Neural Input Search for Large Scale Recommendation Models

Manas R. Joglekar*
joglekarmanas@gmail.com

Cong Li
congcli@google.com
Google

Mei Chen
meic@google.com
Google

Taibai Xu
taibaixu@google.com
Google

Xiaoming Wang
wxm@google.com
Google

Jay K. Adams*
jadams@pinterest.com
Pinterest

Pranav Khaitan
pranavkhaitan@google.com
Google

Jiahui Liu
jiahui@google.com
Google

Quoc V. Le
qvl@google.com
Google

ABSTRACT

Recommendation problems with large numbers of discrete items, such as products, webpages, or videos, are ubiquitous in the technology industry. Deep neural networks are being increasingly used for these recommendation problems. These models use *embeddings* to represent discrete items as continuous vectors, and the vocabulary sizes and embedding dimensions, despite their heavy influence on the model's accuracy, are often manually selected in a heuristical manner.

We present Neural Input Search (NIS), a technique for learning the optimal vocabulary sizes and embedding dimensions for categorical features. The goal is to maximize prediction accuracy subject to a constraint on the total memory used by all embeddings. Moreover, we argue that the traditional Single-size Embedding (SE), which uses the same embedding dimension for all values of a feature, suffers from inefficient usage of model capacity and training data. We propose a novel type of embedding, namely Multi-size Embedding (ME), which allows the embedding dimension to vary for different values of the feature. During training we use reinforcement learning to find the optimal vocabulary size for each feature and embedding dimension for each value of the feature. Experimentation on two public recommendation datasets shows that NIS can find significantly better models with much fewer embedding parameters. We also deployed NIS in production to a real world large scale App ranking model in our company's App store, Google Play, resulting in +1.02% App Install with 30% smaller model size.

*The work was done when the author was a Google employee.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
KDD '20, August 23–27, 2020, Virtual Event, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7998-4/20/08.
https://doi.org/10.1145/3394486.3403288

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Information systems** → *Recommender systems*.

KEYWORDS

AutoML; embeddings; recommendation models

ACM Reference Format:
Manas R. Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K. Adams, Pranav Khaitan, Jiahui Liu, and Quoc V. Le. 2020. Neural Input Search for Large Scale Recommendation Models. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403288

1 INTRODUCTION

Most modern neural network models can be thought of as comprising two components: an input component that converts raw (possibly categorical) input data into floating point values; and a representation learning component that combines the outputs of the input component and computes the final output of the model. Designing neural network architectures in an automated, data driven manner (*AutoML*) has recently attracted a lot of research interest, since the publication of [28]. However, previous research in this area has primarily focused on automated design of the representation learning component, and little attention has been paid to the input component. This is because most research has been conducted on image understanding problems [15, 18, 25, 29], where the representation learning component is very important to model performance, while the input component is trivial since the image pixels are already in floating point form.

For large scale recommendation problems commonly encountered in industry, the situation is quite different. While the representation learning component is important, the input component plays an even more critical role in the model. This is because many recommendation problems involve categorical features with large cardinality, and the input component assigns embedding vectors to each item of these discrete

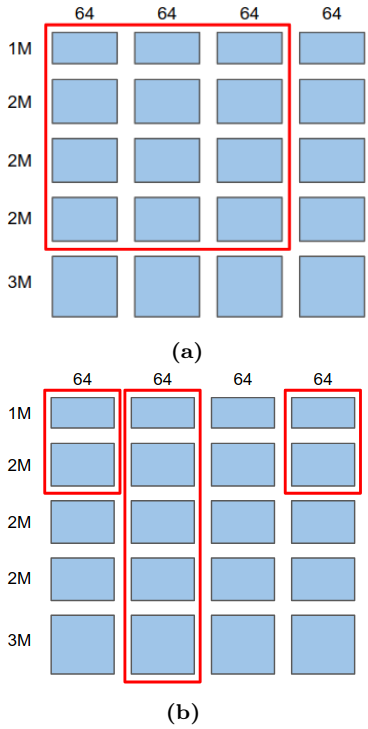

Figure 1: An example of Embedding Blocks and controller choice in a training step. (a) An embedding matrix of size $10M \times 256$ is discretized into 20 Embedding Blocks. The controller samples a $7M \times 192$ sized SE. (b) The controller samples $3M$ items for the first and the last 64-D space, and $7M$ items for the second 64-D space, resulting in an ME with the first $3M$ items have 192-D embeddings while the rest $7M$ have 64-D embeddings.

features. This results in a huge number of embedding parameters in the input component, which dominate both the size and the inductive bias of the model. For example, the YouTube video recommendation model ([7]) uses a video ID vocabulary of size 1 million, with 256 dimensional embedding vectors for each ID. This means 256 million parameters are used just for the video ID feature, and the number grows quickly as more discrete features are added. In contrast, the representation learning component consists of only three fully connected layers. So the number of model parameters is heavily concentrated in the input component, which naturally has high impact on model performance. In practice, despite their importance, vocabulary and embedding sizes for discrete features are often selected heuristically, by trying out a few models with different manually crafted configurations. Since these models are usually large and expensive to train, such an approach is computationally intensive and may result in suboptimal results.

In this paper, we propose **Neural Input Search** (NIS), a novel approach to find embedding and vocabulary sizes automatically for each discrete feature in the model's input component. We create a search space consisting of a collection of Embedding Blocks, where each combination of blocks represents a different vocabulary and embedding configuration. The optimal configuration is searched for in a single training run, using a Reinforcement Learning (RL) algorithm like ENAS [18]. Moreover, we propose a novel type of embedding, which we call **Multi-size Embedding (ME)**. ME allows allocating larger embedding vectors to more common or predictive feature items, and smaller vectors to less common or predictive ones. This is in contrast to the commonly employed approach, which we call Single-size Embedding (SE), where the same-sized embedding is used across all items in the vocabulary. We argue that SE is an inefficient use of the model's capacity and training data. This is because that we need a large embedding dimension for frequent or highly predictive items to encode their nuanced relation with other items, but training good embeddings of the same size for long tail items may take too many epochs due to their rarity in the training set. And when training data is limited, large-sized embeddings for rare items can overfit. With ME, given the same model capacity, we can cover more items in the vocabulary, while reducing the required training data size and computation cost for training good embeddings for long tail items.

