Product-based Neural Networks for User Response Prediction over Multi-field Categorical Data

YANRU QU, BOHUI FANG, and WEINAN ZHANG, Shanghai Jiao Tong University, China
RUIMING TANG, Noah's Ark Lab, Huawei, China
MINZHE NIU, Shanghai Jiao Tong University, China
HUIFENG GUO*, Shenzhen Graduate School, Harbin Institute of Technology, China
YONG YU, Shanghai Jiao Tong University, China
XIUQIANG HE†, Data service center, MIG, Tencent, China

User response prediction is a crucial component for personalized information retrieval and filtering scenarios, such as recommender system and web search. The data in user response prediction is mostly in a multi-field categorical format and transformed into sparse representations via one-hot encoding. Due to the sparsity problems in representation and optimization, most research focuses on feature engineering and shallow modeling. Recently, deep neural networks have attracted research attention on such a problem for their high capacity and end-to-end training scheme. In this paper, we study user response prediction in the scenario of click prediction. We first analyze a coupled gradient issue in latent vector-based models and propose kernel product to learn field-aware feature interactions. Then we discuss an insensitive gradient issue in DNN-based models and propose Product-based Neural Network (PNN) which adopts a feature extractor to explore feature interactions. Generalizing the kernel product to a net-in-net architecture, we further propose Product-network In Network (PIN) which can generalize previous models. Extensive experiments on 4 industrial datasets and 1 contest dataset demonstrate that our models consistently outperform 8 baselines on both AUC and log loss. Besides, PIN makes great CTR improvement (relatively 34.67%) in online A/B test.

CCS Concepts: • **Information systems → Information retrieval**; • **Computing methodologies → Supervised learning**; • **Computer systems organization → Neural networks**;

Additional Key Words and Phrases: Deep Learning, Recommender System, Product-based Neural Network

ACM Reference Format:
Yanru Qu, Bohui Fang, Weinan Zhang, Ruiming Tang, Minzhe Niu, Huifeng Guo, Yong Yu, and Xiuqiang He. 2017. Product-based Neural Networks for User Response Prediction over Multi-field Categorical Data. *ACM Transactions on Information Systems* 9, 4, Article 39 (October 2017), 35 pages. https://doi.org/

*This work is done when Huifeng Guo was an intern at Noah's Ark Lab, Huawei
†This work is done when Xiuqiang He worked at Noah's Ark Lab, Huawei

Authors' addresses: Yanru Qu; Bohui Fang; Weinan Zhang, Shanghai Jiao Tong University, China, kevinqu@apex.sjtu.edu.cn, fangbohui@sjtu.edu.cn, wnzhang@sjtu.edu.cn; Ruiming Tang, Noah's Ark Lab, Huawei, China, tangruiming@huawei.com; Minzhe Niu, Shanghai Jiao Tong University, China, nmzfrank@apex.sjtu.edu.cn; Huifeng Guo, Shenzhen Graduate School, Harbin Institute of Technology, China, huifengguo@yeah.net; Yong Yu, Shanghai Jiao Tong University, China, yyu@apex.sjtu.edu.cn; Xiuqiang He, Data service center, MIG, Tencent, China, xiuqianghe@tencent.com.

1046-8188/2017/10-ART39 $15.00
https://doi.org/

1 INTRODUCTION

Predicting a user's response to some item (e.g., movie, news article, advertising post) under certain context (e.g., website) is a crucial component for personalized information retrieval (IR) and filtering scenarios, such as online advertising [30, 35], recommender system [24, 36], and web search [2, 5].

The core of personalized service is to estimate the probability that a user will "like", "click", or "purchase" an item, given features about the user, the item, and the context [31]. This probability indicates the user's interest in the specific item and influences the subsequent decision-making such as item ranking [48] and ad bidding [52]. Taking online advertising as an example, the estimated Click-Through Rate (CTR) will be utilized to calculate a bid price in an ad auction to improve the advertisers' budget efficiency and the platform revenue [32, 35, 52]. Hence, it is much desirable to gain accurate prediction to not only improve the user experience, but also boost the volume and profit for the service providers.

Table 1. An example of multi-field categorical data.

TARGET	WEEKDAY	GENDER	CITY
1	Tuesday	Male	London
0	Monday	Female	New York
1	Tuesday	Female	Hong Kong
0	Tuesday	Male	Tokyo
Number	7	2	1000

The data collection in these tasks is mostly in a multi-field categorical form. And each data instance is normally transformed into a high-dimensional sparse (binary) vector via one-hot encoding [18]. Taking Table 1 as an example, the 3 fields are one-hot encoded and concatenated as

$$\underbrace{[0,1,0,0,0,0,0]}_{\text{WEEKDAY=Tuesday}}\ \underbrace{[1,0]}_{\text{GENDER=Male}}\ \underbrace{[0,0,1,0,\ldots,0,0]}_{\text{CITY=London}}.$$

Each field is represented as a binary vector, of which only 1 entry corresponding to the input is set as 1 while others are 0. The dimension of a vector is determined by its field size, i.e., the number of unique categories[1] in that field. The one-hot vectors of these fields are then concatenated together in a predefined order.

Without loss of generality, user response prediction can be regarded as a binary classification problem, and 1/0 are used to denote positive/negative responses respectively [14, 37]. A main challenge of such a problem is sparsity. For parametric models, they usually convert the sparse (binary) input into dense representations (e.g., weights, latent vectors), and then search for a separable hyperplane. Fig. 1 shows the model decomposition. In this paper, we mainly focus on modeling and training, thus we exclude preliminary feature engineering [9].

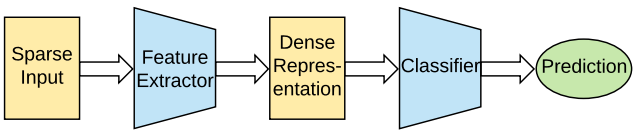

Fig. 1. Model decomposition.

[1] For clarity, we use "category" instead of "feature" to represent a certain value in a categorical field. For consistency with previous literature, we preserve "feature" in some terminologies, e.g., feature combination, feature interaction, and feature representation.

Many machine learning models are leveraged or proposed to work on such a problem, including linear models, latent vector-based models, tree models, and DNN-based models. Linear models, such as Logistic Regression (LR) [25] and Bayesian Probit Regression [14], are easy to implement and with high efficiency. A typical latent vector-based model is Factorization Machine (FM) [36]. FM uses weights and latent vectors to represent categories. According to their parametric representations, LR has a linear feature extractor, and FM has a bi-linear[2] feature extractor. The prediction of LR and FM are simply based on the sum over weights, thus their classifiers are linear. FM works well on sparse data, and inspires a lot of extensions, including Field-aware FM (FFM) [21]. FFM introduces field-aware latent vectors, which gain FFM higher capacity and better performance. However, FFM is restricted by space complexity. Inspired by FFM, we find a **coupled gradient issue** of latent vector-based models and refine feature interactions[3] as field-aware feature interactions. To solve this issue as well as saving memory, we propose kernel product methods and derive *Kernel FM* (KFM) and *Network in FM* (NIFM).

Trees and DNNs are potent function approximators. Tree models, such as Gradient Boosting Decision Tree (GBDT) [6], are popular in various data science contests as well as industrial applications. GBDT explores very high order feature combinations in a non-parametric way, yet its exploration ability is restricted when feature space becomes extremely high-dimensional and sparse. DNN has also been preliminarily studied in information system literature [8, 33, 40, 51]. In [51], FM supported Neural Network (FNN) is proposed. FNN has a pre-trained embedding[4] layer and several fully connected layers. Since the embedding layer indeed performs a linear transformation, FNN mainly extracts linear information from the input. Inspired by [39], we find an **insensitive gradient issue** that fully connected layers cannot fit such target functions perfectly.

