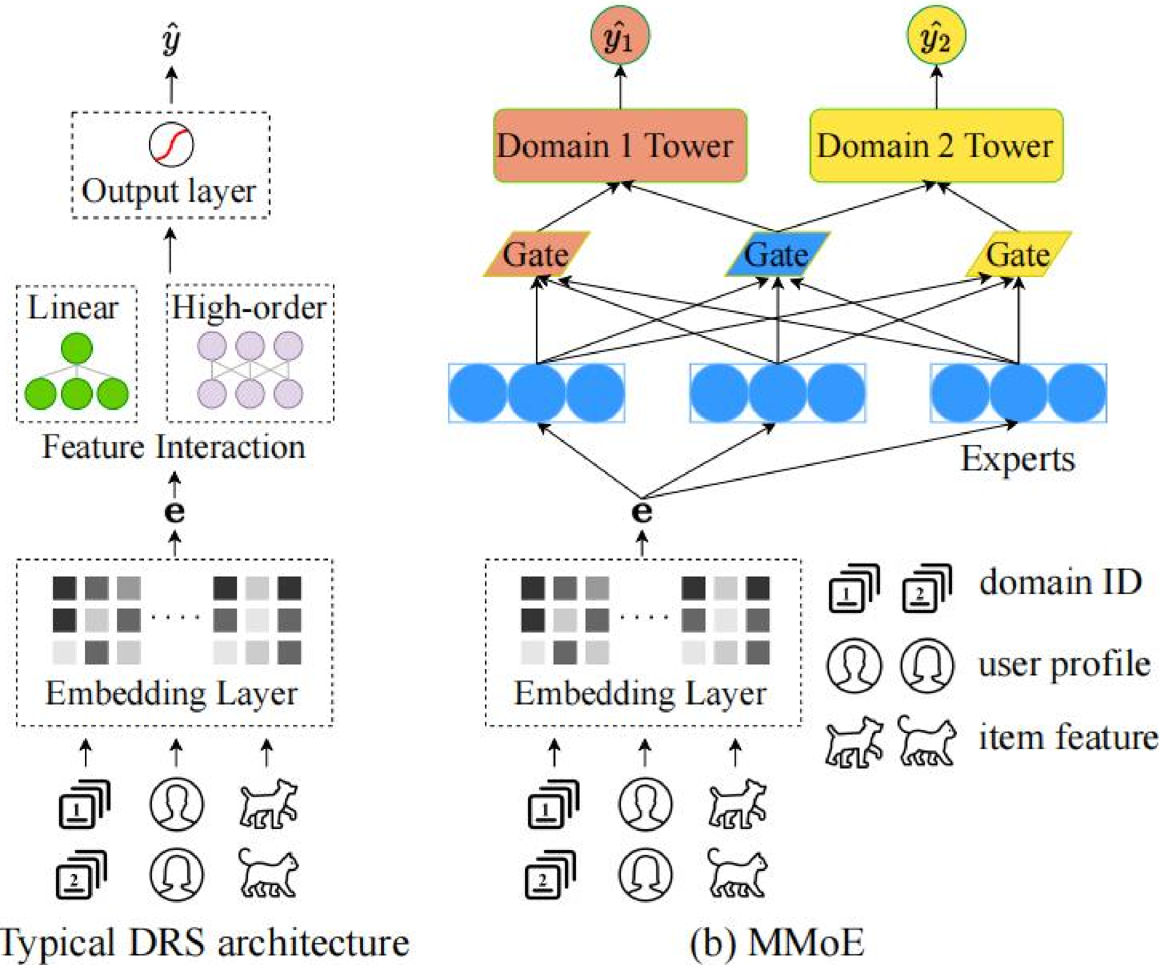
Method



(a) Typical DRS architecture (b) MMoE

Code:https://github.com/mindspore-lab/models/tree/master/research/huawei-noah/Diff-MSR
https://github.com/Applied-Machine-Learning-Lab/Diff-MSR

2024_WSDM_Diff-MSR： A Diffusion Model Enhanced Paradigm for Cold-Start Multi-Scenario Recommendation

Method

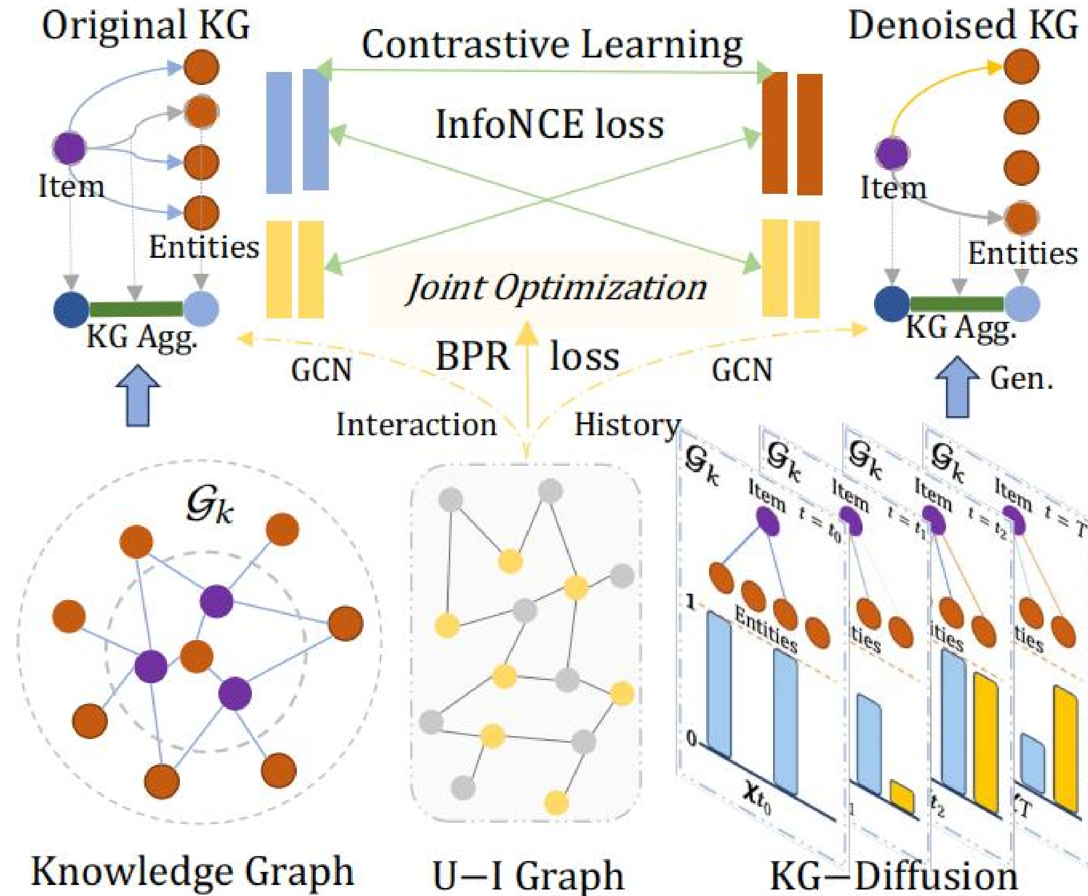


Figure 1: Overall framework of the proposed DiffKG model.