Figure 1 gives a visual preview of the Embedding Blocks-based search space and how the RL controller samples a candidate SE and ME. The details of the NIS approach are discussed in Section 3.

We demonstrate the effectiveness of NIS at finding good configurations of vocabulary and embedding sizes for both SEs and MEs through experiments on two widely studied public datasets for recommendation problems. Given baseline recommendation models, NIS has been able to significantly improve their performance while at the same time significantly reduce the number of embedding parameters (thus the model size) across all experiments. Moreover, we applied NIS to a real world large scale App ranking model in our company's App store and conducted both offline and online A/B experiment, resulting in +1.02% App Install with 30% smaller model size. This new NIS App ranking model is deployed in production, and experiment details are reported.

2 RELATED WORK

Neural Architecture Search (NAS) has been an active research area since [28], which takes a Reinforcement Learning approach that requires training thousands of candidate models to convergence. Due to its resource intensive nature, a lot of research has focused on developing cheaper NAS methods. One active research direction is to design a large model that connects smaller model components, so that different candidate architectures can be expressed by selecting a subset of the components. The optimal set of components (and thus the architecture) is learned in a single training run. For example, ENAS ([18]) uses a controller to sample the submodels, and SMASH ([3]) generates weights for sampled networks using a hyper-network. DARTS ([15]) and SNAS ([25]) takes

a differentiable approach by representing the connection as a weight, which is optimized with backpropagation. A similar approach in combination of ScheduledDropPath ([29]) on the weights is taken in [2] and [5]. Luo et al. [16] takes another approach by mapping the neural architectures into an embedding space, where the optimal embedding is learned and decoded back to the final architecture.

Another research direction is to reduce the size of the search space. [4, 14, 19, 27] propose searching convolution cells, which are later stacked repeatedly into a deep network. Zoph et al. [29] developed the NASNet architecture and showed the cells learned from smaller datasets can achieve good results even on larger datasets in a transfer learning setting. MNAS [20] proposed a search space comprised of a hierarchy of convolution cell blocks, where cells in different blocks are searched separately and thus may results in different structures.

Almost all previous NAS research works have focused on finding the optimal representation learning component for image/video understanding problems. For large scale recommendation problems, great results have also been reported by leveraging advanced representation learning components, such as CNN ([12], [21]), RNN ([1], [8]), etc. However, the input component, although it contains a great portion of model parameters due to large-sized embeddings, has been frequently designed heuristically across industry, such as YouTube ([7]), Google Play ([6]), Netflix ([9]), etc. Our work, to the best of our knowledge, for the first time brings automated neural network design into the input component for large scale recommendation problems.

3 NEURAL INPUT SEARCH

3.1 Definitions and Notations

We assume that the model input consists of a set of categorical features $\mathcal{F}$. Each input example can contain any number of values per feature. For each feature $F \in \mathcal{F}$, we have a list of its possible values, sorted in decreasing order of frequency of occurrence in the dataset. This list implicity maps each feature value to an integer: we refer to this list as a vocabulary. An embedding variable E is a trainable matrix. If it's shape is $v \times d$, then v is referred to as the vocabulary size and d as the embedding dimension. For any $0 \leq i < v$, we use $E[i]$ to refer to the i^{th} row the embedding matrix E, i.e. the embedding vector of the i^{th} item within the vocabulary. Throughout the paper, we use $\mathcal{B}$ to refer to our 'memory budget', the total number of floating point values the embedding matrices of the model can use. A $v \times d$ shaped embedding matrix uses $v \times d$ values.

3.2 Neural Input Search Problems

We start with introducing our first proposed Neural Input Search problem based on the regular embedding matrix, which we call Single-size Embedding:

Single-size Embedding (SE). A single-size embedding is a regular embedding matrix with shape $v \times d$, where each of the v items within the vocabulary is represented as an d-dimensional vector. As stated in Section 1, most previous works use SEs to represent discrete features, and the value of v and d for each feature is selected in a heuristic manner, which can be suboptimal. Below we propose a Neural Input Search problem, namely NIS-SE, for automatically finding the optimal SE for each feature, and the approach for solving this problem is introduced later in Section 3.3.

Problem 1 (NIS-SE). *Find a vocabulary size v_F and embedding dimension d_F for each $F \in \mathcal{F}$ to maximize the objective function value of the resulting model, subject to:*

$$\sum_{F \in \mathcal{F}} v_F \times d_F \leq \mathcal{B}$$

The problem involves two trade-offs:

- Memory budget between features: More useful features should get a higher budget.
- Memory budget between vocabulary size and embedding dimension within each feature.

A large vocabulary for a feature gives us higher coverage, letting us include tail items as input signal. A large embedding dimension improves our predictions for head items, since head items have more training data and larger embeddings can encode more nuanced information. SE makes it difficult to simultaneously obtain high coverage and high quality embeddings within the memory budget. To conquer this difficulty, we introduce a novel type of embedding, namely Multi-size Embedding.

Multi-size Embedding (ME). Multi-size Embedding allows *different items in the vocabulary to have different sized embeddings*. It lets us use large embeddings for head items and small embeddings for tail items. It makes sense to have fewer parameters for tail items as they have lesser training data. The vocabulary and embedding size for a variable is now given by a Multisize Embedding Spec (MES). A MES is a list of pairs: $[(v_1, d_1), (v_2, d_2), \cdots (v_M, d_M)]$ for any $M \geq 1$ such that $v_m \in [1, v]$ and $\sum_{i=1}^{m} v_i \leq v$ for all $1 \leq m \leq M$, and $d_1 > d_2 > \cdots d_M \geq 1$. This can be interpreted as: the first v_1 most frequent items have embedding dimension d_1, the next v_2 frequent items have embedding dimension d_2, etc. The total vocabulary size is $v = \sum_{m=1}^{M} v_m$. When $M = 1$, an ME is equivalent to an SE.

