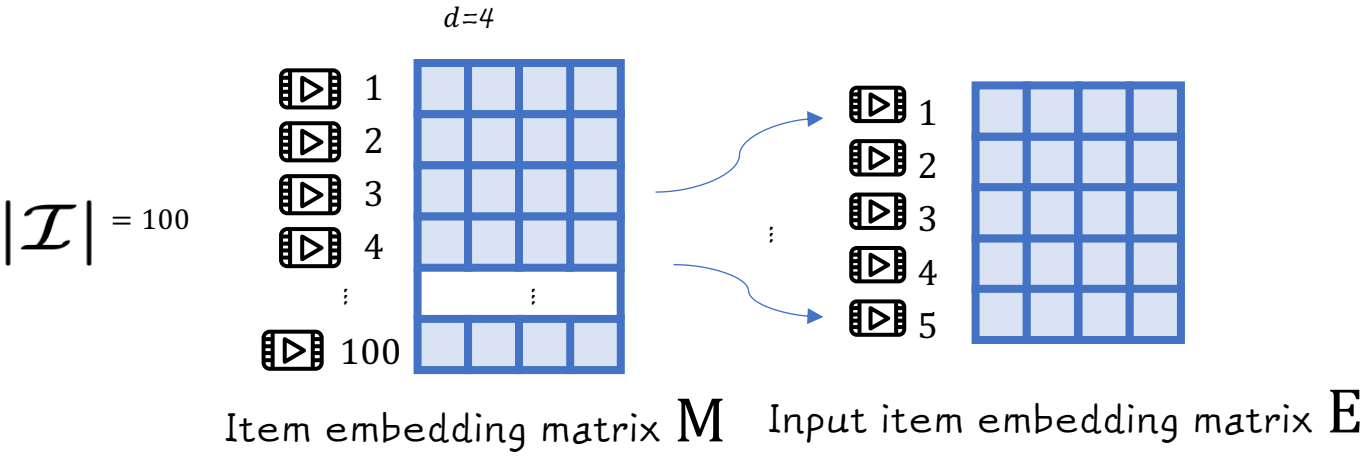
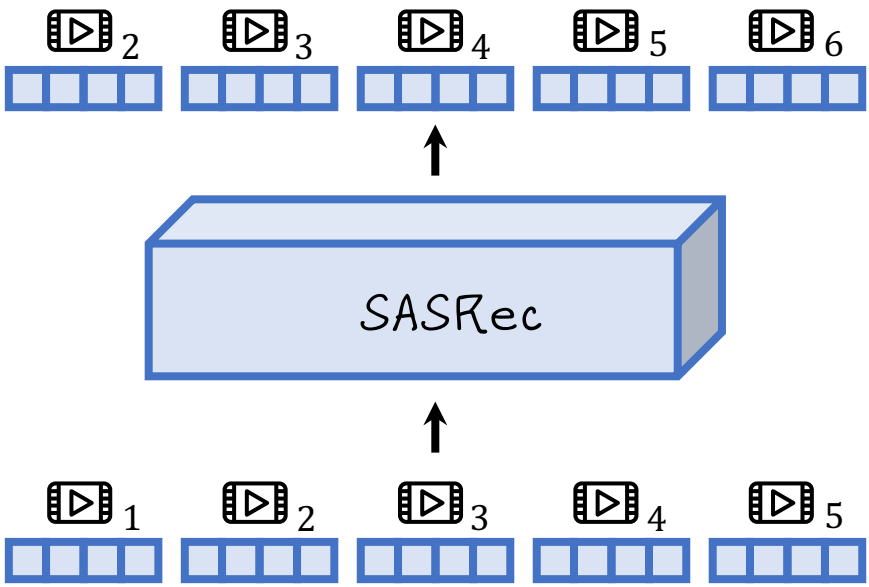
총 아이템 수는 100개, 아이템 임베딩 차원은 d=4차원, max length는 n=5 라고 해보자.

예시 유저 시퀀스 : 



마지막 아이템은 인풋에 안들어감!



III. METHODOLOGY

In the setting of sequential recommendation, we are given a user's action sequence $\mathcal{S}^u = (\mathcal{S}^u_1, \mathcal{S}^u_2, \ldots, \mathcal{S}^u_{|\mathcal{S}^u|})$, 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 $(\mathcal{S}^u_1, \mathcal{S}^u_2, \ldots, \mathcal{S}^u_{|\mathcal{S}^u|-1})$ and its expected output as a 'shifted' version of the same sequence: $(\mathcal{S}^u_2, \mathcal{S}^u_3, \ldots, \mathcal{S}^u_{|\mathcal{S}^u|})$. 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.

A. Embedding Layer

We transform the training sequence $(\mathcal{S}^u_1, \mathcal{S}^u_2, ..., \mathcal{S}^u_{|\mathcal{S}^u|-1})$ 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.