Code:https://github.com/HKUDS/DiffKG

2024_WSDM_DiffKG： Knowledge Graph Diffusion Model for Recommendation

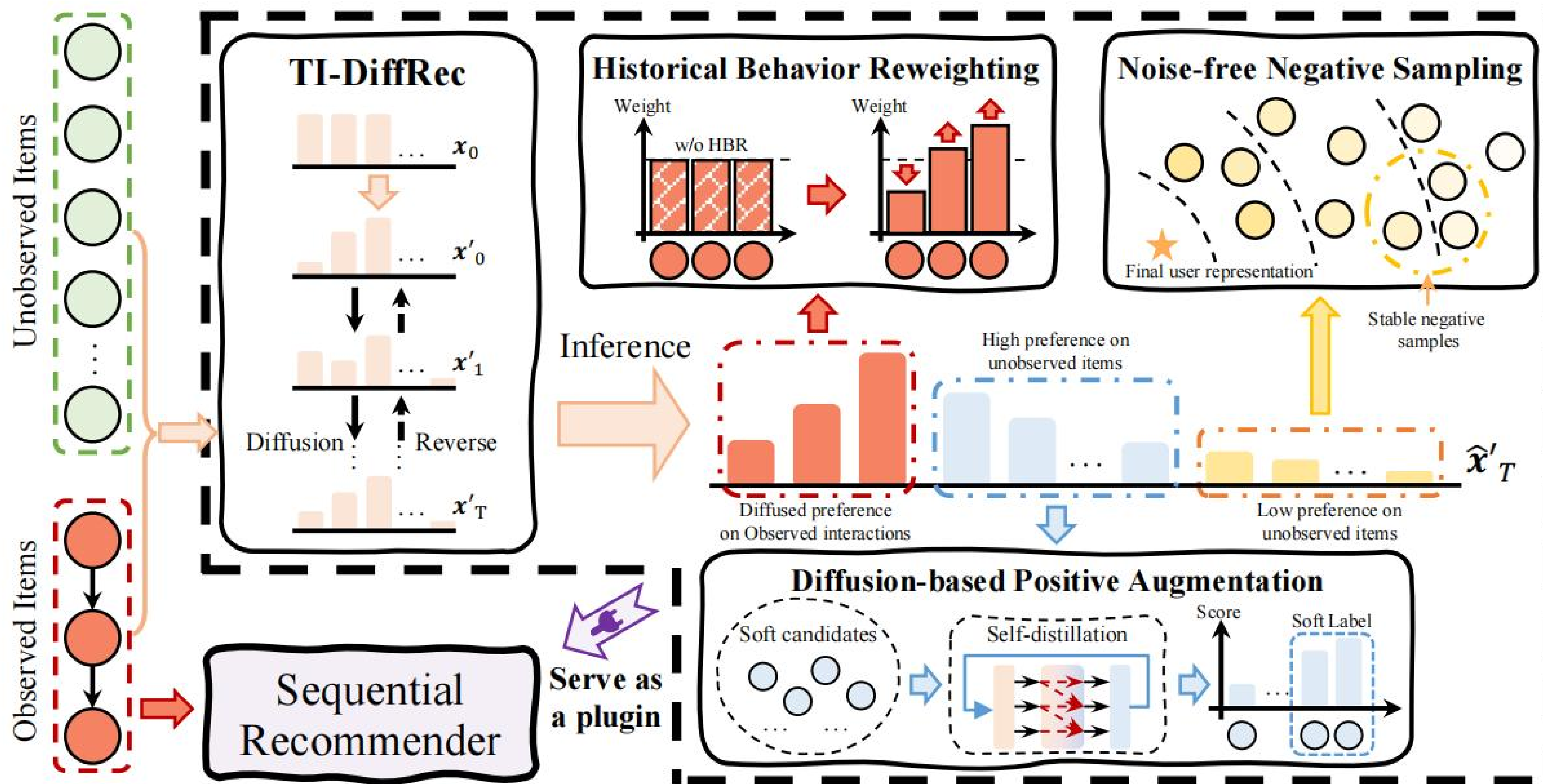
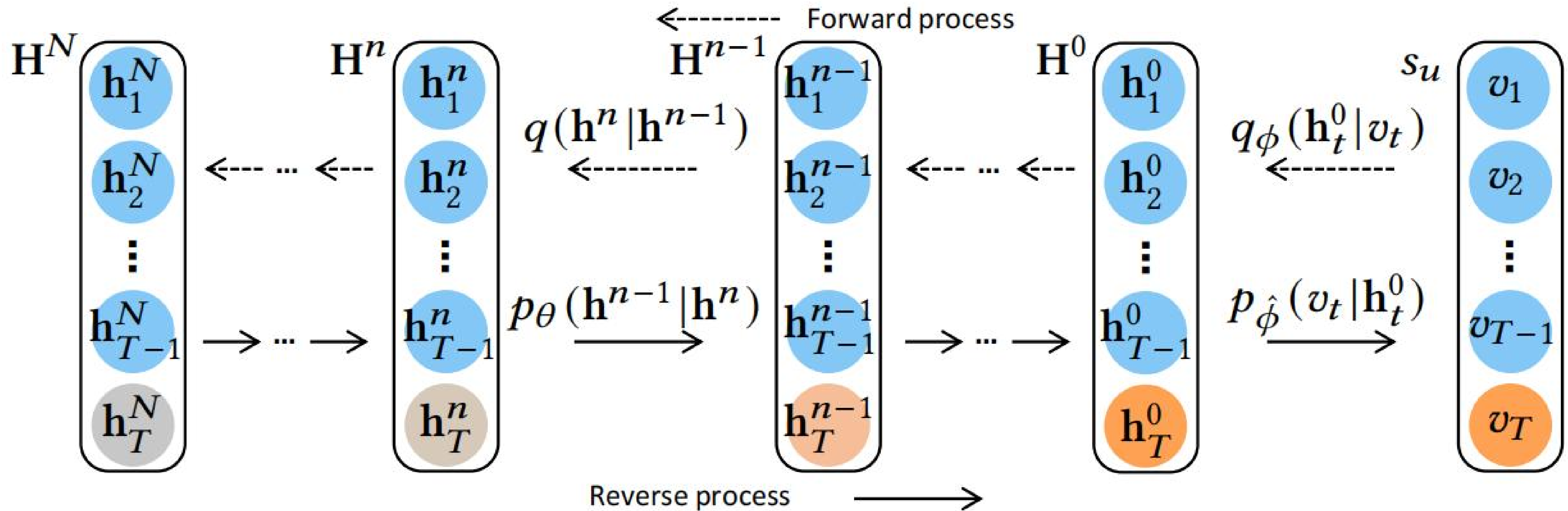


Figure 3: The overall structure of the proposed PDRec.