From the model decomposition perspective, the above models are restricted by inferior feature extractors or weak classifiers. Incorporating product operations in DNN, we propose *Product-based Neural Network* (PNN). PNN consists of an embedding layer, a product layer, and a DNN classifier. The product layer serves as the feature extractor which can make up for the deficiency of DNN in modeling feature interactions. We take FM, KFM, and NIFM as feature extractors in PNN, leading to *Inner Product-based Neural Network* (IPNN), *Kernel Product-based Neural Network* (KPNN), and *Product-network In Network* (PIN).

CTR estimation is a fundamental task in personalized advertising and recommender systems, and we take CTR estimation as the working example to evaluate our models. Extensive experiments on 4 large-scale real-world datasets and 1 contest dataset demonstrate the consistent superiority of our models over 8 baselines [6, 15, 21, 25, 27, 36, 46, 51] on both AUC and log loss. Besides, PIN makes great CTR improvement (34.67%) in online A/B test. To sum up, our contributions can be highlighted as follows:

- We analyze a coupled gradient issue of latent vector-based models. We refine feature interactions as field-aware feature interactions and extend FM with kernel product methods. Our experiments on KFM and NIFM successfully verify our assumption.
- We analyze an insensitive gradient issue of DNN-based models and propose PNNs to tackle this issue. PNN has a flexible architecture which can generalize previous models.
- We study optimization, regularization, and other practical issues in training and generalization. In our extensive experiments, our models achieve consistently good results in 4 offline datasets, 1 contest, and 1 online A/B test.

[2] Although FM has higher-order formulations [36], due to the efficiency and practical performance, FM is usually implemented with second-order interactions.
[3] In [36], the cross features learned by FM are called feature interactions.
[4] We use "latent vector" in shallow models, and "embedding vector" in DNN models.

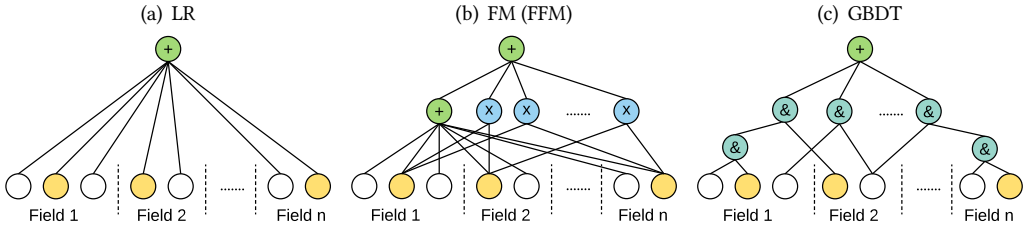

Fig. 2. Illustration of LR, FM, FFM, and GBDT. *Note:* Yellow node means one-hot input of a field; green '+' node means addition operation; blue '×' node means multiplication operation; cyan '&' node means feature combination.

The rest of this paper is organized as follows. In Section 2, we introduce related work in user response prediction and other involved techniques. In Section 3 and 4, we present our PNN models in detail and discuss several practical issues. In Section 5, we show offline evaluation, parameter study, online A/B test, and synthetic experiments respectively. We finally conclude this paper and discuss future work in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 User Response Prediction

User response prediction is normally formulated as a binary classification problem with prediction log-likelihood or cross-entropy as the training objective [1, 14, 37]. area under ROC curve (AUC), log loss and relative information gain are common evaluation metrics [14]. Due to the one-hot encoding of categorical data, sparsity is a big challenge in user response prediction.

From the modeling perspective, linear Logistic Regression (LR) [25, 35], bi-linear Factorization Machine (FM) [36] and Gradient Boosting Decision Tree (GBDT) [18] are widely used in industrial applications. As illustrated in Fig. 2, LR extracts linear information from the input, FM further extracts bi-linear information, while GBDT explores feature combinations in a non-parametric way. From the training perspective, many adaptive optimization algorithms can speed up training of sparse data, including Follow the Regularized Leader (FTRL) [30], Adaptive Moment Estimation (Adam) [22], etc. These algorithms follow a per-coordinate learning rate scheme, making them converge much faster than stochastic gradient descent (SGD).

From the representation perspective, latent vectors are expressive in representing categorical data. In FM, the side information and user/item identifiers are represented by low-dimensional latent vectors, and the feature interactions are modeled as the inner product of latent vectors. As an extension of FM, Field-aware FM (FFM) [21] enables each category to have multiple latent vectors. From the classification perspective, powerful function approximators like GBDT and DNN are more suitable for continuous input. Therefore, in many contests, the winning solutions take FM/FFM as feature extractors to process discrete data, and use the latent vectors or interactions as the input of successive classifiers (e.g., GBDT, DNN). According to model decomposition (Fig. 1), latent vector-based models make predictions simply based on the sum of interactions. This weakness motivates the DNN variants of latent vector-based models.

2.2 DNN-based Models

With the great success of deep learning, it is not surprising there emerge some deep learning techniques for recommender systems [50]. Primary work includes: (i) Pretraining autoencoders to extract feature representations, e.g., Collaborative Deep Learning [44]. (ii) Using DNN to model

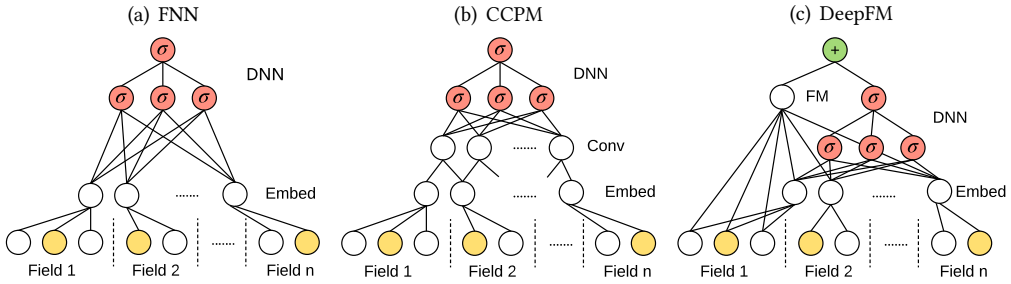

Fig. 3. Illustration of FNN, DeepFM, and CCPM. Yellow node means one-hot input of a field; green '+' node means add operation; red 'σ' node means nonlinear operation.

general interactions [16, 17, 33]. (iii) Using DNN to process images/texts in content-based recommendation [45]. We mainly focus on (ii) in this paper.

The input to DNN is usually dense and numerical, while the case of multi-field categorical data has not been well studied. FM supported Neural Network (FNN) [51] (Fig. 3(a)) has an embedding layer and a DNN classifier. Besides, FNN uses FM to pre-train the embedding layer. Other models use DNN to improve FM, e.g., Neural Collaborative Filtering (NCF) [17], Neural FM (NFM) [16], Attentional FM (AFM) [46]. NCF uses DNN to solve collaborative filtering problem. NFM extends NCF to more general recommendation scenarios. Based on NFM, AFM uses attentive mechanism to improve feature interactions, and becomes a state-of-the-art model.

Convolutional Click Prediction Model (CCPM) [27] (Fig. 3(b)) uses convolutional layers to explore local-global dependencies. CCPM performs convolutions on the neighbor fields in a certain alignment, but fails to model convolutions among non-neighbor fields. RNNs are leveraged to model historical user behaviors [53]. In this paper, we do not consider sequential patterns.

Wide & Deep Learning (WDL) [7] trains a wide model and a deep model jointly. The wide part uses LR to "memorize", meanwhile, the deep part uses DNN to "generalize". Compared with single models, WDL achieves better AUC in offline evaluations and higher CTR in online A/B test. WDL requires human efforts for feature engineering on the input to the wide part, thus is not end-to-end. DeepFM [15], as illustrated in Fig. 3(c), can both utilize the strengths of WDL and avoid expertise in feature engineering. It replaces the wide part of the WDL with FM. Besides, the FM component and the deep component share same embedding parameters. DeepFM is regarded as one state-of-the-art model in user response prediction.

To complete our discussion of DNN-based models, we list some less relevant work, such as the following. Product Unit Neural Network [11] defines the output of each neuron as the product of all its inputs. Multilinear FM [28] studies FM in a multi-task setting. DeepMood [4] presents a neural network view for FM and Multi-view Machine.