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:

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

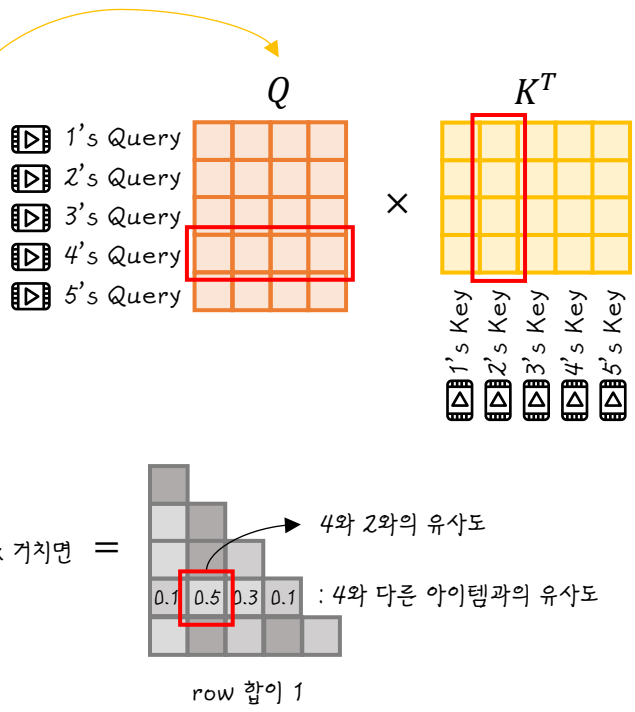
We also tried the fixed position embedding as used in [3], but found that this led to worse performance in our case. We analyze the effect of the position embedding quantitatively and qualitatively in our experiments.

B. Self-Attention Block

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

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

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.



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:

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

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).

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 $\mathbf{K}_j$ $(j > i)$.

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):

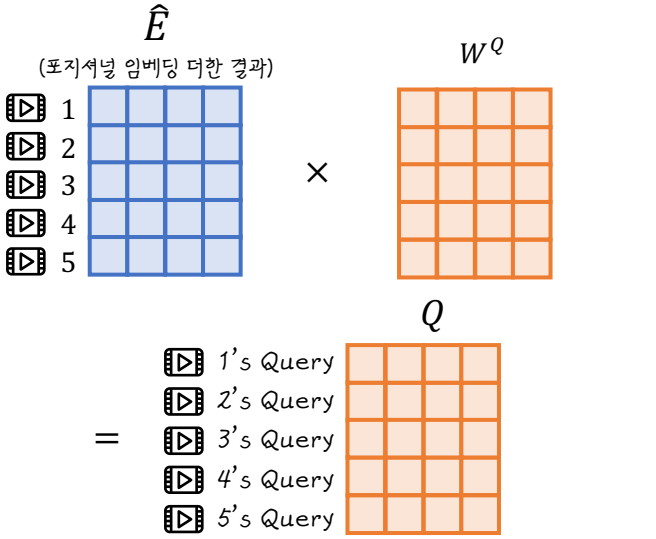
$$\mathbf{F}_i = \mathrm{FFN}(\mathbf{S}_i) = \mathrm{ReLU}(\mathbf{S}_i\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)}, \quad (4)$$

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).

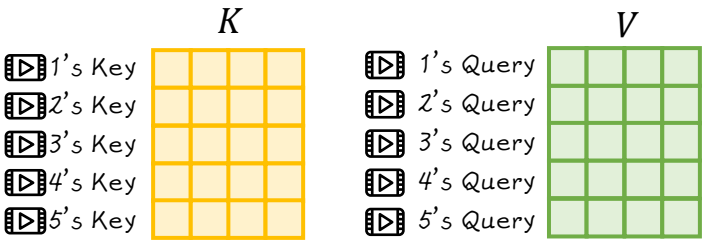
C. Stacking Self-Attention Blocks

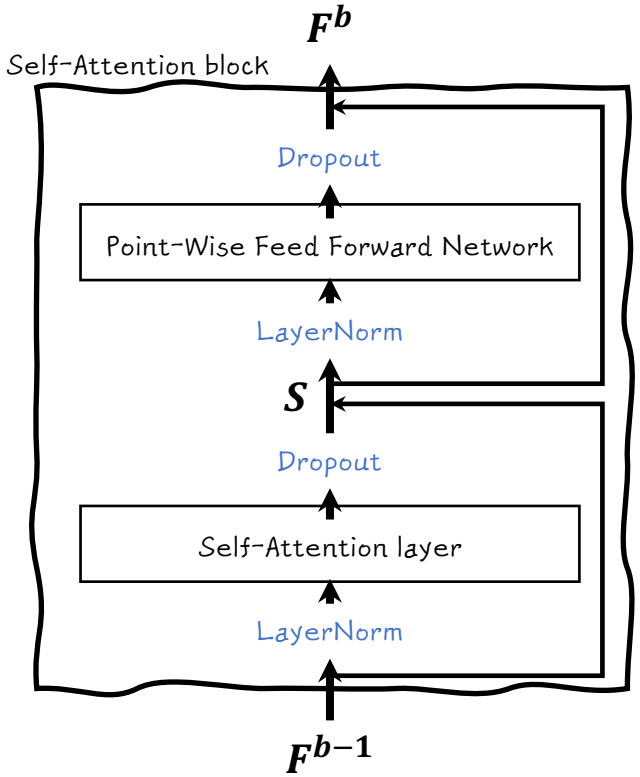
After the first self-attention block, $\mathbf{F}_i$ essentially aggregates all previous items' embeddings (i.e., $\widehat{\mathbf{E}}_j, j \leq i$). However, it might be useful to learn more complex item transitions via another self-attention block based on $\mathbf{F}$. Specifically, we stack the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