Code:https://github.com/hulkima/PDRec

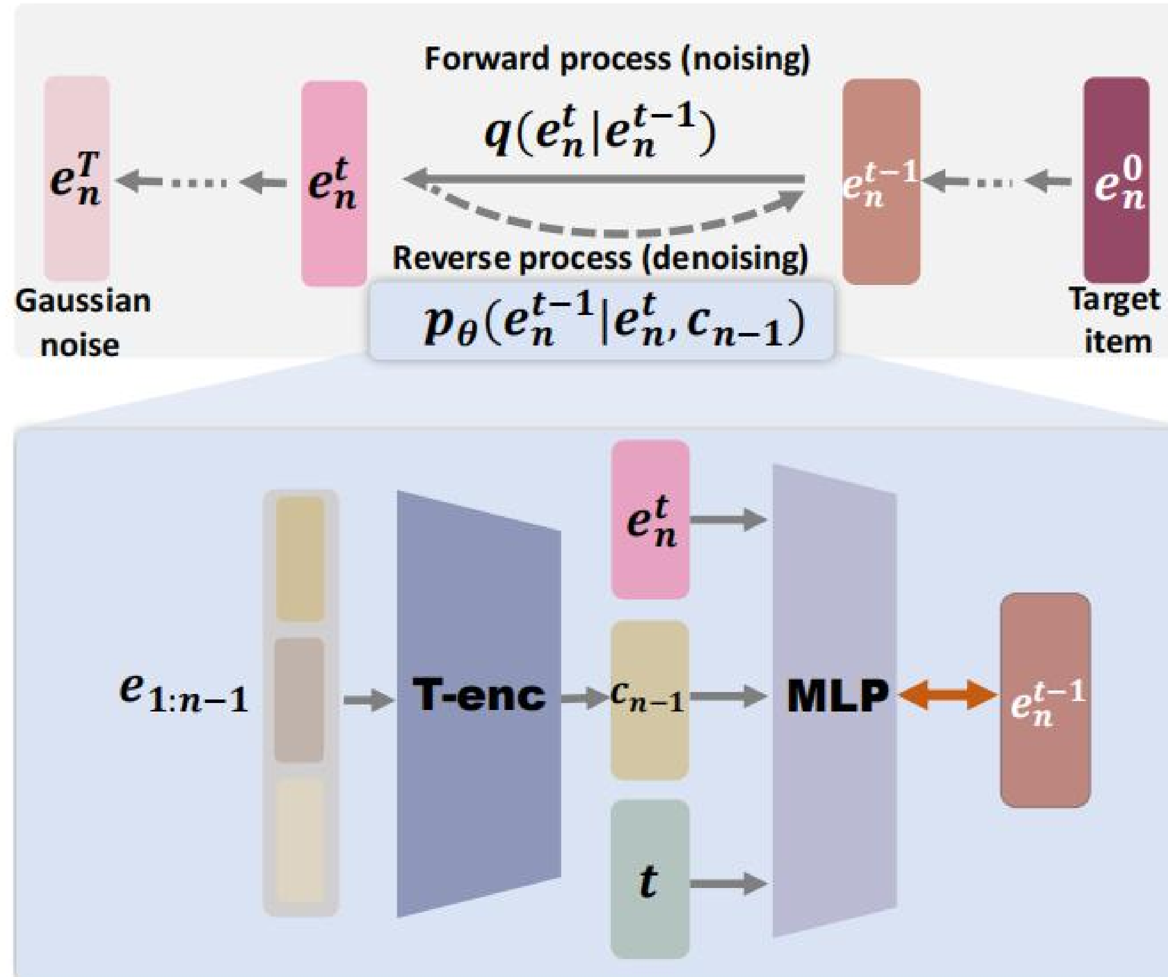
2024_AAAI_Plug-in Diffusion Model for Sequential Recommendation

Method



2023_o_Sequential Recommendation with Diffusion Models

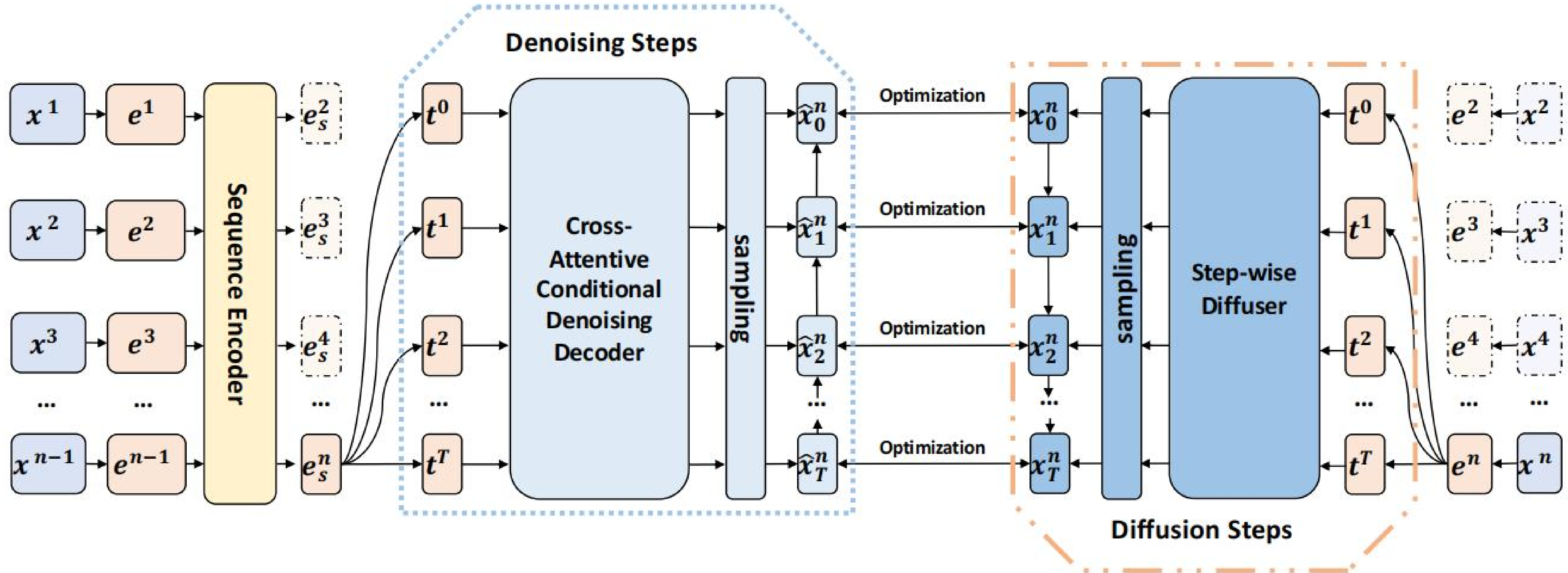
Method



Code:https://github.com/YangZhengyi98/DreamRec

2023_NeurIPS_generate-what-you-prefer-reshaping-sequential-recommendation-via-guided-diffusion-Paper-Conference