Instead of having only one embedding matrix E like in a SE, we create one embedding matrix E_m of shape $v_m \times d_m$ for each $1 \leq m \leq M$. Moreover, a trainable projection matrix P_m of shape $d_m \times d_1$ is created for each $1 \leq m \leq M$, which maps a d_m-dimensional embedding to a d_1-dimensional space. This facilitates downstream reduction operations to be conducted in the same d_1-dimensional space. Define $V_0 = 0$ and $V_m = \sum_{i=1}^{m} v_i$ for $1 \leq m \leq M$ to be the cumulative vocabulary size for the first m embedding matrices, then the ME for k^{th} item in the vocabulary e_k is defined as

$$e_k = E_{m_k}[k - V_{m_k - 1}]P_{m_k}$$

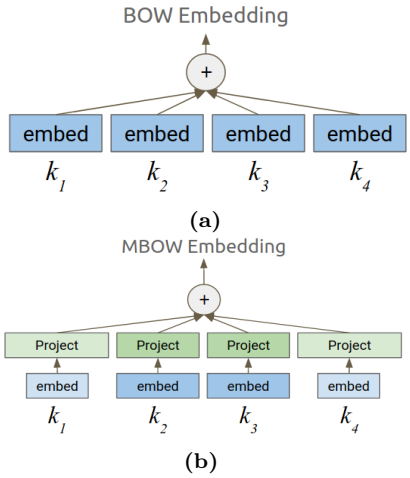

Figure 2: An example of BOW based on SE and ME. (a) BOW with SE: 4 items from the feature vocabulary are assigned with same sized embeddings, followed by a sum operator. (b) BOW with ME: k_2 and k_3 are assigned with larger sized embeddings than the other 2 items. These 4 embeddings are projected to the same space and summed.

where $m_k \in \{1, \cdots, M\}$ is chosen such that $k \in [V_{m_k-1}, V_{m_k})$, and clearly e_k is d_1-dimensional. We remind the readers that $E[i]$ represents the i^{th} row of the matrix E.

With an appropriate MES for each feature, ME is able to achieve high coverage on tail items and high quality representation of head items at the same time. However, finding the optimal MSE for all features manually is very hard, necessitating an automated approach for searching the right MESs. Below we introduce the Neural Input Search problem with Multi-size Embedding, namely NIS-ME, and the approach for solving this problem is introduced later in Section 3.3.

Problem 2 (NIS-ME). *Find a MES* $[(v_{F_1}, d_{F_1}), (v_{F_2}, d_{F_2}), \cdots (v_{F_{M_F}}, d_{F_{M_F}})]$ *for each* $F \in \mathcal{F}$ *to maximize the objective function value of the resulting model subject to:*

$$\sum_{F \in \mathcal{F}} \sum_{i=1}^{M_F} v_{F_i} \times d_{F_i} \leq \mathcal{C}$$

MEs can be used as a direct replacement for SEs in any model that uses embeddings. Typically, given a set of vocabulary IDs K, each element in K is mapped to its corresponding SE, followed by one or more reduce operations to these SEs. For example, a commonly used reduction operation is bag-of-words (BOW), where the embeddings are summed or averaged. To see how MEs can directly replace SEs in this case, the ME version of BOW, which we call MBOW, is given by:

$$\sum_{k \in K} e_k = \sum_{k \in K} (E_{m_k}[k - V_{m_k-1}] P_{m_k})$$

where the MEs are summed. This is illustrated in Figure 2. Note that for the k's whose m_k 's are equal, it is more efficient to sum the embeddings before applying the projection matrix.

3.3 Neural Input Search Approach

We now detail our method for solving Problems 1 and 2. As stated in the introduction, most large scale recommendation models are very expensive to train; it is desirable to solve each of these problems in one training run. Therefore, we leverage a variant of ENAS [18]: We develop a novel search space in the input component of the model, which contains the SEs or MEs we want to search over. A separate controller is used to make *choices* to pick an SE or ME for each discrete feature in each step. These selected SEs or MEs are trained in together with the rest of the main model (excluding the controller). In addition, we use the feedforward pass of the main model to compute a reward (a combination of accuracy and memory cost, detailed in Section 3.3.2) of the controller's choices, and the reward is used to train the controller variables using the A3C [17] policy gradient method.

3.3.1 Search Space. We now describe the search space, which is a key novel ingredient of our work.

Embedding Blocks. For a given feature $F \in \mathcal{F}$ with vocabulary size v, we create a grid of $S \times T$ matrices with $S > 1$ and $T > 1$, where the (s,t)-th matrix $E_{s,t}$ is of size $\bar{v}_s \times \bar{d}_t$, such that $v = \sum_{s=1}^{S} \bar{v}_s$, and $d = \sum_{t=1}^{T} \bar{d}_t$. Here d is the maximum allowed embedding size for any item within the vocabulary. We call these matrices Embedding Blocks. This can be thought as discretizing an embedding matrix of size $v \times d$ into $S \times T$ sub-matrices. As an example, suppose $v = 10\text{M}$ ('M' stands for million) and $d = 256$, we may discretize the rows into five chunks: $[1\text{M}, 2\text{M}, 2\text{M}, 2\text{M}, 3\text{M}]$, and discretize the columns into four chunks: $[64, 64, 64, 64]$, which results in 20 Embedding Blocks, as illustrated in Figure 1. Moreover, a projection matrix $\bar{P}_t$ of size $\bar{d}_t \times d$ is created for each $t = 1, \cdots, T$, in order to map each $\bar{d}_t$ dimensional embedding to a common d dimensional space for facilitating downstream reduction operations. Clearly we should have $\bar{v}_s >> d$ for all s so the number of parameters in the projection matrix are negligible compared to the number in the embeddings. The Embedding Blocks are the building blocks of the search space that allow the controller to sample different SEs or MEs at each training step.

Controller Choices. The controller is a neural network that samples different SEs or MEs from softmax probabilities. Its exact behavior depends on whether we are optimizing over SEs or MEs. Below we describe the controller's behavior on one feature $F \in \mathcal{F}$, and drop the F subscript for notational convenience.