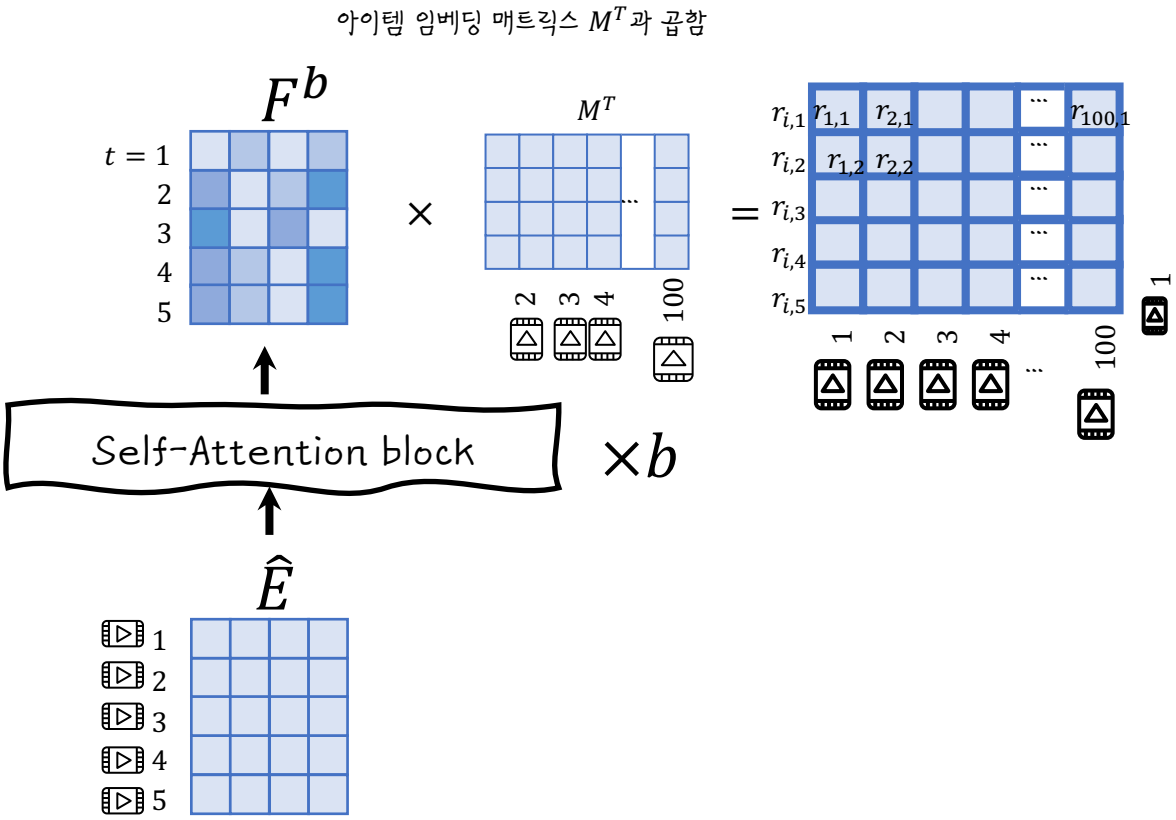
$$\begin{aligned} \mathbf{S}^{(b)} &= \mathrm{SA}(\mathbf{F}^{(b-1)}), \\ \mathbf{F}_i^{(b)} &= \mathrm{FFN}(\mathbf{S}_i^{(b)}), \quad \forall i \in \{1, 2, \ldots, n\}, \end{aligned} \quad (5)$$



같은 식으로 W^K, W^V와 연산하면







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.

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.

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:

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

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.

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:

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

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:

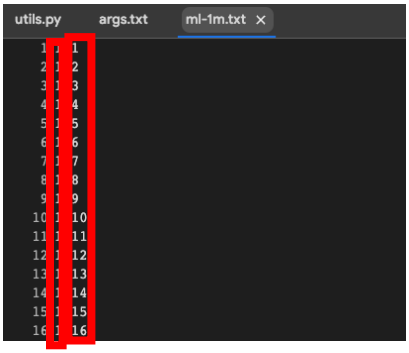
$$-\sum_{\mathcal{S}^u \in \mathcal{S}} \sum_{t \in [1,2,\ldots,n]} \left[\log(\sigma(r_{o_t,t})) + \sum_{j \notin \mathcal{S}^u} \log(1 - \sigma(r_{j,t})) \right].$$

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

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

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).

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

user_valid로

[]

user_train으로 user_test로

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:

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

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

논문에서의 목적 함수와 비교해보자.

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

Binary Cross Entropy를 쓴다고 했었다.

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.