Method



2023_Conditional Denoising Diffusion for Sequential Recommendation

Method

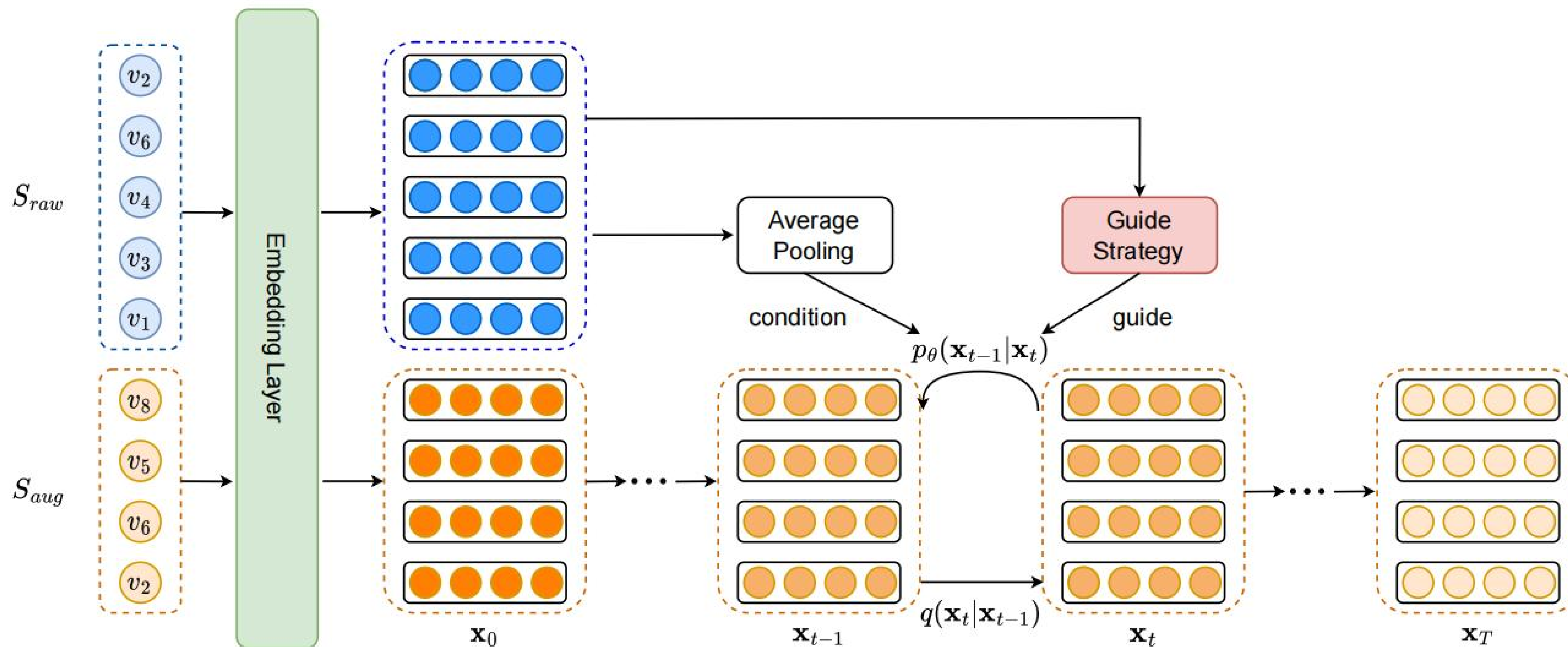


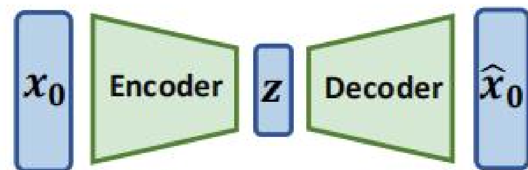
Figure 3: The overview of the proposed Diffusion Augmentation for Sequential Recommendation (DiffuASR).

Code:https://github.com/liuqidong07/DiffuASR

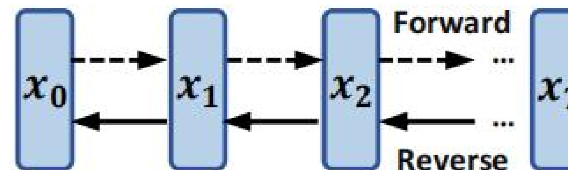
https://gitee.com/mindspore/models/tree/master/research/recommend/DiffuASR

2023_CIKM_Diffusion Augmentation for Sequential Recommendation

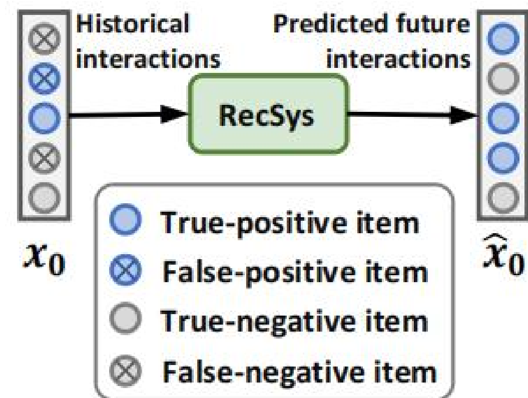
Method



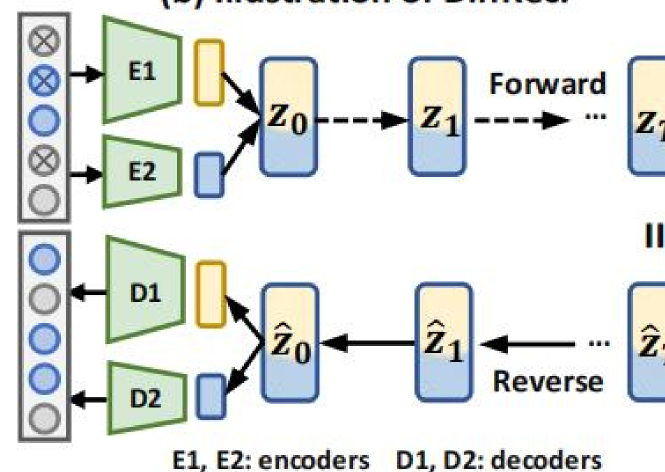
(a) Illustration of VAE.



(b) Illustration of DiffRec.



(c) Objective of recommender systems.



(d) Illustration of L-DiffRec.

Figure 1: Illustration of VAE, DiffRec, the objective of recommender systems, and L-DiffRec.