SE: To optimize over SEs, at each training step, the controller samples one $(\bar{s}, \bar{t})$ pair from the set $\{(s,t) \mid 1 \leq s \leq S,\ 1 \leq t \leq T\} \cup \{(0,0)\}$. For a selected $(\bar{s}, \bar{t})$, only Embedding Blocks $\{E_{s,t} \mid 1 \leq s \leq \bar{s}, 1 \leq t \leq \bar{t}\}$ are involved in that particular training step. Therefore, the controller effectively picks an SE, such as the one within the red rectangle in Figure 1a, which represents an SE of size $5M \times 192$. The embedding of the k^{th} item in the vocabulary in this step is calculated as

$$e_k = \sum_{t=1}^{\bar{t}} E_{s_k,t}[k - \bar{V}_{s_k-1}]\bar{P}_t$$

for all $k < \bar{V}_{\bar{s}}$, where $\bar{V}_0 = 0$, $\bar{V}_s = \sum_{i=1}^{s} \bar{v}_i$ is the cumulative vocabulary size, and $s_k \in \{1, \cdots, S\}$ such that $\bar{V}_{s_k-1} \leq k < \bar{V}_{s_k}$. Define $\bar{D}_t = \sum_{i=1}^{t} d_t$ to be the cumulative embedding size, it is clear that e_k is equivalent to using a $D_{\bar{t}}$-dimensional embedding to represent the k^{th} item followed by a projection to a d-dimensional space, where the project matrix P is the concatenation of $\{\bar{P}_1, \cdots, \bar{P}_{\bar{t}}\}$ along the rows. Any item whose vocabulary id $k \geq \bar{V}_{\bar{s}}$ is considered as out-of-vocabulary and is handled specially; a commonly employed approach is using zero vector as their embedding. The corresponding memory cost (the number of parameters) induced by this choice of SE is therefore computed as $C = \bar{V}_{\bar{s}} \times \bar{D}_{\bar{t}}$ (the projection matrix cost is ignored, since $\bar{v}_s >> d$ for all s).

If the pair $(0, 0)$ is selected in a training step, it is equivalent to removing the feature from the model. Thus the zero embedding is used for all items of this feature within this training step, and the corresponding memory cost is 0. As the controller explores different SEs, it's trained based on the reward induced by each selection, and eventually converges to the optimal one, as described in Section 3.4. If it converges to the pair $(0, 0)$, it means this feature should be removed. Clearly, for a given feature, the search space size is $(S \times T + 1)$.

ME: When optimizing over MEs, instead of making a single choice, the controller makes a sequence of T choices, one for each $t \in 1, \cdots, T$. Each choice is an $\bar{s}_t \in \{1, \cdots, S\} \cup \{0\}$. If $\bar{s}_t > 0$, only Embedding Blocks $\{E_{s,t} \mid 1 \leq s \leq \bar{s}_t\}$ are involved in that particular training step. Similarly, if $\bar{s}_t = 0$, it means the whole $\bar{d}_t$-dimensional embedding is removed for all items within the vocabulary. Therefore, the controller picks a custom subset (not just a subgrid) of Embedding Blocks, which comprises an MES. This is visually illustrated in Figure 1b, where the first 64-D embeddings are utilized by the first 3M items, the second 64-D embeddings are utilized by all of the 10M items, the third 64-D embeddings are not used by any item, while the last 64-D embeddings have the same utilization as the first 64-D embeddings. As a result, the first 3M items in the vocabulary are allocated with 192 dimensional embeddings, while the last 7M items are assigned with only 64 dimensional embeddings. In other words, an MES $[(3M, 192), (7M, 64)]$ is realized at this training step.

Mathematically, let $\mathcal{T}_s = \{t \mid E_{s,t} \text{ is selected}\}$, then the embedding of the k^{th} item in the vocabulary in this step is calculated as

$$e_k = \sum_{t \in \mathcal{T}_{s_k}} E_{s_k,t}[k - \bar{V}_{s_k-1}]\bar{P}_t$$

for all $k < v$ whose corresponding $\mathcal{T}_{s_k}$ is non-empty, and e_k is an zero vector if $\mathcal{T}_{s_k}$ is empty. The calculation of memory cost is straightforward: $C = \sum_{t=1}^{T} \bar{d}_t \times \bar{V}_{\bar{s}_t}$.

Similar to the SE scenario, it is not difficult to see that, for a given feature, the search space size is $(S + 1)^T$.

3.3.2 Reward. As the main model is trained with the controller's choices of SEs or MEs, the controller is trained with the reward calculated from feedforward passes of the main model on validation set examples. We define the reward as

$$R = R_Q - \lambda * C_M,$$

where R_Q represents the (potentially non-differentiable) quality reward that we want to maximize, and C_M is the memory cost term to penalize the controller's selections of over-budget embedding configurations. λ is a coefficient whose meaning is explained below.

Quality Reward: One common class of recommendation problems is the retrieval problem. It aims at finding the M most relevant items out of a potentially very large vocabulary v, given the model's input. Achieving high recall is usually the objective of such problems; therefore, we can directly let the Recall@N ($N \leq M$) metric being the quality reward. When v is too large, computing Recall@N becomes too expensive, and we can use Sampled Recall@N as proxy by sampling a small negative set. Another common class of problems is the ranking problem. The quality of ranking models is usually measured by the Area Under the Receiver Operating Characteristic Curve (ROC-AUC), which can be used as R_Q. Similarly, for regression problems, R_Q can be set to the negative of the L_2-loss between the prediction and the label.

Memory Cost: We define the memory cost C_M as follows:

$$C_M = \max(\frac{\sum_{F \in \mathcal{F}} C_F}{\mathcal{B}} - 1, 0)$$

We remind the reader that C_F is the memory usage based on the controller's choices, which is defined in Section 3.3.1 (we dropped the subscript F in Section 3.3.1 to avoid cluttered notation). $\mathcal{B}$ is the pre-defined memory budget.

Given the definition of C_M, it is clear that λ represents the amount of increase of R_Q that worths the cost of $\mathcal{B}$ additional over-budget embedding parameters. For example, if R_Q is the ROC-AUC and $\lambda = 0.1$, it means that we are willing to afford $\mathcal{B}$ additional over-budget embedding parameters if it can incrase ROC-AUC by 0.1. This is because the additional $\mathcal{B}$ over-budget parameters decrease the reward by λ, so it is only worth the cost if it can increase R_Q by at least λ.

3.4 Training the Neural Input Search Model

3.4.1 The Warm up Phase. As stated in Section 3.3, training the NIS model involves an alternative training between the main model and the controller between consecutive steps. If we start training the controller from step 0, we get a vicious cycle where the Embedding Blocks not selected by the controller don't get enough training and hence give bad rewards, resulting in them being selected even less in future. To prevent this, the first several training steps consist of a *warm-up phase* where we train all the Embedding Blocks and leave the controller variables fixed. The warm up phase ensures that all Embedding Blocks get some training. After the warm up phase, we switch to training the main model and the controller in alternating steps using A3C.