2.3 Net-in-Net Architecture

Network In Network (NIN) [26] is originally proposed in CNN. NIN builds micro neural networks between convolutional layers to abstract the data within the receptive field. Multilayer perceptron as a potent function approximator is used in micro neural networks of NIN. GoogLeNet [42] makes use of the micro neural networks suggested in [26] and achieves great success. NIN is powerful in modeling local dependencies. In this paper, we borrow the idea of NIN, and propose to explore inter-field feature interactions with flexible micro networks.

3 METHODOLOGY

As in Fig. 1, the difficulties in learning multi-field categorical data are decomposed into 2 phases: representation and classification. Following this idea, we first explain field-aware feature interactions, then we study the deficiency of DNN classifiers, finally, we present our Product-based Neural Networks (PNNs) in detail.

A commonly used objective for user response prediction is to minimize cross entropy, or namely log loss, defined as

$$\mathcal{L}(y, \sigma(\hat{y})) = -y \log(\sigma(\hat{y})) - (1 - y) \log(1 - \sigma(\hat{y})) \,, \tag{1}$$

where $y \in \{0, 1\}$ is the label and $\sigma(\hat{y}) \in (0, 1)$ is the predicted probability of $y = 1$, more specifically, the probability of a user giving a positive response on the item. We adopt this training objective in all experiments.

3.1 Field-aware Feature Interactions

In user response prediction, the input data contains multiple fields, e.g., WEEKDAY, GENDER, CITY. A field contains multiple categories and takes one category in each data instance. Table 1 shows 4 data examples, each of which contains 3 fields, and each field takes a single value. For example, a MALE customer located in LONDON buys some beer on TUESDAY. From this record we can extract a useful feature combination: "MALE and LONDON and TUESDAY implies True". The efficacy of feature combinations (a.k.a., cross features) has already been proved [31, 36]. In FM, the 2nd order combinations are called feature interactions.

Assume the input data has n categorical fields, $\mathbf{x} \in \mathbb{N}^n$, where x_i is an ID indicating a category of the i-th field. The feature representations learned by parametric models could be weight coefficients (e.g., in LR) or latent vectors (e.g., in FM). For an input instance $\mathbf{x}$, each field is converted into corresponding weight $x_i \rightarrow w_i$ or latent vector $x_i \rightarrow \mathbf{v}_i$. For an output $\hat{y}$, the probability is obtained from sigmoid function $\hat{y} \rightarrow \sigma(\hat{y}) = 1/(1 + \exp(-\hat{y}))$. For simplicity, we use w_i and $\mathbf{v}_i$ to represent the input, and we use $\hat{y}$ to represent the output.

3.1.1 A Coupled Gradient Issue of Latent Vector-based Models. The prediction of FM [36] can be formulated as

$$\hat{y}_{FM} = \sum_{i=1}^{n} w_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle + b \,, \tag{2}$$

where $w_i \in \mathbb{R}$ is the weight of category x_i, $\mathbf{v}_i \in \mathbb{R}^k$ is the latent vector of x_i, and $b \in \mathbb{R}$ is the global bias. Take the first example in Table 1,

$$\hat{y}_{FM} = w_{\text{Male}} + w_{\text{London}} + w_{\text{Tue}} + \langle \mathbf{v}_{\text{Male}}, \mathbf{v}_{\text{London}} \rangle + \langle \mathbf{v}_{\text{London}}, \mathbf{v}_{\text{Tue}} \rangle + \langle \mathbf{v}_{\text{Male}}, \mathbf{v}_{\text{Tue}} \rangle + b \,.$$

The gradient of the latent vector $\mathbf{v}_{\text{Male}}$ is $\nabla_{\mathbf{v}_{\text{Male}}} \hat{y}_{FM} = \mathbf{v}_{\text{London}} + \mathbf{v}_{\text{Tue}}$. FM makes an implicit assumption that a field interacts with different fields in the same manner, which may not be realistic. Suppose GENDER is independent of WEEKDAY, it is desirable to learn $\mathbf{v}_{\text{Male}} \perp \mathbf{v}_{\text{Tue}}$. However, the gradient $\nabla_{\mathbf{v}_{\text{Male}}} \hat{y}_{FM}$ continuously updates $\mathbf{v}_{\text{Male}}$ in the direction of $\mathbf{v}_{\text{Tue}}$. Conversely, the latent vector $\mathbf{v}_{\text{Tue}}$ is updated in the direction of $\mathbf{v}_{\text{Male}}$. To summarize, FM uses the same latent vectors in different types of inter-field interactions, which is an over-simplification and degrades the model capacity. We call this problem a coupled gradient issue.

The gradients of latent vectors could be decoupled by assigning different weights to different interactions, such as the attentive mechanism in Attentional FM (AFM) [46]:

$$\hat{y}_{AFM} = \sum_{i=1}^{n} w_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} f(\mathbf{v}_i, \mathbf{v}_j)\langle \mathbf{v}_i, \mathbf{v}_j \rangle + b \,, \tag{3}$$

where f denotes the attention network which takes embedding vectors as input and assigns weights to interactions, $f(\mathbf{v}_i, \mathbf{v}_j) \in \mathbb{R}$. In AFM, the gradient of $\mathbf{v}_{\text{Male}}$ becomes $\nabla_{\mathbf{v}_{\text{Male}}} \hat{y}_{AFM} = f(\mathbf{v}_{\text{Male}}, \mathbf{v}_{\text{London}})\mathbf{v}_{\text{London}} + f(\mathbf{v}_{\text{Male}}, \mathbf{v}_{\text{Tue}})\mathbf{v}_{\text{Tue}} + others$, where the gradients of $\mathbf{v}_{\text{Male}}$ and $\mathbf{v}_{\text{Tue}}$ are decoupled when $f(\mathbf{v}_{\text{Male}}, \mathbf{v}_{\text{Tue}})$ approaches 0. However, when the attention score approaches 0, the attention network becomes hard to train.

This problem is solved by Field-aware FM (FFM) [21]

$$\hat{y}_{FFM} = \sum_{i=1}^{n} w_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \langle \mathbf{v}_{i,j}, \mathbf{v}_{j,i} \rangle + b \,, \tag{4}$$

where $\mathbf{v}_i \in \mathbb{R}^{n \times k}$, meaning the i-th category has n independent k-dimensional latent vectors when interacting with n fields. Excluding intra-field interactions, we usually use $\mathbf{v}_i \in \mathbb{R}^{(n-1) \times k}$ in practice. Using field-aware latent vectors, the gradients of different interactions are decoupled, e.g., $\nabla_{\mathbf{v}_{\text{Male, CITY}}} \hat{y}_{FFM} = \mathbf{v}_{\text{London, GENDER}}$, $\nabla_{\mathbf{v}_{\text{Male, WEEKDAY}}} \hat{y}_{FFM} = \mathbf{v}_{\text{Tue, GENDER}}$. This leads to the main advantage of FFM over FM and brings a higher capacity.

FFM makes great success in various data science contests. However, it has a memory bottleneck, because its latent vectors have $O(Nnk)$ parameters (FM is $O(Nk)$), where N, the total number of categories, is usually in million to billion scales in practice. When Nn is large, k must be small enough to fit FFM in memory, which severely restricts the expressive ability of latent vectors. This problem is also discussed in Section 5.3 through visualization. To tackle the problems of FM and FFM, we propose kernel product.

3.1.2 Kernel Product. Matrix factorization (MF) learns low-rank representations of a matrix. A matrix A can be represented by the product of two low-rank matrices P and Q, i.e., $A = PQ^\top$. MF estimates each observed element $A_{i,j}$ with the inner product of two latent vectors $\mathbf{p}_i$ and $\mathbf{q}_j$. The target of MF is to find optimal latent vectors which can minimize the empirical error

$$\hat{A}_{i,j} = \langle \mathbf{p}_i, \mathbf{q}_j \rangle \tag{5}$$

$$P^*, Q^* = \arg\min_{P,Q} \sum_{i,j} \mathcal{L}(A_{i,j}, \hat{A}_{i,j}) \,, \tag{6}$$

where $\langle \mathbf{p}, \mathbf{q} \rangle = \sum_s p_s q_s$ is the inner product of two vectors, $\mathcal{L}$ represents the loss function (e.g., root mean square error, log loss).

