0.0000,  ...,  0.0000, -0.0000, -0.0000],
        ...,
        [-0.1515,  0.0820, -0.1521,  ...,  0.0450, -0.2075, -0.1690],
        [ 0.0361,  0.2827,  0.1651,  ..., -0.0030, -0.0713,  0.1172],
        [ 0.0227, -0.0636, -0.0312,  ..., -0.0462, -0.3081,  0.1393]],
       grad_fn=<SelectBackward0>)
```

그래서 지금 Q, seqs의 shape는 다음과 같다.



MultiheadAttention의 forward로 다시 돌아가보자.

```python
def forward(
        self,
        query: Tensor,
        key: Tensor,
        value: Tensor,
        key_padding_mask: Optional[Tensor] = None,
        need_weights: bool = True,
        attn_mask: Optional[Tensor] = None,
        average_attn_weights: bool = True,
        is_causal: bool = False) -> Tuple[Tensor, Optional[Tensor]]:
```

(50, 128, 50)의 query, key, value를 받았고, attn_mask도 받았다.

```python
is_batched = query.dim() == 3

key_padding_mask = F._canonical_mask(
    mask=key_padding_mask,None
    mask_name="key_padding_mask",
    other_type=F._none_or_dtype(attn_mask),
    other_name="attn_mask",
    target_type=query.dtype
)

attn_mask = F._canonical_mask(
    mask=attn_mask,
    mask_name="attn_mask",
    other_type=None,
    other_name="",
    target_type=query.dtype,
    check_other=False,
)
```

is_batch : batch인지 단일 데이터인지. 배치니까 True

torch.nn.functional._canonical_mask는

```python
def _canonical_mask(
        mask: Optional[Tensor],
        mask_name: str,
        other_type: Optional[DType],
        other_name: str,
        target_type: DType,
        check_other: bool = True,
) -> Optional[Tensor]:

    if mask is not None:
        _mask_dtype = mask.dtype
        _mask_is_float = torch.is_floating_point(mask)
        if _mask_dtype != torch.bool and not _mask_is_float:
            raise AssertionError(
                f"only bool and floating types of {mask_name} are supported")
        if check_other and other_type is not None:
            if _mask_dtype != other_type:
                warnings.warn(
                    f"Support for mismatched {mask_name} and {other_name} "
                    "is deprecated. Use same type for both instead."
                )
        if not _mask_is_float:
            mask = (
                torch.zeros_like(mask, dtype=target_type)
                .masked_fill_(mask, float("-inf"))
            )
    return mask
```

attn_mask는 대각요소 포함한 upper 1 행렬이니까 _canonical_mask에 들어갔다 나오면 그냥 그대로 나온다..ㅎ