3.4.2 The Baseline Network. As part of the A3C algorithm, we use a baseline network to predict the expected reward prior to each controller choice (but using the choices that have already been made). The baseline network has the same structure as the controller network, but has its own variables, which are trained alongside the controller variables using validation set. Then we subtract the baseline from the reward at each step to compute the advantage, which is used to train the controller.

4 EXPERIMENTS

In this section, we report experimental results from applying NIS to two public datasets and one real life large scale App recommendation model in our company's App store.

4.1 Experiments on Public Datasets

4.1.1 Dataset Description. We conduct experiments on two commonly used publicly accessible datasets, namely the MovieLens-1M dataset[1] and the KKBox's music recommendation dataset[2].

The MovieLens-1M dataset contains 1 million records of movie ratings created by more than 6k users. We formulate a movie recommendation problem by following a widely adapted experiment setting [10, 11, 26]: User ratings are treated as implicit feedback, i.e. the user is considered as being interested in the movie as long as they rated the movie. Moreover, for each user, 4 random movies are uniformly sampled from the movie vocabulary as negative examples for the user. Accidental hits are removed (see Section A.2.3 for details). Since each rating record is associated with a timestamp, we hold out the latest rating record of each user for test. During the test stage, for each positive example, 99 random movies are sampled as negative examples for calculating eval metrics; the sampling strategy is the same as training.

The KKBox dataset comes from the WSDM KKBox music recommendation challenge. The task is to predict if the user will listen to a song repeatedly after the first observable listening event within a time window. The dataset contains both positive examples (indicating the user repeatedly listened to the song) and negative examples (indicating the user didn't listen to the song again). Therefore, unlike the MovieLens-1M dataset, we didn't manually curate negative examples by random sampling; instead, we directly use the positive and negative examples provided in the dataset. Since the dataset does not have timestamp associated with each record, following [23], we randomly sample 80% records for training and 20% for test.

For both of the two datasets, we further randomly extract 10% examples from the training set to formulate the validation set for training the controller.

Table 1 lists the vocabulary size (i.e. the number of unique items) of features that we apply NIS on. Note that the original KKBox dataset has a few more features, which are either float-valued (e.g. "song length") or categorical with small vocabulary size (e.g. "genre"). Since NIS is more impactful to categorical features with large vocabulary size, in our experiments, we only use features listed in Table 1 for simplicity.

[1] https://grouplens.org/datasets/movielens/1m/
[2] https://www.kaggle.com/c/kkbox-music-recommendation-challenge/data

Table 1: The vocabulary size of each feature.

Dataset	Feature	Vocabulary size
MovieLens	User ID	6040
	Movie ID	3706
KKBox	User ID	30755
	Song ID	359914
	Artist Name	42479
	Composer	82846
	Lyricist	38933

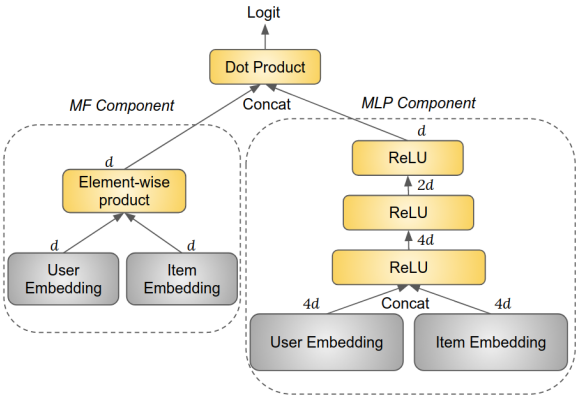

Figure 3: The NCF model in the experiments. Yellow boxes represent computational layers, e.g. ReLU represents a fully connected layer with ReLU activation. The dimensionality of the output of each layer is specified on top of each box.

4.1.2 Experiment Setting. The NIS method is practically agnostic to model structures. In our experiments, we use the Neural Collaborative Filtering (NCF) model [10] on both of the two datasets, since it has been widely applied and thoroughly studied in many research works (e.g. [11, 13, 22]).

The NCF model comprises two components, i.e. the Matrix Factorization (MF) component and the Multi-Layer Perceptron (MLP) component, as depicted in Figure 3. In our experiments, the user and item embeddings in the MF component are d-dimensional, whose element-wise multiplication generates the d-dimensional MF embedding. The user and item embeddings in the MLP component are $4d$-dimensional, which are concatenated and fed into 3 MLP layers with sizes being $4d$, $2d$ and d respectively. The output of the top MLP layer is called the MLP embedding. The concatenation of the d-dimensional MF embedding and MLP embedding is used to compute the logit by dot-producting with a $2d$-dimensional trainable weight. Cross-entropy loss is used.

Table 2: Experimental results on the MovieLens dataset. Metrics are reported as percentages.

Model	d	$\mathcal{B}$	# Parameters	Recall@1	Recall@5	Recall@10	MRR@5	MRR@10	NDCG@5	NDCG@10
Baseline	8	N/A	390K	8.48	33.18	51.95	16.77	19.26	20.81	26.86
NIS-SE	16	390K	367K	9.32	35.70	55.31	18.22	20.83	22.43	28.63
NIS-SE	16	195K	261K	8.42	31.37	50.30	15.04	17.57	19.59	25.12
NIS-ME	16	390K	384K	9.41	35.90	55.68	18.31	20.95	22.60	28.93
NIS-ME	16	195K	243K	8.57	33.29	52.91	16.78	19.37	20.83	27.15
Baseline	16	N/A	780K	9.77	34.62	54.25	17.50	20.46	21.63	28.33
NIS-SE	32	780K	636K	9.90	37.50	55.69	19.18	21.60	23.70	29.58
NIS-SE	32	390K	518K	9.79	34.84	53.23	17.85	20.26	22.00	27.91
NIS-ME	32	780K	789K	10.40	38.66	57.02	19.59	22.18	24.28	30.60
NIS-ME	32	390K	496K	10.19	37.44	56.62	19.56	22.09	23.98	30.14

Table 3: Experimental results on the KKBox dataset.