2023_ACM_Diffusion Recommender Model

Method

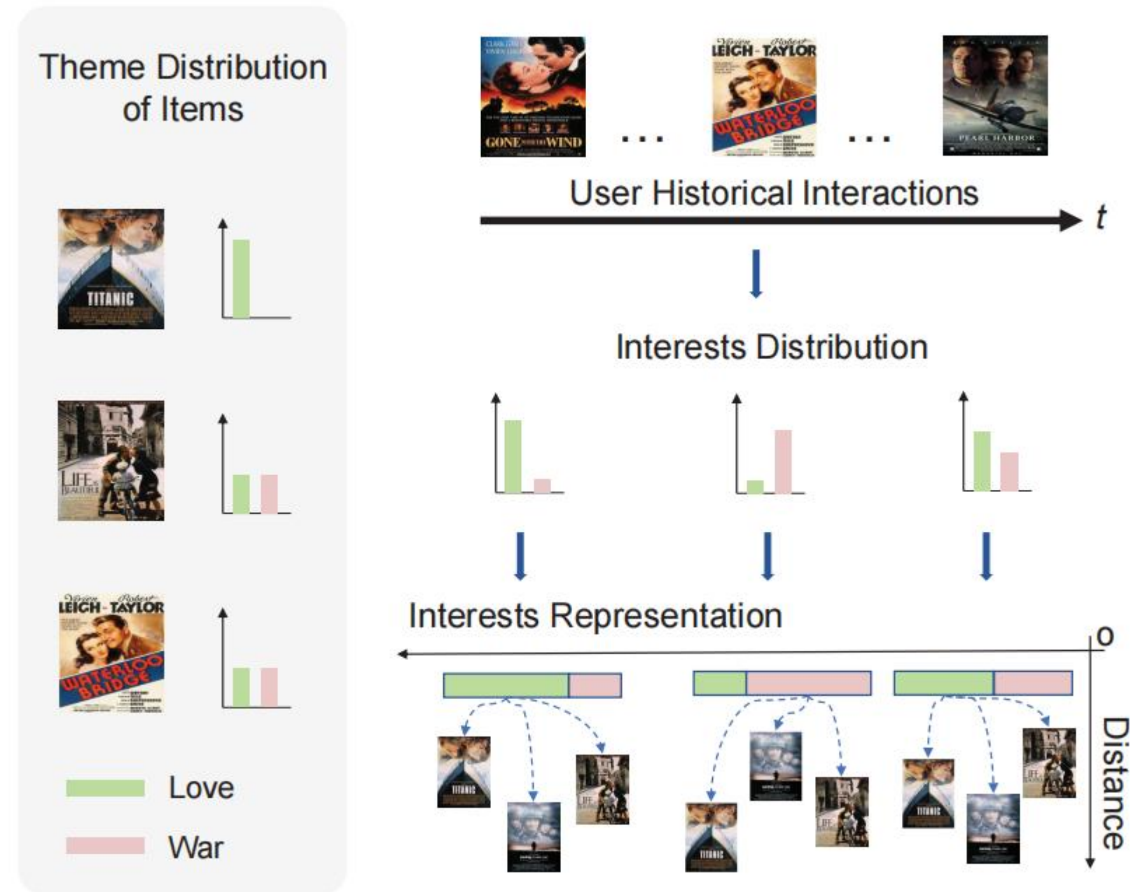
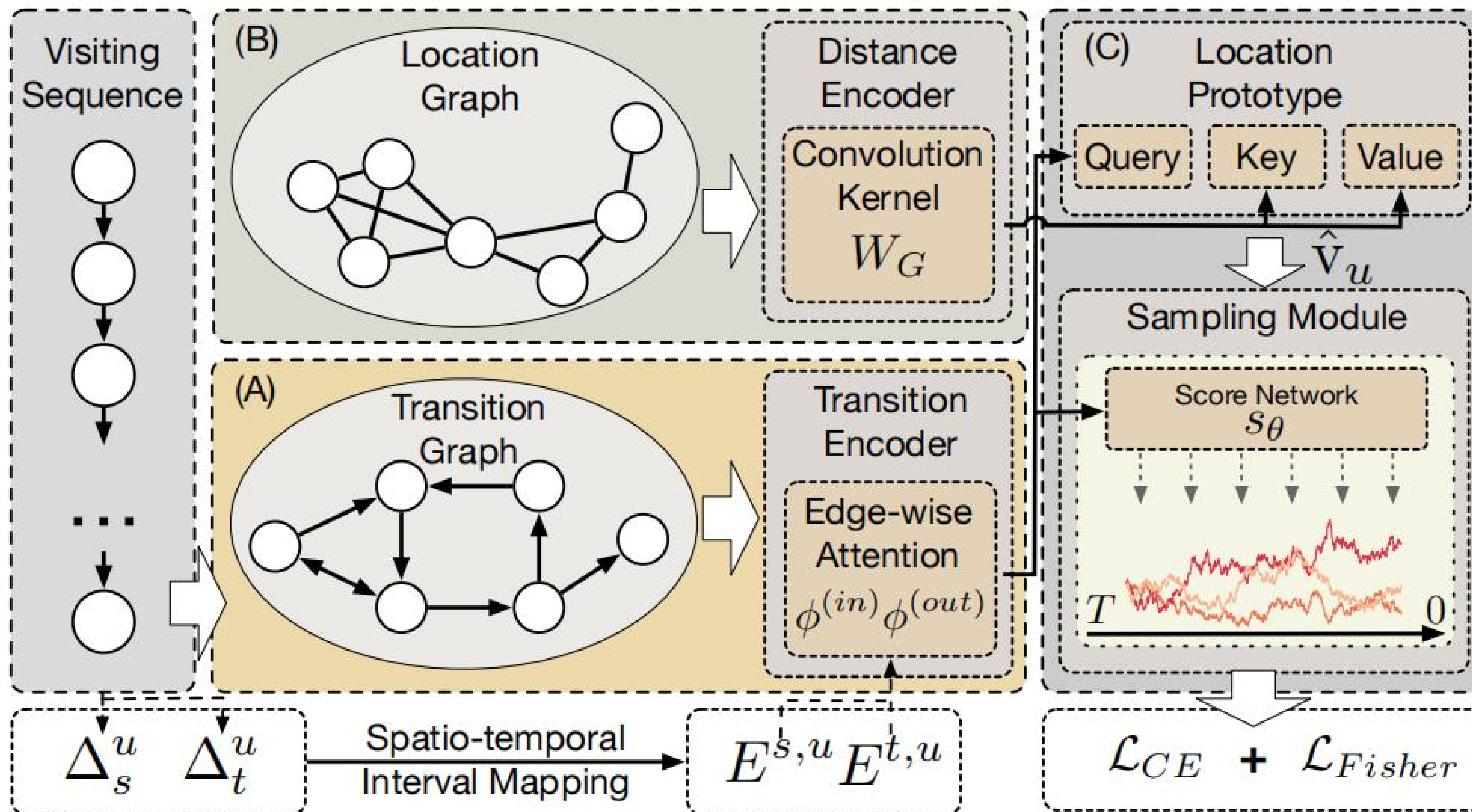


Fig. 1. An example of multiple interests of users and multiple aspects of items.

Code:https://github.com/WHUIR/DiffuRec

2023_ACM_DiffuRec： A Diffusion Model for Sequential Recommendation

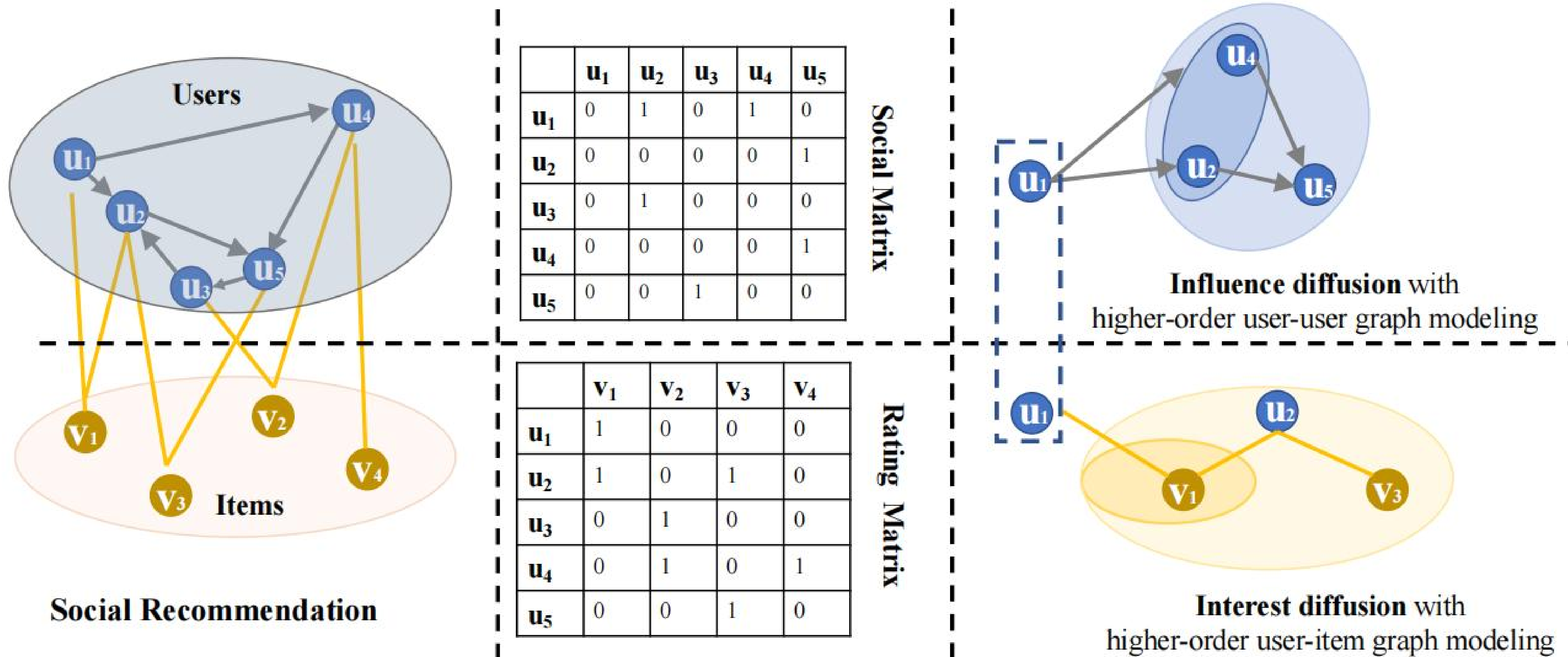
Method



Code:https://github.com/Yifang-Qin/Diff-POI.