MF has another form, $A = UDV^\top$, where D can be regarded as a projection matrix. U and V factorize A in the projected space like P and Q do in the original space. We define the inner product in a projected space, namely kernel product, $\langle \mathbf{p}, \mathbf{q} \rangle_\phi = \mathbf{p}^\top \phi \mathbf{q}$, then we can extend MF

$$\hat{A}^{\phi}_{i,j} = \langle \mathbf{p}_i, \mathbf{q}_j \rangle_\phi \tag{7}$$

$$P^*, Q^*, \phi^* = \arg\min_{P,Q,\phi} \sum_{i,j} \mathcal{L}_{\mathcal{F}}(A_{i,j}, \hat{A}^{\phi}_{i,j}) \,, \tag{8}$$

where $\phi \in \mathcal{F}$ is a projection matrix, namely kernel, and $\mathcal{F}$ is the parameter space. Vector inner product can be regarded as a special case of kernel product when $\phi = I$. MF can be regarded as a

special case of kernel MF when $\mathcal{F} = \{I\}$. Kernel product also generalizes vector outer product. The convolution sum of an outer product is equivalent to a kernel product

$$\mathbf{p}\mathbf{q}^\top \odot \phi = \sum_{s=1}^{k}\sum_{t=1}^{k} p_s \phi_{s,t} q_t = \mathbf{p}^\top \phi \mathbf{q}\,, \tag{9}$$

where $\odot$ denotes convolution sum. It is worth noting that, the outer product form $\mathbf{p}\mathbf{q}^\top \odot \phi$ has $2k^2$ multiplication and k^2 addition operations, while the kernel product form $\mathbf{p}^\top \phi \mathbf{q}$ has $k^2 + k$ multiplication and $k^2 + k$ addition operations. Therefore, kernel product generalizes both vector inner product and outer product. In Eq. (8), the kernel matrix is optimized in a parameter space, and a kernel matrix maps two vectors to a real value. From this point, a kernel is equivalent to a function. We can define kernel product in parameter or function spaces to adjust to different problems. In this paper, we study (i) linear kernel, and (ii) micro network kernel.

In practice, field size (number of categories in a field) varies dramatically (e.g., GENDER=2, CITY=7). Field size reflects the amount of information contained in one field. It is natural to represent a large (small, respectively) field in a large (small, respectively) latent space, and we call it adaptive embedding. In [8], an adaptive embedding size is decided by the logarithm of the field size. The challenge is how to combine adaptive embeddings with MF, since inner product requires the latent vectors to have the same length. Kernel product can solve this problem

$$\langle \mathbf{p}, \mathbf{q} \rangle_\phi = \sum_{s=1}^{k_1}\sum_{t=1}^{k_2} p_s \phi_{s,t} q_t\,, \tag{10}$$

where $\mathbf{p} \in \mathbb{R}^{k_1}$, $\mathbf{q} \in \mathbb{R}^{k_2}$, and $\phi \in \mathbb{R}^{k_1 \times k_2}$.

3.1.3 Field-aware Feature Interactions. The coupled gradient issue of latent vector-based models can be solved by field-aware feature interactions. FFM learns field-aware feature interactions with field-aware latent vectors. However, the space complexity of FFM is $O(Nnk)$, which restricts FFM from using large latent vectors. A relaxation of FFM is projecting latent vectors into different kernel spaces. Corresponding to $O(n^2)$ inter-field interactions, the $O(n^2)$ kernels require $O(n^2k^2)$ extra space. Since $n^2k^2 \ll Nk$, the total space complexity is still $O(Nk)$. In this paper, we extend FM with (i) linear kernel, and (ii) micro network kernel.

Kernel FM (KFM):

$$\hat{y}_{KFM} = \sum_{i=1}^{n} w_i + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle_{\phi_{i,j}} + b\,, \tag{11}$$

where $\phi_{i,j} \in \mathbb{R}^{k \times k}$ is the kernel matrix of field i and j.

Network in FM (NIFM):

$$\hat{y}_{NIFM} = \sum_{i=1}^{n} w_i + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} f_{i,j}(\mathbf{v}_i, \mathbf{v}_j) + b \tag{12}$$

$$f_{i,j}(\mathbf{v}_i, \mathbf{v}_j) = \sigma([\mathbf{v}_i, \mathbf{v}_j]^\top \mathbf{w}^1_{i,j} + \mathbf{b}^1_{i,j})^\top \mathbf{w}^2_{i,j}\,, \tag{13}$$

where $f_{i,j}$ denotes a micro network taking latent vectors as input and producing feature interactions with nonlinearity. In Eq. (13), the micro network output has no bias term because it is redundant with respect to the global bias b. This model is inspired by net-in-net architecture [26, 42]. With flexible micro networks, we can control the complexity and take the advantages of enormous deep learning techniques.

Recall the first example in Table 1. Suppose GENDER is independent of WEEKDAY, we can have $\langle \mathbf{v}_{\text{Male}}, \mathbf{v}_{\text{Tue}} \rangle_{\phi_{\text{GENDER,WEEKDAY}}} = 0$ if (i) $\phi_{\text{GENDER,WEEKDAY}} \mathbf{v}_{\text{Tue}} \perp \mathbf{v}_{\text{Male}}$, or (ii) $\phi_{\text{GENDER,WEEKDAY}} = 0$. Thus, the gradients of latent vectors are decoupled. Comparing KFM with FM/FFM: (i) KFM bridges FM with FFM. (ii) The capacity of KFM is between FM and FFM, because KFM shares kernels among certain types of inter-field interactions. (iii) KFM re-parametrizes FFM, which is "time for space". Comparing KFM/NIFM with AFM, AFM uses a universal attention network which is field-sharing, while KFM/NIFM use multiple kernels which are field-aware. If we share one kernel among all inter-field interactions, it will become an attention network. Thus, KFM/NIFM generalize AFM. Comparing kernel product with CNN, their kernels are both used to extract feature representations. Besides, kernel product shares projection matrices/functions among interactions, and CNN shares kernels in space.

3.2 Training Feature Interactions with Trees or DNNs is Difficult

In the previous section, we propose kernel product to solve the coupled gradient issue in latent vector-based models. On the other hand, trees and DNNs can approximate very complex functions. In this section, we analyze the difficulties of trees and DNNs in learning feature interactions.

3.2.1 A Sparsity Issue of Trees. Growing a decision tree performs greedy search among categories and splitting points [34]. Tree models encounter a sparsity issue in multi-field categorical data. Here gives an example. A tree with depth 10 has at most 10^3 leaf nodes ($2^{10} \approx 10^3$), and a tree with depth 20 has at most 10^6 leaf nodes ($2^{20} \approx 10^6$). Suppose we have a dataset with 10 categorical fields, each field contains 100 categories, and the input dimension is 10^3 after one-hot encoding. This dataset has $C_{10}^1 100$ categories, $C_{10}^2 100^2$ 2nd order feature combinations, and $C_{10}^{10} 100^{10}$ full order feature combinations. From this example, we can see that even a very deep tree model can only explore a small fraction of all possible feature combinations on such a small dataset. Therefore, modeling capability of tree models, e.g., Decision Tree, Random Forest and Gradient Boosting Decision Tree (GBDT) [6], is restricted in multi-field categorical settings.