Model	d	$\mathcal{B}$	# Parameters	AUC (%)
Baseline	8	N/A	22.2M	73.7
NIS-SE	16	22M	23.4M	74.4
NIS-SE	16	11M	17.6M	74.1
NIS-ME	16	22M	18.2M	74.5
NIS-ME	16	11M	15.9M	74.4
Baseline	16	N/A	44.4M	74.3
NIS-SE	32	44M	45.6M	74.4
NIS-SE	32	22M	31.5M	74.3
NIS-ME	32	44M	48.1M	75.1
NIS-ME	32	22M	30.0M	74.4

For the MovieLens-1M dataset, the item embedding is simply the movie embedding. For the KKBox dataset, since each item (i.e. song) has multiple features, namely the song ID, artist name, composer, and lyricist, the item embedding is computed as follows: For the MF component, each of the four feature is represented as a d-dimensional embedding, whose concatenation is mapped to a d dimensional vector by a $4d \times d$-sized matrix; this vector represents the item embedding in the MF component. Similarly, for the MLP component, each feature's embedding size is $4d$, whose concatenation is mapped to $4d$ dimensional space by a $16d \times 4d$ matrix.

Given a pre-selected d, which results in a model with m_d embedding parameters, we first benchmark it as the baseline without applying NIS. Then, we double the embedding size of each feature, which results in $2m_d$ parameters, and apply NIS with memory budget $\mathcal{B} = m_d$, i.e. the same as the baseline model. We expect NIS to perform better than the baseline by better allocating the m_d parameters. We further push the limit of NIS with $\mathcal{B} = 0.5m_d$ and see if it can still perform better than the baseline model. We experimented with $d = 8$ and $d = 16$ for the baseline model, thus the corresponding NIS models would have $d = 16$ and $d = 32$.

4.1.3 Controller. The controller is used to generate a probability distribution over all candidate choices. We used a simple controller in all experiments, where for the SE setting, a $(S \times T + 1)$-dimensional vector is assigned to each feature, and each value in the vector represents the logit to the multinomial distribution, from which different SE are sampled. We remind the reader that the search space size is $(S \times T + 1)$ for a feature (Section 3.3.1). The vector is initialized to zero vector for uniform distribution. For a model with $|\mathcal{F}|$ features, the controller simply consists with $|\mathcal{F}|$ independent vectors, thus the SEs of different features are sampled independently. We didn't use a more sophisticated controller structure such as an RNN [28], because it is not clear if the decision made for one feature should affect the others, and in which order the decisions should be made.

Similarly, for the ME setting, T independent $(S + 1)$-dimensional vectors are associated with each feature, since T independent choices need to be made. When there are $|\mathcal{F}|$ features, $T \times |\mathcal{F}|$ independent vectors comprise the controller.

20 Embedding Blocks are constructed for each feature, with $\bar{d}_t$'s being $[0.25d, 0.25d, 0.25d, 0.25d]$, and $\bar{v}_s$'s being $[0.2v, 0.2v, 0.2v, 0.2v, 0.2v]$, where v is the total vocabulary size of the feature. In practice, we found this configuration gives good balance between fine-graineess and the size of the search space. The controller is trained using the validation set, as stated Section 4.1.1.

For the MovieLens-1M dataset, since it is formulated as a retrieval-type problem with sampled negatives, we used Recall@1 (also called HitRate@1 in some literatures [24, 26]) as the quality reward R_Q, and we report multiple Recall, MRR and NDCG metrics calculated from the eval set. For the KKBox dataset, since we formulated it as a ranking problem with ground truth negatives (instead of sampled negatives), we used ROC-AUC as reward and report ROC-AUC on the eval set. In all experiments, we set $\lambda = 0.01$ to penalize over-budget memory usage.

4.1.4 Result. The experimental results on the MovieLens-1M dataset and the KKBox dataset are reported in Table 2 and Table 3, respectively. We remind the reader that $\mathcal{B}$ represents the memory budget to the NIS models.

It can be seen through the results that NIS is able to consistently achieve better performance than the baseline model,

very often with signficantly less parameters, evidencing the effective of our approach. A few observations can be made:

- The NIS-ME model always outperforms NIS-SE given the same $\mathcal{B}$ and usually with even fewer parameters than NIS-SE.
- The NIS models almost always outperform the baseline even when $\mathcal{B}$ is half of the baseline's model size. This not only demonstrates the effectiveness of our approach, but also shows that it is possible to achieve superior performance under heavy constraints.
- Some NIS models received superior performance by exceeding the provided memory budget. Such a tradeoff between model quality and size is reflected in the design of reward function. This is particularly useful when the memory budget is only a guideline and not strict.

4.2 Experiments on a Real World Large Scale Dataset

In this section, we describe how we apply NIS to a ranking model in Google Play App store, one of the largest App store in the world.

4.2.1 Problem Description. We applied NIS to a ranking model that was serving in production. The model's objective is to rank a set of Apps based on the likelihood they will be installed. The dataset used to train the model consists (Context, App, Label) tuples, where the Label is either 0 or 1, indicating if the App is installed or not. A total of 20 discrete features are used to represent the Context and App, such as App ID, Developer ID, App Title, etc. The vocabulary size of the discrete features varies from hundreds to millions. The embedding dimension and vocabulary size of each feature has been heavily optimized manually over several years.

4.2.2 Experiment Setting. When preparing the Embedding Blocks, we set $\bar{v}_s = [0.1v, 0.2v, 0.2v, 0.2v, 0.3v]$, where v is the vocabulary size of the feature. Note that instead of evenly splitting the vocabulary, for each feature, we allocated an Embedding Block to the first 10% items, since most features in the dataset are top heavy, i.e. the first 10% items usually appear in most of the training examples. This further justifies the necessity of Multi-size Embedding in practice. Moreover, we set $\bar{d}_t = [0.25D, 0.25D, 0.25D, 0.25D]$, where $D = 3d$ with d being the embedding size that was used in the production model, ranging from 8 to 32 acrossing different features. While tripling the embedding size to allow higher model capacity, we set the memory budget $\mathcal{B} = 0.5\mathcal{B}_p$, where $\mathcal{B}_p$ is the number of embedding parameters of the production model. Clearly, the objective here is to increase the model's prediction power, while at the same time reducing the model size.

Since it is a ranking model with ground truth negative examples, we used ROC-AUC as R_Q when calculating the reward. By setting $\lambda = 0.001$, the controller can decide to use $0.5\mathcal{B}_p$ more parameters in exchange for a 0.001 increase of ROC-AUC.