2023_ACM_A Diffusion model for POI recommendation

Method



	u_1	u_2	u_3	u_4	u_5
u_1	0	1	0	1	0
u_2	0	0	0	0	1
u_3	0	1	0	0	0
u_4	0	0	0	0	1
u_5	0	0	1	0	0

	v_1	v_2	v_3	v_4
u_1	1	0	0	0
u_2	1	0	1	0
u_3	0	1	0	0
u_4	0	1	0	1
u_5	0	0	1	0

Fig. 1. An overall illustration of social recommendation. The second column shows how traditional models treat this problem with matrix representations of users' two kinds of behaviors. In this paper, we try to model both the influence diffusion and interest diffusion with graph representation of users' two kinds of behaviors.

Code:https://github.com/PeiJieSun/diffnet

2021_IEEE_DiffNet++： A Neural Influence and Interest Diffusion Network for Social Recommendation

Method

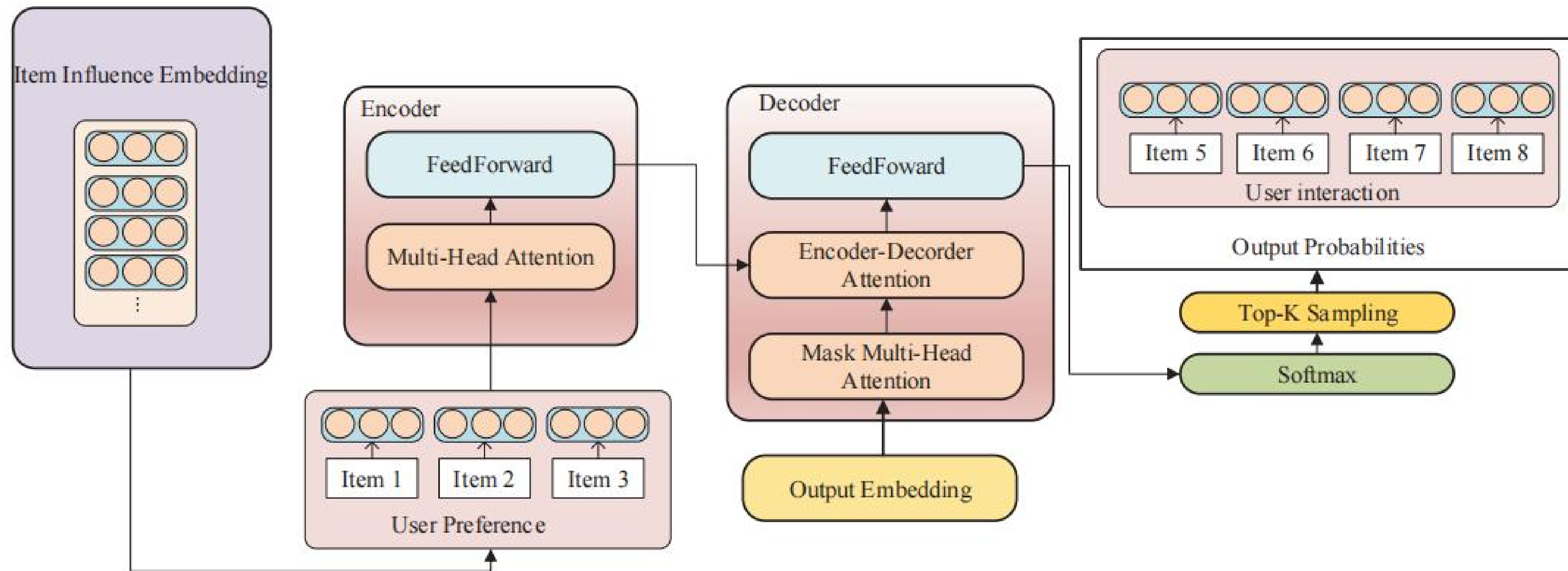


Fig. 1: The Framwork of User Preference Translation model with Item Influence diffusion Embedding

2020_ASONAM_User Preference Translation Model for Recommendation System with Item Influence Diffusion Embedding

Method

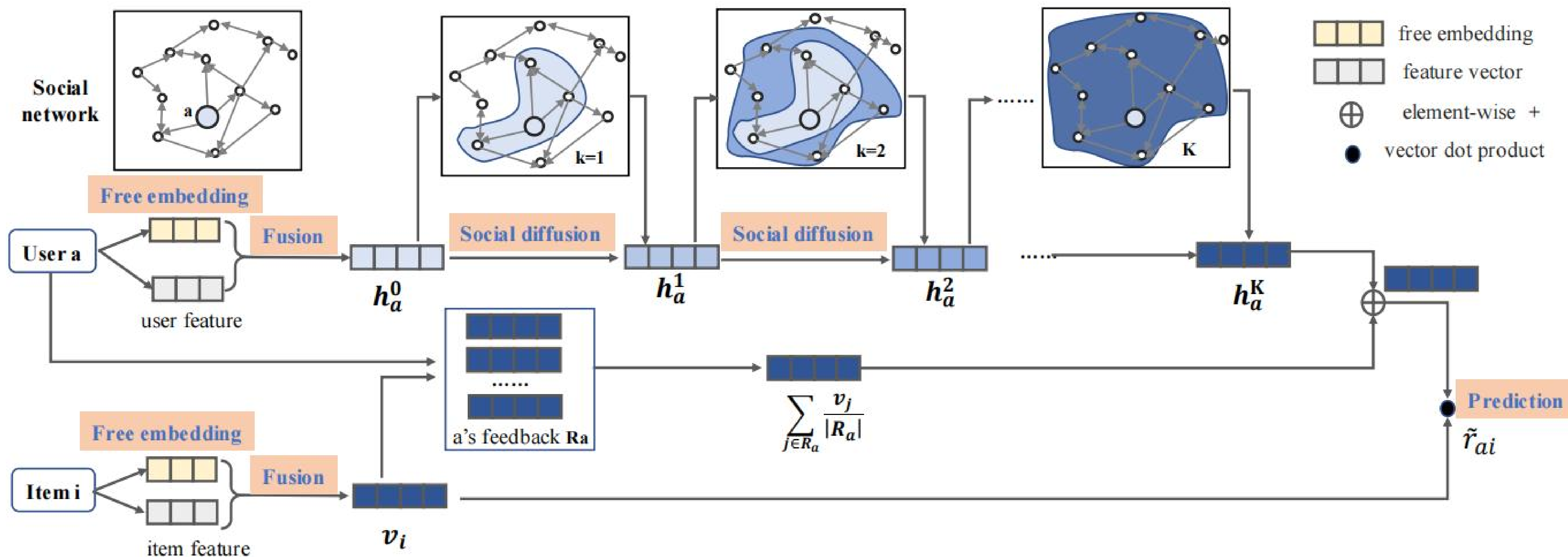


Figure 1: The overall architecture of our proposed model. The four parts of DiffNet are shown with orange background.