3.2.2 An Insensitive Gradient Issue of DNNs. Gradient-based DNNs refer to the DNNs based on gradient descent and backpropagation. Despite the universal approximation property [19], there is no guarantee that a DNN naturally converges to any expected functions using gradient-based optimization. In user response prediction, the target function can be represented by a set of rules, e.g., "Male and London and Tuesday implies True". Here we show the periodic property of the target function via parity check. Recall the examples in Table 1, a feasible classifier is $parity(\mathbf{x}, \mathbf{v}), \mathbf{v} \in \mathcal{H}$, where $\mathbf{x}$ is the input, $\mathbf{v} = \{\text{Male}, \text{London}, \text{Tuesday}\}$ is the checking rule, and $\mathcal{H}$ is the hypothesis space. $\mathbf{x}_1 = \{\text{Male}, \text{London}, \text{Tuesday}\}$ is accepted by the predictor because 3 (which is odd) conditions are matched and $\mathbf{x}_3 = \{\text{Female}, \text{Hong Kong}, \text{Tuesday}\}$ is also accepted because 1 (which is also odd) condition is matched. In contrast, $\mathbf{x}_2 = \{\text{Male}, \text{Tokyo}, \text{Tuesday}\}$ and $\mathbf{x}_4 = \{\text{Female}, \text{New York}, \text{Monday}\}$ are rejected since 2 and 0 (which are even) conditions are matched. Two examples are shown in Fig. 4.

From this example, we observe that, in multi-field categorical data: (i) Basic rules can be represented by feature combinations, and several basic rules induce a parity check. (ii) The periodic property reveals that a feature set giving positive results does not conclude its superset nor its subset being positive. (iii) Any target functions should also reflect the periodic property.

A recent work [39] proves an insensitive gradient issue of DNN: (i) If the target is a large collection of uncorrelated functions, the variance (sensitivity) of DNN gradient to the target function decreases linearly with $|\mathcal{H}|$. (ii) When variance decreases, the gradient at any point will be extremely concentrated around a fixed point independent of $\mathbf{v}$. (iii) When the gradient is independent of the target $\mathbf{v}$, it contains little useful information to optimize DNN, thus gradient-based optimization

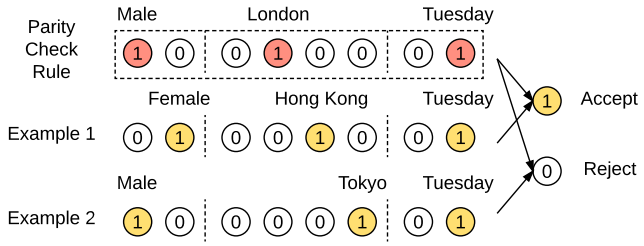

Fig. 4. Two examples of parity check from Table 1.

fails to make progress. The authors in [39] use the variance of gradient w.r.t. hypothesis space to measure the useful information in gradient, and explain the failure of gradient-based deep learning with an example of parity check.

Considering the large hypothesis space of DNNs, we conclude that learning feature interactions with gradient-based DNNs is difficult. In another word, DNNs can hardly learn feature interactions implicitly or automatically. We conduct a synthetic experiment to support this idea in Section 5.4. And we propose product layers to help DNNs tackle this problem.

3.3 Product-based Neural Networks

3.3.1 DNN-based Models. For consistency, we introduce DNN-based models according to the 3 components: the embedding module, the interaction module, and the DNN classifier

$$\hat{y}_{DNN} = \text{net}(\text{interact}(\text{embed}(\mathbf{x}))) \ . \tag{14}$$

AFM [46] has already been discussed in Section 3.1. AFM uses an attention network to improve FM, yet its prediction is simply based on the sum of interactions,

$$\hat{y}_{AFM} = \text{sum}(\text{attend}(\text{embed}(\mathbf{x}))) \ . \tag{15}$$

FM supported Neural Network (FNN) [51] (Fig. 3(a)) is formulated as

$$\hat{y}_{FNN} = \text{net}(\text{embed}(\mathbf{x})) \ , \tag{16}$$

where embed$(\cdot)$ is initialized from a pre-trained FM model. Compared with shallow models, FNN has a powerful DNN classifier, which gains it significant improvement. However, without the interaction module, FNN may fail to learn expected feature interactions automatically.

Similarly, Convolutional Click Prediction Model (CCPM) [27] (Fig. 3(b)) is formulated as

$$\hat{y}_{CCPM} = \text{net}(\text{conv}(\text{embed}(\mathbf{x}))) \ , \tag{17}$$

where conv$(\cdot)$ denotes convolutional layers which are expected to explore local-global dependencies. CCPM only performs convolutions on the neighbor fields in a certain alignment but fails to model the full convolutions among non-neighbor fields.

DeepFM [15] (Fig. 3(c)) improves Wide & Deep Learning (WDL) [7]. It replaces the wide model with FM and gets rid of feature engineering expertise.

$$\hat{y}_{DeepFM} = \hat{y}_{FM} + \hat{y}_{FNN} \ . \tag{18}$$

Note that the embedding vectors of the FM part and the FNN part are shared in DeepFM. WDL and DeepFM follows a joint training scheme. In another word, other single models can also be integrated into a mixture model, yet the joint training scheme is beyond our discussion.

FNN has a linear feature extractor (without pre-training) and a DNN classifier. CCPM additionally explores local/global dependencies with convolutional layers, but the exploration is limited in

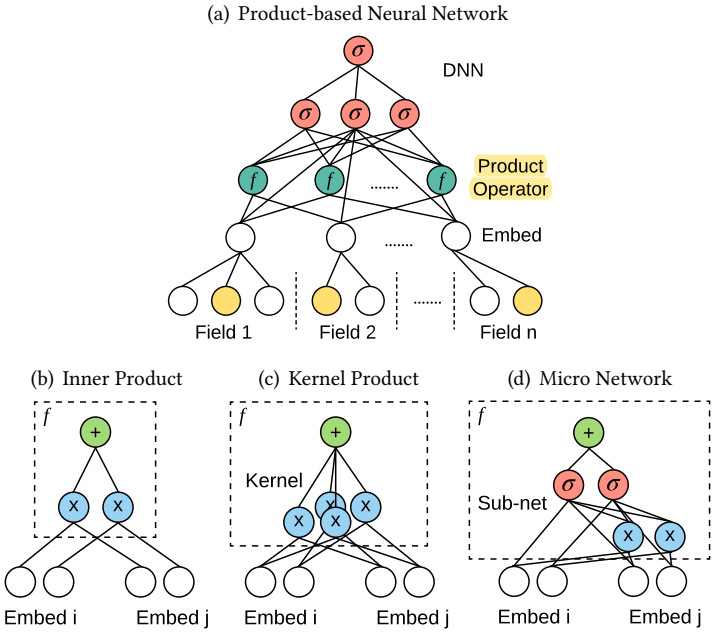

Fig. 5. Product-based Neural Networks. *Note:* Yellow node means one-hot input of a field; blue '×' node means multiply operation; green '+' node means add operation; red 'σ' node means nonlinear operation; cyan 'f' node means some product operator. PNN uses multiple product operators in different inter-field interactions. There are 3 types of product operators: (b) inner product, (c) kernel product, and (d) micro network. In (b)-(d), we use 2 white nodes to represent an embedding vector of length 2.

neighbor fields. DeepFM has a bi-linear feature extractor, yet the bi-linear feature representations are not fed to its DNN classifier. The insensitive gradient issue of DNN-based models has already been discussed in Section 3.2. To solve this issue, we propose Product-based Neural Network (PNN). The general architecture of PNN is illustrated in Fig. 5(a).

$$\mathbf{v} = \text{embed}(\mathbf{x}) \tag{19}$$

$$\mathbf{p} = \text{product}(\mathbf{v}) \tag{20}$$

$$\hat{y}_{PNN} = \text{net}(\mathbf{v}, \mathbf{p}) \tag{21}$$

The embedding layer (19) is field-wisely connected. This layer looks up the corresponding embedding vector for each field $x_i \rightarrow \mathbf{v}_i$, and produces dense representations of the sparse input, $\mathbf{v}^\top = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$. The product layer (20) uses multiple product operators to explore feature interactions $\mathbf{p}^\top = [\mathbf{p}_{1,2}, \ldots, \mathbf{p}_{n-1,n}]$. The DNN classifier (21), takes both the embeddings $\mathbf{v}$ and the interactions $\mathbf{p}$ as input, and conduct the final prediction $\hat{y}_{PNN}$.