4.2.3 Offline Experiment Results. For offline experiments, we evaluate the ROC-AUC metric and report the results in Table 4. As expected, we observed that NIS is able to improve the product model's performance on ROC-AUC with less number parameters in both SE and ME setting. It is not surprising to see that the NIS-ME model is able to achieve superior performance over NIS-SE with even less number of parameters, since Multi-size Embedding can better use the memory budget by giving more parameters to head items and fewer parameters to tail items. Comparing with the production model, NIS-ME received 0.4% improvement on ROC-AUC with 30% model size reduction.

Table 4: Comparison between NIS and production App ranking model.

Model	AUC (%)	# Parameters
Production	77.4	160M
NIS-SE	77.6	151M
NIS-ME	77.8	124M

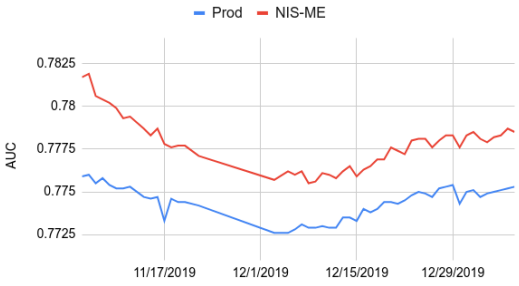

Figure 4: A 2-month long comparison of ROC-AUC between the production and the NIS model.

4.2.4 Online A/B Testing Results. We further conducted online experiment with A/B testing using live traffic. The A/B testing is only performed on the NIS-ME model due to its superiority over the NIS-SE model. We monitored the App Install metric and concluded that the NIS-ME model is able to increase App Install by 1.02%. The NIS-ME model is currently deployed with 100% traffic, replacing the production baseline model used in this experiment.

4.2.5 Stability. When deployed in production, the NIS-ME model needs to be re-trained and refreshed everyday. It is important to know that how long the model's performance can sustain. This is because that the data distribution of each feature may change significantly over time, so that the MES may not be the optimal anymore, in which case we need to re-run NIS to find a MES that better fits the new data distribution.

We conducted a 2-month study and monitored the ROC-AUC of the original production model and the NIS-ME model, which is shown in Figure 4. Clearly, the NIS-ME model's advantage over the production model has been very stable over the 2-month period, indicating there is no need to re-run NIS very often. In practice, we only run NIS when there is a model architecture change or new features are to be added.

5 CONCLUSION

We presented **Neural Input Search** (NIS), a technique for automatically searching the optimal vocabulary and embedding sizes in the input component of a model. We also introduced **Multi-size Embedding** (ME), a novel type of embedding that achieves high coverage of tail items while keeping accurate representation for head items. We demonstrated the effectiveness of NIS and ME with experiments on both public datasets and real world large scale recommendation dataset. NIS is able to find embedding configurations that lead to significant improvement on model performance over the baseline models with a much smaller model size.

REFERENCES

[1] Trapit Bansal, David Belanger, and Andrew McCallum. 2016. Ask the GRU: Multi-task Learning for Deep Text Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. ACM, New York, NY, USA, 107–114. https://doi.org/10.1145/2959100.2959180

[2] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. 2018. Understanding and Simplifying One-Shot Architecture Search. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). Stockholmsmässan, Stockholm Sweden, 550–559.

[3] Andrew Brock, Theo Lim, J.M. Ritchie, and Nick Weston. 2018. SMASH: One-Shot Model Architecture Search through HyperNetworks. In *International Conference on Learning Representations*.

[4] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. 2018. Path-Level Network Transformation for Efficient Architecture Search. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). Stockholmsmässan, Stockholm Sweden, 678–687.

[5] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations*.

[6] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (Boston, MA, USA) *(DLRS 2016)*. ACM, New York, NY, USA, 7–10. https://doi.org/10.1145/2988450.2988454

[7] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. ACM, New York, NY, USA, 191–198. https://doi.org/10.1145/2959100.2959190

[8] Tim Donkers, Benedikt Loepp, and Jürgen Ziegler. 2017. Sequential User-based Recurrent Neural Network Recommendations. In *Proceedings of the Eleventh ACM Conference on Recommender Systems* (Como, Italy) *(RecSys '17)*. ACM, New York, NY, USA, 152–160. https://doi.org/10.1145/3109859.3109877

[9] Carlos A. Gomez-Uribe and Neil Hunt. 2015. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Trans. Manage. Inf. Syst.* 6, 4, Article 13 (Dec. 2015), 19 pages. https://doi.org/10.1145/2843948

[10] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[11] Binbin Hu, Chuan Shi, Wayne Xin Zhao, and Philip S Yu. 2018. Leveraging meta-path based context for top-n recommendation with a neural co-attention model. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1531–1540.

[12] Donghyun Kim, Chanyoung Park, Jinoh Oh, Sungyoung Lee, and Hwanjo Yu. 2016. Convolutional Matrix Factorization for Document Context-Aware Recommendation. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. ACM, New York, NY, USA, 233–240. https://doi.org/10.1145/2959100.2959165

[13] Dawen Liang, Rahul G Krishnan, Matthew D Hoffman, and Tony Jebara. 2018. Variational autoencoders for collaborative filtering. In *Proceedings of the 2018 World Wide Web Conference*. 689–698.

[14] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive Neural Architecture Search. In *The European Conference on Computer Vision (ECCV)*.

[15] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.

[16] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. 2018. Neural Architecture Optimization. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7816–7827.

[17] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). PMLR, New York, New York, USA, 1928–1937.

[18] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameters Sharing. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 4095–4104.

[19] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2018. Regularized Evolution for Image Classifier Architecture Search. *CoRR* abs/1802.01548 (2018). arXiv:1802.01548

[20] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. 2018. MnasNet: Platform-Aware Neural Architecture Search for Mobile. *CoRR* abs/1807.11626 (2018). arXiv:1807.11626

[21] Aaron van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep content-based music recommendation. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 2643–2651.

[22] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. 2019. Neural graph collaborative filtering. In *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*. 165–174.

[23] Xiang Wang, Dingxian Wang, Canran Xu, Xiangnan He, Yixin Cao, and Tat-Seng Chua. 2019. Explainable reasoning over knowledge graphs for recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 5329–5336.

[24] Xiang Wang, Dingxian Wang, Canran Xu, Xiangnan He, Yixin Cao, and Tat-Seng Chua. 2019. Explainable reasoning over knowledge graphs for recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 5329–5336.