2019_SIGIR_A Neural Influence Diffusion Model for Social Recommendation

code: https://github.com/PeiJieSun/diffnet

Code:A Diffusion Model for Sequential Recommendation

```python
import numpy as np
from abc import ABC, abstractmethod
import torch as th
import torch.distributed as dist
```

```python
class UniformSampler(ScheduleSampler):
    def __init__(self, num_timesteps):
        self.num_timesteps = num_timesteps
        self._weights = np.ones([self.num_timesteps])

    def weights(self):
        return self._weights
```

step_sample.py

```python
class ScheduleSampler(ABC):
    """
    A distribution over timesteps in the diffusion process, intended to reduce
    variance of the objective.

    By default, samplers perform unbiased importance sampling, in which the
    objective's mean is unchanged.
    However, subclasses may override sample() to change how the resampled
    terms are reweighted, allowing for actual changes in the objective.
    """

    @abstractmethod
    def weights(self):
        """
        Get a numpy array of weights, one per diffusion step.
        The weights needn't be normalized, but must be positive.
        """

    def sample(self, batch_size, device):
        """
        Importance-sample timesteps for a batch.

        :param batch_size: the number of timesteps.
        :param device: the torch device to save to.
        :return: a tuple (timesteps, weights):
                 - timesteps: a tensor of timestep indices.
                 - weights: a tensor of weights to scale the resulting losses.
        """
        w = self.weights()
        p = w / np.sum(w)
        indices_np = np.random.choice(len(p), size=(batch_size,), p=p)
        indices = th.from_numpy(indices_np).long().to(device)
        weights_np = 1 / (len(p) * p[indices_np])
        weights = th.from_numpy(weights_np).float().to(device)
        return indices, weights
```

```python
class LossAwareSampler(ScheduleSampler):
    def update_with_local_losses(self, local_ts, local_losses):
        """
        Update the reweighting using losses from a model.

        Call this method from each rank with a batch of timesteps and the
        corresponding losses for each of those timesteps.
        This method will perform synchronization to make sure all of the ranks
        maintain the exact same reweighting.

        :param local_ts: an integer Tensor of timesteps.
        :param local_losses: a 1D Tensor of losses.
        """
        batch_sizes = [
            th.tensor([0], dtype=th.int32, device=local_ts.device)
            for _ in range(dist.get_world_size())
        ]
        dist.all_gather(
            batch_sizes,
            th.tensor([len(local_ts)], dtype=th.int32, device=local_ts.device),
        )

        # Pad all_gather batches to be the maximum batch size.
        batch_sizes = [x.item() for x in batch_sizes]
        max_bs = max(batch_sizes)

        timestep_batches = [th.zeros(max_bs).to(local_ts) for bs in batch_sizes]
        loss_batches = [th.zeros(max_bs).to(local_losses) for bs in batch_sizes]
        dist.all_gather(timestep_batches, local_ts)
        dist.all_gather(loss_batches, local_losses)
        timesteps = [
            x.item() for y, bs in zip(timestep_batches, batch_sizes) for x in y[:bs]
        ]
        losses = [x.item() for y, bs in zip(loss_batches, batch_sizes) for x in y[:bs]]
        self.update_with_all_losses(timesteps, losses)
```

```python
    @abstractmethod
    def update_with_all_losses(self, ts, losses):
        """
        Update the reweighting using losses from a model.

        Sub-classes should override this method to update the reweighting
        using losses from the model.

        This method directly updates the reweighting without synchronizing
        between workers. It is called by update_with_local_losses from all
        ranks with identical arguments. Thus, it should have deterministic
        behavior to maintain state across workers.

        :param ts: a list of int timesteps.
        :param losses: a list of float losses, one per timestep.
        """
```

step_sample.py

Code:A Diffusion Model for Sequential Recommendation


```
107 class LossSecondMomentResampler(LossAwareSampler):
108     def __init__(self, num_timesteps, history_per_term=10, uniform_prob=0.001):
109         self.num_timesteps = num_timesteps
110         self.history_per_term = history_per_term
111         self.uniform_prob = uniform_prob
112         self._loss_history = np.zeros(
113             [self.num_timesteps, history_per_term], dtype=np.float64
114         )
115         self._loss_counts = np.zeros([self.num_timesteps], dtype=np.int)
116 
117     def weights(self):
118         if not self._warmed_up():
119             return np.ones([self.num_timesteps], dtype=np.float64)
120         weights = np.sqrt(np.mean(self._loss_history ** 2, axis=-1))
121         weights /= np.sum(weights)
122         weights *= 1 - self.uniform_prob
123         weights += self.uniform_prob / len(weights)
124         return weights
125 
126     def update_with_all_losses(self, ts, losses):
127         for t, loss in zip(ts, losses):
128             if self._loss_counts[t] == self.history_per_term:
129                 # Shift out the oldest loss term.
130                 self._loss_history[t, :-1] = self._loss_history[t, 1:]
131                 self._loss_history[t, -1] = loss
132             else:
133                 self._loss_history[t, self._loss_counts[t]] = loss
134                 self._loss_counts[t] += 1
135 
136     def _warmed_up(self):
137         return (self._loss_counts == self.history_per_term).all()
```

step_sample.py

Code:A Diffusion Model for Sequential Recommendation


```
140 class FixSampler(ScheduleSampler):
141     def __init__(self, num_timesteps):
142         self.num_timesteps = num_timesteps
143         ###############################################################
144         ### You can custome your own sampling weight of steps here. ###
145         ###############################################################
146         self._weights = np.concatenate([np.ones([num_timesteps//2]), np.zeros([num_timesteps//2]) + 0.5])
147 
148     def weights(self):
149         return self._weights
150 
151 
152 def create_named_schedule_sampler(name, num_timesteps):
153     """
154     Create a ScheduleSampler from a library of pre-defined samplers.
155     :param name: the name of the sampler.
156     :param diffusion: the diffusion object to sample for.
157     """
158     if name == "uniform":
159         return UniformSampler(num_timesteps)
160     elif name == "lossaware":
161         return LossSecondMomentResampler(num_timesteps)  ## default setting
162     elif name == "fixstep":
163         return FixSampler(num_timesteps)
164     else:
165         raise NotImplementedError(f"unknown schedule sampler: {name}")
166
```

step_sample.py

Code:A Diffusion Model for Sequential Recommendation


```
1  import torch.nn as nn
2  import torch as th
3  from recbole.model.sequential_recommender.step_sample import create_named_schedule_sampler
4  import numpy as np
5  import math
6  import torch
7  import torch.nn.functional as F
10 def _extract_into_tensor(arr, timesteps, broadcast_shape):
11     """
12     Extract values from a 1-D numpy array for a batch of indices.
13
14     :param arr: the 1-D numpy array.
15     :param timesteps: a tensor of indices into the array to extract.
16     :param broadcast_shape: a larger shape of K dimensions with the batch
17                             dimension equal to the length of timesteps.
18     :return: a tensor of shape [batch_size, 1, ...] where the shape has K dims.
19     """
20
21     res = th.from_numpy(arr).to(device=timesteps.device)[timesteps].float()
22     while len(res.shape) < len(broadcast_shape):
23         res = res[..., None]
24     return res.expand(broadcast_shape)
```