Using FM, KFM, and NIFM as feature extractors, we develop 3 PNN models: Inner Product-based Neural Network (IPNN), Kernel Product-based Neural Network (KPNN), and Product-network In Network (PIN), respectively. We decompose all discussed parametric models in Table 2. A component level comparison is in Table 3, e.g., FM + kernel product → KFM.

One may concern the complexity, initialization, or training of PNNs. As for the complexity, there are $O(n^2)$ interactions, yet the complexity may not be a bottleneck: (i) In practice, n is usually a small number. In our experiments, the datasets involved contain at most 39 fields. (ii) The computation of

Table 2. Model decomposition.

	linear features	feature interactions	field-aware feature interactions
linear classifier	LR	FM	FFM, KFM, NIFM
DNN classifier	FNN (w/o pre-train), CCPM	DeepFM, IPNN	KPNN, PIN

Table 3. Component-level comparison.

	FM	KFM	NIFM	FNN	KPNN	PIN
embedding	yes	yes	yes	yes	yes	yes
kernel		yes			yes	
net-in-net			yes			yes
DNN				yes	yes	yes

interactions is independent and can speed up via parallelization. PNN concatenates embeddings and interactions as the DNN input. The embeddings and the interactions follow different distributions, which may cause problems in initialization and training. One solution to this potential risk is careful initialization and normalization. These practical issues are discussed in Section 4, and corresponding experiments are in Section 5.2.

3.3.2 Inner Product-based Neural Network. IPNN uses FM as the feature extractor, where the feature interactions are defined as inner products of the embeddings, as illustrated in Fig. 5(b). The n embeddings of $\mathbf{v}$, and the $n(n-1)/2$ interactions of $\mathbf{p}$ are flattened and fully connected with the successive hidden layer

$$\mathbf{p}^\top = [\langle \mathbf{v}_1, \mathbf{v}_2 \rangle, \ldots, \langle \mathbf{v}_{n-1}, \mathbf{v}_n \rangle] \tag{22}$$

$$\hat{y}_{IPNN} = \text{net}([\mathbf{v}_1, \ldots, \mathbf{v}_n, p_{1,2}, \ldots, p_{n-1,n}]) \, . \tag{23}$$

Comparing IPNN with Neural FM (Section 2.2), their inputs to DNN classifiers are quite different: (i) In Neural FM, the interactions are summed up and passed to DNN. (ii) In IPNN, the interactions are concatenated and passed to DNN. Since AFM has no DNN classifier, it is compared with KFM/NIFM in Section 3.1. Comparing IPNN with FNN: (i) FNN explores feature interactions through pre-training). (ii) IPNN explores feature interactions through the product layer. Comparing IPNN with DeepFM: (i) DeepFM adds up the feature interactions to the model output. (ii) IPNN feeds the feature interactions to the DNN classifier.

3.3.3 Kernel Product-based Neural Network. KPNN utilizes KFM as the feature extractor, where the feature interactions are defined as kernel products of the embeddings, as illustrated in Fig. 5(c). Since kernel product generalizes outer product, we use kernel product as a general form. The formulation of KPNN is similar to IPNN, except that

$$\mathbf{p}^\top = [\langle \mathbf{v}_1, \mathbf{v}_2 \rangle_{\phi_{1,2}}, \ldots, \langle \mathbf{v}_{n-1}, \mathbf{v}_n \rangle_{\phi_{n-1,n}}] \tag{24}$$

$$\hat{y}_{KPNN} = \text{net}([\mathbf{v}_1, \ldots, \mathbf{v}_n, p_{1,2}, \ldots, p_{n-1,n}]) \, . \tag{25}$$

3.3.4 Product-network In Network. A micro network is illustrated in Fig. 5(d)[5]. In PIN, the computation of several sub-network[6] forward passes are merged into a single tensor multiplication

$$\mathbf{H}_1^\top = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m] \tag{26}$$

$$\begin{aligned}\mathbf{H}_2^\top &= \sigma([\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m]^\top \mathbf{H}_1) \\ &= \sigma([\mathbf{w}_1^\top \mathbf{v}_1, \mathbf{w}_2^\top \mathbf{v}_2, \ldots, \mathbf{w}_m^\top \mathbf{v}_m]) \,,\end{aligned} \tag{27}$$

where $\mathbf{v}_i \in \mathbb{R}^{d_1}$ is the input to sub-network i, and d_1 is the input size. $\mathbf{H}_1$ concatenates all $\mathbf{v}_i$ together, $\mathbf{H}_1 \in \mathbb{R}^{m \times d_1}$, where m is the number of sub-networks. The weights $\mathbf{w}_i$ of sub-network i are also concatenated to a weight tensor $[\mathbf{w}_1, \ldots, \mathbf{w}_m] \in \mathbb{R}^{d_2 \times m \times d_1}$, where d_2 is the output size of a sub-network. Applying tensor multiplication on dimension d_1 and element-wise nonlinearity σ, we get $\mathbf{H}_2 \in \mathbb{R}^{m \times d_2}$.

Layer normalization (LN) [3] is proposed to stabilize the activation distributions of DNN. In PIN, we use fused LN on sub-networks to stabilize training. For each data instance, LN collects statistics from different neurons and normalizes the output of these neurons. However, the sub-networks are too small to provide stable statistics. Fused LN instead collects statistics from all sub-networks, thus is more stable than LN. More detailed discussions are in Section 4.4, and corresponding experiments are in Section 5.2.

$$\text{LN}(\mathbf{w}_i^\top \mathbf{v}_i) = \frac{\mathbf{w}_i^\top \mathbf{v}_i - \text{mean}_{k=1}^{d_2}(\mathbf{w}_i^\top \mathbf{v}_i)}{\text{std}_{k=1}^{d_2}(\mathbf{w}_i^\top \mathbf{v}_i)} \mathbf{g}_i + \mathbf{b}_i \tag{28}$$

$$\text{fused-LN}(\mathbf{w}_i^\top \mathbf{v}_i) = \frac{\mathbf{w}_i^\top \mathbf{v}_i - \text{mean}_{i=1,k=1}^{m,d_2}(\mathbf{w}_i^\top \mathbf{v}_i)}{\text{std}_{i=1,k=1}^{m,d_2}(\mathbf{w}_i^\top \mathbf{v}_i)} \mathbf{g} + \mathbf{b} \,. \tag{29}$$

Replacing $\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ with $\{\mathbf{v}_{1,2}, \ldots, \mathbf{v}_{n-1,n}\}$, the PIN model is presented as follows,

$$\mathbf{v}_{i,j} = [\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_i \odot \mathbf{v}_j] \tag{30}$$

$$f_{i,j}(\mathbf{v}_i, \mathbf{v}_j) = \sigma(\mathbf{v}_{i,j}^\top \mathbf{w}_{i,j}^1 + \mathbf{b}_{i,j}^1)^\top \mathbf{w}_{i,j}^2 + \mathbf{b}_{i,j}^2 \tag{31}$$

$$\mathbf{p}_{i,j} = f_{i,j}(\mathbf{v}_i, \mathbf{v}_j), \; j \neq i \tag{32}$$

$$\hat{y}_{PIN} = \text{net}([\mathbf{p}_{1,2}, \ldots, \mathbf{p}_{n-1,n}]) \,, \tag{33}$$

where $\odot$ denotes element-wise product instead of convolution sum. To stabilize micro network outputs, LN can be inserted into the hidden layers of the micro networks.