[25] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2019. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*.

[26] Xin Xin, Xiangnan He, Yongfeng Zhang, Yongdong Zhang, and Joemon Jose. 2019. Relational collaborative filtering: Modeling multiple item relations for recommendation. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 125–134.

[27] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. 2018. Practical Block-Wise Neural Network Architecture Generation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[28] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations*.

[29] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

A APPENDIX

In this appendix, we provide data processing and modeling details to help reproduce our experiments on public datasets. All our experiments are conducted with TensorFlow[3].

A.1 Data Processing Details

In this section, we describe how we process the public datasets and transform them into TFRecord examples for training and testing.

A.1.1 MovieLens-1M Dataset. The MovieLens-1M dataset was processed by the following steps:

- Traverse each user-movie rating record, and count how many records each user appeared in the entire dataset. Assigns vocabulary ID to each user based on the count, i.e. the most frequent user is assigned to vocabulary ID 0, and the least frequent user is assigned to vocabulary ID 6039. Similarly, count the number of appearances of each movie and assigns vocabulary ID to each movie.
- Index each user-movie rating record by user id, and formulate a movie list for each user. Sort the list by timestamp, and use the latest rating record (the one with the largest timestamp) as test example. The rest are used as training examples. The resulting training and test dataset contains user-movie pairs from all users, where users and movies are represented by their vocabulary ID.
- Shuffle the training set. Then, for each user-movie pair, convert them into a TFRecord protocol buffer record[4]. Do the same to the test set. Each TFRecord example thus contains two int64 features, one for user vocabulary ID, another one for movie vocabulary ID.
- Randomly sample 10% records out of the training set as the validation set for training the controller.
- Store the training, validation, and test set in files.

Note that the resulting dataset only has positive examples, since we treat each rating as implicit positive feedback from the user. The sampling of negative examples is conducted in the TensorFlow graph and discussed in Section A.2.3.

A.1.2 KKBox Dataset. We used two files from the original KKBox data:

- The *train.csv* file that contains the user-song records, each of which is associated with user ID, song ID, and the label (1.0 or 0.0).
- The *songs.csv* file that contains features of each song, and we used artist name, composer and lyricist.

Note that we did not use the *test.csv* file provided in the original data since it does not have label and is supposed to be used for submission to the WSDM competition.

The KKBox dataset was processed by the following steps:

- Traverse each user-music record in *train.csv*, look up the corresponding 3 features in the *songs.csv* file, and combine them with the record.
- Build the 5 feature vocabularies in the same way as described in Section A.1.1. This results in vocabulary for the user ID, song ID, artist name, composer, and lyricist.
- Assigns vocabulary ID to all the feature values in the entire datset.
- Convert each record into a TFRecord protocol buffer, which contains 6 features, i.e. 5 int64 features and 1 float feature for the label.
- Shuffle the dataset, and randomly sample 20% as test set. The rest 80% is the training set.
- Randomly sample 10% records out of the training set as the validation set for training the controller.
- Store the training, validation, and test set in files.

A.2 Model Configurations

In this section, we describe the model configurations in detail.

A.2.1 The NCF Model. The NCF model was developed with the following specifications:

- The structure of the NCF model we used in the experiment is show in Figure 3. As stated in Section 4.1.2, the MLP component contains 3 ReLu layers. We did not use any normalization technique, such as BatchNorm, LayerNorm, etc.
- All weights, including hidden layer weights and embedding matrices, were randomly initialized with Gaussian distribution, whose mean is 0 and standard deviation is $1/\sqrt{n}$, where n is either the hidden layer size or the embedding dimension.
- No regularization or dropout was used in any weights.
- When there are multiple values for a feature, e.g. multiple composers for a song, the embeddings of these feature values are averaged. When there are missing feature values in some training examples, zero-valued embedding is used.

A.2.2 Training. We first describe the training details of the main model.

- The main model was trained with mini-batch examples whose batch size is 256.
- We applied the `tf.clip_by_global_norm()` function to all variables in the main model to clip the gradient. The `clip_norm` parameter is set to 1.0. With gradient clipping, we set the learning rate to 0.1 when training the main model. Adagrad optimizer was used.
- Before starting the A3C algorithm, the warm up phase (Section 3.4.1) was conducted for $100K$ steps.

The controller was trained with the following specifications:

- The Adagrad optimizer was used with learning rate 0.1.
- Unlike the main model, we did not use gradient clipping when training the controller. This is because that a low probability but high reward action should get a very high gradient to significantly increase its probability; clipping the gradient may unnecessarily reduce the probability increase.

[3] https://www.tensorflow.org/
[4] https://www.tensorflow.org/tutorials/load_data/tfrecord

- Each controller decision is shared with 64 examples from the validation set, from which one reward value is calculated. For example, when computing Recall@1 for the MovieLens dataset, 64 recall values are generated (with the same controller choices) and averaged, which is used as the reward for training the controller. Similarly, for the KKBox dataset, 64 prediction values are used to calculate one ROC-AUC reward.
- The controller and the main model are trained alternative every other mini-batch step, after the warm up phase is finished.

In all experiments, we stop the training after both the main model and the controller are converged. This varies from 1 million to 3 million steps. We used a distributed training setting with 5 workers training independently in parallel.

A.2.3 Negative Sampling for the MovieLens Dataset. As stated above, for the MovieLens dataset, we sample negative examples within the TensorFlow graph during the training process. Specifically, in each step, we used the `tf.random.uniform_candidate_sampler()` function to sample N movie vocabulary IDs out of the entire movie vocabulary, where $N = 4$ for training (both the main model and the controller), and $N = 99$ for test. The `unique` parameter is set to `True`. The same N negatives are shared within the mini-batch example.

It is possible that the sampled vocabulary IDs, which are used as negative examples, contain the positive classes within the mini-batch examples, which is usually called "accidental hits". We used the `tf.nn.compute_accidental_hits()` function to detect the accidental hits, and set the corresponding logit to `-FLOAT_MAX`. This is equivalent to removing these training examples from the mini-batch.

We adopted this negative sampling strategy instead of preparing fixed negative examples for each positive example before the training starts, because this approch provides negative examples more thoroughly for every positive example. In fact, this approach has been widely used in real life recommendation models in industry, and we were following the same practice in our experiments.