diffurec.py

Code:A Diffusion Model for Sequential Recommendation

```
27  def get_named_beta_schedule(schedule_name, num_diffusion_timesteps):
28
29      Get a pre-defined beta schedule for the given name.
30      The beta schedule library consists of beta schedules which remain similar in the limit of num_diffusion_timesteps. Beta schedules may be added,
31      """
32      if schedule_name == "linear":
33          # Linear schedule from Ho et al, extended to work for any number of
34          # diffusion steps.
35          scale = 1000 / num_diffusion_timesteps
36          beta_start = scale * 0.0001
37          beta_end = scale * 0.02
38          return np.linspace(beta_start, beta_end, num_diffusion_timesteps, dtype=np.float64)
39      elif schedule_name == "cosine":
40          return betas_for_alpha_bar(num_diffusion_timesteps, lambda t: math.cos((t + 0.008) / 1.008 * math.pi / 2) ** 2,)
41      elif schedule_name == 'sqrt':
42          return betas_for_alpha_bar(num_diffusion_timesteps,lambda t: 1-np.sqrt(t + 0.0001),  )
43      elif schedule_name == "trunc_cos":
44          return betas_for_alpha_bar_left(num_diffusion_timesteps, lambda t: np.cos((t + 0.1) / 1.1 * np.pi / 2) ** 2,)
45      elif schedule_name == 'trunc_lin':
46          scale = 1000 / num_diffusion_timesteps
47          beta_start = scale * 0.0001 + 0.01
48          beta_end = scale * 0.02 + 0.01
49          if beta_end > 1:
50              beta_end = scale * 0.001 + 0.01
51          return np.linspace(beta_start, beta_end, num_diffusion_timesteps, dtype=np.float64)
52      elif schedule_name == 'pw_lin':
53          scale = 1000 / num_diffusion_timesteps
54          beta_start = scale * 0.0001 + 0.01
55          beta_mid = scale * 0.0001  #scale * 0.02
56          beta_end = scale * 0.02
57          first_part = np.linspace(beta_start, beta_mid, 10, dtype=np.float64)
58          second_part = np.linspace(beta_mid, beta_end, num_diffusion_timesteps - 10 , dtype=np.float64)
59          return np.concatenate([first_part, second_part])
60      else:
61          raise NotImplementedError(f"unknown beta schedule: {schedule_name}")
```

diffurec.py

Code:A Diffusion Model for Sequential Recommendation

```
def betas_for_alpha_bar(num_diffusion_timesteps, alpha_bar, max_beta=0.999):
    """
    Create a beta schedule that discretizes the given alpha_t_bar function, which defines the cumulative product of (1-beta) over time from t = [0,
    :param num_diffusion_timesteps: the number of betas to produce.
    :param alpha_bar: a lambda that takes an argument t from 0 to 1 and produces the cumulative product of (1-beta) up to that part of the diffusio
    :param max_beta: the maximum beta to use; use values lower than 1 to prevent singularities.
    """
    betas = []
    for i in range(num_diffusion_timesteps):  ## 2000
        t1 = i / num_diffusion_timesteps
        t2 = (i + 1) / num_diffusion_timesteps
        betas.append(min(1 - alpha_bar(t2) / alpha_bar(t1), max_beta))
    return np.array(betas)


def betas_for_alpha_bar_left(num_diffusion_timesteps, alpha_bar, max_beta=0.999):
    """
    Create a beta schedule that discretizes the given alpha_t_bar function, but shifts towards left interval starting from 0
    which defines the cumulative product of (1-beta) over time from t = [0,1].

    :param num_diffusion_timesteps: the number of betas to produce.
    :param alpha_bar: a lambda that takes an argument t from 0 to 1 and
                      produces the cumulative product of (1-beta) up to that
                      part of the diffusion process.
    :param max_beta: the maximum beta to use; use values lower than 1 to
                     prevent singularities.
    """
    betas = []
    betas.append(min(1-alpha_bar(0), max_beta))
    for i in range(num_diffusion_timesteps-1):
        t1 = i / num_diffusion_timesteps
        t2 = (i + 1) / num_diffusion_timesteps
        betas.append(min(1 - alpha_bar(t2) / alpha_bar(t1), max_beta))
    return np.array(betas)
```

diffurec.py

```
100    def space_timesteps(num_timesteps, section_counts):
118                             is a number of steps to use the striding from the
119                             DDIM paper.
120        :return: a set of diffusion steps from the original process to use.
121        """
122        if isinstance(section_counts, str):
123            if section_counts.startswith("ddim"):
124                desired_count = int(section_counts[len("ddim") :])
125                for i in range(1, num_timesteps):
126                    if len(range(0, num_timesteps, i)) == desired_count:
127                        return set(range(0, num_timesteps, i))
128                raise ValueError(
129                    f"cannot create exactly {num_timesteps} steps with an integer stride"
130                )
131            section_counts = [int(x) for x in section_counts.split(",")]
132        size_per = num_timesteps // len(section_counts)
133        extra = num_timesteps % len(section_counts)
134        start_idx = 0
135        all_steps = []
136        for i, section_count in enumerate(section_counts):
137            size = size_per + (1 if i < extra else 0)
138            if size < section_count:
139                raise ValueError(
140                    f"cannot divide section of {size} steps into {section_count}"
141                )
142            if section_count <= 1:
143                frac_stride = 1
144            else:
145                frac_stride = (size - 1) / (section_count - 1)
146            cur_idx = 0.0
147            taken_steps = []
148            for _ in range(section_count):
149                taken_steps.append(start_idx + round(cur_idx))
150                cur_idx += frac_stride
151            all_steps += taken_steps
152            start_idx += size
153        return set(all_steps)
```

diffurec.py

详细代码：
https://github.com/WHUIR/DiffuRec

文件：diffurec.py

示例代码

```
109             '''扩散模型'''
110             self.diff = DiffuRec(hidden_size=self.hidden_size,
111                                  schedule_sampler_name='lossaware',#diffusion for t generation
112                                  diffusion_steps=32,               #diffusion step
113                                  noise_schedule='trunc_lin',       #beta generation ##cosine, linear, trunc_cos, trunc_lin, pw_lin, sqrt
114                                  rescale_timesteps=True,           #rescal timesteps
115                                  lambda_uncertainty=0.001,         #uncertainty weight
116                                  dropout=0.5,
117                                  num_blocks=4)                     #Number of Transformer blocks
```

```
def forward(self, item_seq, item_seq_len,target=None, train_flag=False):
    item_emb = self.item_embedding(item_seq)

    '''扩散模型'''
    item_embeddings=self.dropout(item_emb)
    mask_seq = (item_seq>0).float()
    if train_flag:
        tag_emb = self.item_embedding(target.squeeze(-1))  ## B x H #得到目标item的表示
        rep_diffu, rep_item, weights, t = self.diff(item_embeddings, tag_emb, mask_seq)

        # item_rep_dis = self.regularization_rep(rep_item, mask_seq)
        # seq_rep_dis = self.regularization_seq_item_rep(rep_diffu, rep_item, mask_seq)

        item_rep_dis = None
        seq_rep_dis = None
    else:
        # noise_x_t = th.randn_like(tag_emb)
        noise_x_t = torch.randn_like(item_embeddings[:,-1,:],device=item_seq.device)
        rep_diffu = self.diff.reverse_p_sample(item_embeddings, noise_x_t, mask_seq)
        weights, t, item_rep_dis, seq_rep_dis = None, None, None, None
```

输入：[batch_size,seq_len,hiddensize]
输出：[batch_size,hiddensize]

输入为初始的item的Embedding([batch_size,seq_len,hiddensize])以及标签embedding([batch_size,hiddensize])，输入标签是想加入与标签类似的正态分布噪音，在训练时候加入，在测试时使用随机生成的噪音。最终模型的输出可直接用来预测推荐序列。

Thanks