Compared with NIFM, the sub-networks of PIN are slightly different. (i) Each sub-network takes an additional product term as the input. (ii) The sub-network bias $\mathbf{b}_2$ is necessary because there is no global bias like NIFM. (iii) The sub-network output is a scaler in NIFM, however, it could be a vector in PIN. Compared with other PNNs, the embedding vectors are no longer fed to the DNN classifier because the embedding-DNN connections are redundant. The embedding-DNN connections:

$$[\mathbf{v}_1, \ldots, \mathbf{v}_n]^\top [\mathbf{w}_1, \ldots, \mathbf{w}_n] = \sum_{i=1}^{n} \mathbf{v}_i^\top \mathbf{w}_i \,. \tag{34}$$

In terms of the embedding-subnet connections, the input contains several concatenated embedding pairs $[[\mathbf{v}_1, \mathbf{v}_2], [\mathbf{v}_1, \mathbf{v}_3], \ldots, [\mathbf{v}_i, \mathbf{v}_j], \ldots, [\mathbf{v}_{n-1}, \mathbf{v}_n]]$, and each pair is passed through some micro network. For simplicity, we regard the micro networks as linear transformations, thus the

[5]We test several sub-net architectures and the structure in Fig. 5(d) is a relatively good choice.
[6]In this paper, we use micro network and sub-network interchangeably.

weight matrix can be represented by $[\mathbf{w}_{i,j}, \mathbf{w}'_{i,j}]$, where the input dimension is twice the embedding size, and the output dimension is determined by the following classifier.

$$\begin{aligned}
&\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}[\mathbf{v}_i, \mathbf{v}_j]^\top[\mathbf{w}_{i,j}, \mathbf{w}'_{i,j}] \\
=&\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbf{v}_i^\top\mathbf{w}_{i,j} + \mathbf{v}_j^\top\mathbf{w}'_{i,j} \\
=&\sum_{i=1}^{n-1}\mathbf{v}_i^\top\sum_{j=i+1}^{n}\mathbf{w}_{i,j} + \sum_{j=2}^{n}\mathbf{v}_j^\top\sum_{i=1}^{j-1}\mathbf{w}'_{i,j} \\
=&\sum_{i=1}^{n}\mathbf{v}_i^\top(\sum_{j<i}\mathbf{w}'_{j,i} + \sum_{j>i}\mathbf{w}_{i,j}) \,. \qquad (35)
\end{aligned}$$

From these two formulas, we find the embedding-DNN connections can be yielded from the embedding-subnet connections: $\mathbf{v}_i^\top(\sum_{j<i}\mathbf{w}'_{j,i} + \sum_{j>i}\mathbf{w}_{i,j}) \rightarrow \mathbf{v}_i^\top\mathbf{w}''_i$.

4 PRACTICAL ISSUES

This section discusses several practical issues, some of which are mentioned in Section 3.3 (e.g., initialization), the others are related to applications (e.g., data processing). Corresponding experiments are located at Section 5.2.

4.1 Data Processing

The data in user response prediction is usually categorical, and sometimes are numerical or multi-valued. When the model input contains both one-hot vectors and real values, this model is hard to train: (i) One-hot vectors are usually sparse and high-dimensional while real values are dense and low-dimensional, therefore, they have quite different distributions. (ii) Different from real values, one-hot vectors are not comparable in numbers. For these reasons, categorical and numerical data should be processed differently.

Numerical data is usually processed by bucketing/clustering: First, a numerical field is partitioned by a series of thresholds according to its histogram distribution. Then a real value is assigned to a bucket. Finally, the bucket identifier is used to replace the original value. For instance, age $< 18 \rightarrow$ Child, $18 \leq$ age $< 28 \rightarrow$ Youth, age $\geq 28 \rightarrow$ Adult. Another solution is trees [18]. Since decision trees split data examples into several groups, each group can be regarded as a category.

Different from numerical and categorical data, set data takes multiple values in one data instance, e.g., an item has multiple tags. The permutation invariant property of set data has been studied in [49]. In this paper, the set data embeddings are averaged before feeding to DNN.

4.2 Initialization

Initializing parameters with small random numbers is widely adopted, e.g., uniform or normal distribution with 0 mean and small variance. For a hidden layer in DNN, an empirical choice for standard deviation is $\sqrt{1/n_{in}}$, where n_{in} is the input size of that layer. Xavier initialization [12] takes both forward and backward passes into consideration: taking uniform distribution $[-v_{max}, v_{max}]$ as an example, the upper bound v_{max} should be $\sqrt{6/(n_{in} + n_{out})}$, where n_{in} and n_{out} are the input and output sizes of a hidden layer. This setting stabilizes activations and gradients of a hidden layer at the early stage of training.

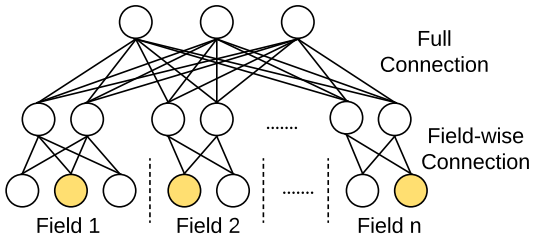

Fig. 6. An example of field-wise connection.

The above discussion is limited to (i) dense input, and (ii) fully connected layers. Fig. 6 shows an embedding layer followed by a fully connected layer. An embedding layer has sparse input and is field-wisely connected, i.e., each field is connected with only a part of the neurons. Since an embedding layer usually has extremely large input dimension, typical standard deviations include: $\sqrt{c/Nk}$, $\sqrt{c/nk}$, $\sqrt{c/k}$, and pre-training, where c is a constant, N is the input dimension, n is the number of fields, and k is the embedding size. Pre-training is used in [46, 51]. For convenience, we use random initialization in most experiments for end-to-end training. And we compare random initialization with pre-training in Section 5.2.

4.3 Optimization

In this section, we discuss potential risks of adaptive optimization algorithms in the scenario of sparse input. Compared with SGD, adaptive optimization algorithms converge much faster, e.g., AdaGrad [10], Adam [22], FTRL [30, 43], among which Adam is an empirically good choice [13, 47].

Even though adaptive algorithms speed up training and sometimes escape from local minima in practice, there is no theoretical guarantee of better performance. Take Adam as an example,

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{36}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{37}$$

$$g'_t = \frac{m_t/(1-\beta_1^t)}{\sqrt{v_t/(1-\beta_2^t)} + \epsilon}, \tag{38}$$

where m_t and v_t are estimations of the first and the second moments, g_t is the real gradient at training step t, g'_t is the estimated gradient at training step t, β_1 and β_2 are decay coefficients, and ϵ is a small constant for numerical stability. Empirical values for β_1, β_2 and ϵ are 0.9, 0.999, 10^{-8}, respectively.

Before our discussion, we should notice that the gradient of the logit $\hat{y}$

$$\nabla_{\hat{y}} \mathcal{L} = \sigma(\hat{y}) - y \tag{39}$$

is bounded in $(-1, 1)$. Fig. 7 shows $\nabla_{\hat{y}} \mathcal{L}$ of typical models with SGD/Adam. From this figure, we find the logit gradient decays exponentially at the early stage of training. Because of chain rule and backpropagation, the gradients of any parameters depend on $\nabla_{\hat{y}} \mathcal{L}$. This indicates the gradients of any parameters decrease dramatically at the early stage of training.

The following discussion uses Adam to analyze the behaviors of adaptive optimization algorithms on (unbalanced) sparse dataset, and the parameter sensitivity of Adam is studied in Section 5.2. Considering an input $\mathbf{v}_{\text{London}}$ appears for first time in training examples at time t, without loss of generality, we assume $g_t(\mathbf{v}_{\text{London}}) > 0$.

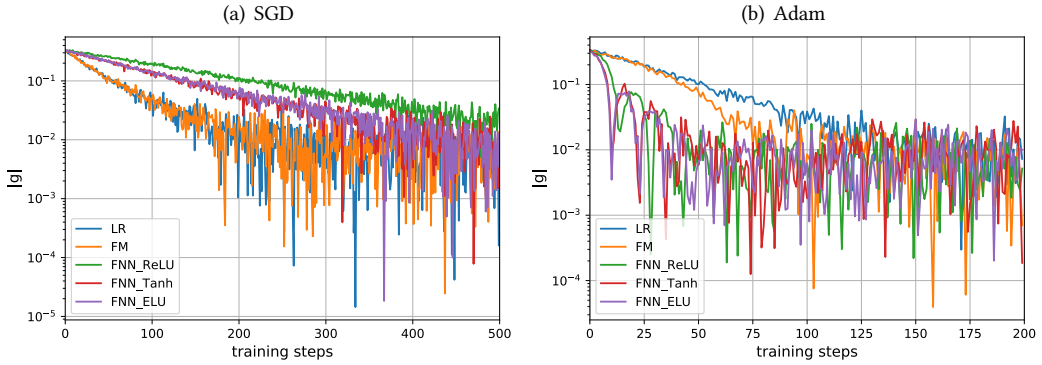

Fig. 7. The gradient magnitude of logit decays exponentially at the early stage of training. *Note:* FNN_ReLU, FNN_Tanh, FNN_ELU are FNNs with different activation functions. The x-axis means the number of mini-batches fed in a model, and the mini-batch size is 2000. The y-axis means the absolute gradient of the logit.

4.3.1 Sensitive Gradient. Firstly, we discuss the instant behavior of g'_t at time t,

$$g'_t(\mathbf{v}_{\text{London}}) = \frac{g_t(1-\beta_1)/(1-\beta_1^t)}{g_t\sqrt{(1-\beta_2)/(1-\beta_2^t)}+\epsilon} > 0\ . \tag{40}$$

The estimated gradient g'_t mainly depends on g_t, ϵ, and t, as shown in Fig. 8(a)-(c). At a certain training step t, the estimated gradient g'_t changes dramatically when g_t approaches some threshold. In another word, g'_t saturates across some of the value domain of g_t. Denoting $x = \log(g_t)$, $a = \frac{1-\beta_1}{1-\beta_1^t}$, $b = \sqrt{\frac{1-\beta_2}{1-\beta_2^t}}$, we have $f(x) = g'_t = \frac{ae^x}{be^x+\epsilon}$. Then we can find the threshold g_t^* where $\partial f(x)/\partial x$ is maximal

$$\frac{\partial f(x)}{\partial x} = \frac{a\epsilon}{b^2e^x + \epsilon^2e^{-x} + 2b\epsilon} \le \frac{a\epsilon}{4b\epsilon} \tag{41}$$

$$g_t^* = \frac{\epsilon}{b} = \frac{\epsilon}{\sqrt{(1-\beta_2)/(1-\beta_2^t)}}\ . \tag{42}$$

Fig. 8(d)-(f) show the thresholds g^* within 10^4 training steps. From these figures, we find g^* increases with ϵ and t. As training goes on, more and more gradients will cross the threshold and vanish. Thus, we conclude ϵ has a large impact on model convergence and training stability. And this impact will be amplified if the dataset is unbalanced. Suppose the positive ratio of a dataset is α = # positive/# total, then

$$\mathbb{E}[\nabla_{\hat{y}}\mathcal{L}] = \mathbb{E}[\sigma(\hat{y}) - y] = \alpha(\mathbb{E}[\sigma(\hat{y})]/\alpha - 1)\ . \tag{43}$$

Thus $\nabla_{\hat{y}}\mathcal{L}$ is proportional to α when $\mathbb{E}[\sigma(\hat{y})]/\alpha \to 1$.

4.3.2 Long-tailed Gradient. Secondly, we discuss the long-tailed behavior of g'_t in a time window T after time t. If a sparse input $\mathbf{v}_{\text{London}}$ appears only once at training step t, then $g_{\neq t}(\mathbf{v}_{\text{London}}) = 0$

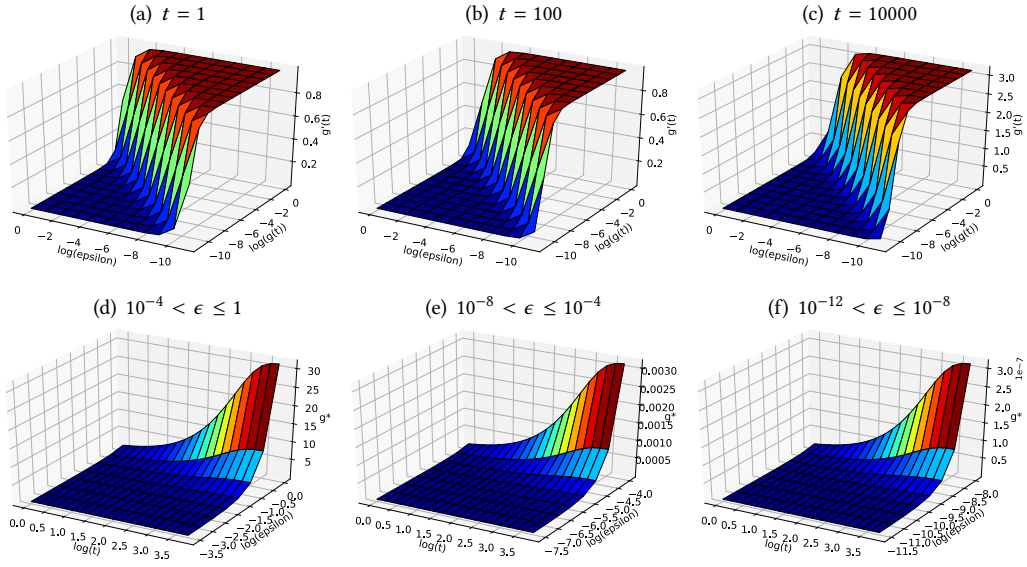

Fig. 8. Adam gradient estimation with respect to ϵ and training steps t. *Note:* In (a)-(c), the x- and y-axis represent logarithm of real gradient g_t and constant ϵ, the z-axis represents the estimated gradient of Adam g'_t. In (d)-(f), the x- and y-axis represent logarithm of constant ϵ and training step t, the z-axis represents thresholds g^* of real gradient g_t.

and $g'_{t+T}(\mathbf{v}_{\text{London}})$ decays in a time window T. Assume $g'_t(\mathbf{v}_{\text{London}}) > 0$,

$$g'_{t+T}(\mathbf{v}_{\text{London}}) = \frac{(1-\beta_1)\beta_1^T/(1-\beta_1^{t+T})}{\sqrt{(1-\beta_2)\beta_2^T/(1-\beta_2^{t+T})} + \epsilon/g_t} > 0\ . \tag{44}$$

Fig. 9 illustrates gradient decay and cumulative gradient in a window $T \leq 60$ at $t = 1, 100, 10000$, respectively. From 9(a)-(c) we can see, the gradient larger than a threshold (different from g^*) is scaled up to a "constant", and the gradient smaller than that threshold shrinks to a "constant". If g'_{t+T} is continuously updated to $\mathbf{v}_{\text{London}}$, the cumulative effect is shown in 9(d)-(f). The long-tailed effect may result in training instability or parameter divergence on sparse input. A solution is sparse update, i.e., the estimated moments m_t and v_t are updated normally, but the estimated gradient g'_t is only applied to parameters involved in the forward propagation.

4.4 Regularization

4.4.1 L2 Regularization. L2 regularization is usually used to control overfitting of latent vectors, yet it may cause severe computation overhead when the dataset is extremely sparse. Denoting $\mathbf{V} \in \mathbb{R}^{N\times k}$ as the embedding matrix, L2 regularization adds a term $\frac{1}{2}\|\mathbf{V}\|_2^2$ to the loss function. This term results in an extra gradient term $\nabla_{\mathbf{V}}\frac{1}{2}\|\mathbf{V}\|_2^2 = \mathbf{V}$. For sparse input, L2 regularization is very expensive because it will update all the parameters, usually $10^6 \sim 10^9$ of them, in $\mathbf{V}$.

An alternative to L2 regularization is sparse L2 regularization [24], i.e., we only penalize the parameters involved in the forward propagation rather than all of them. A simple implementation is to penalize $\mathbf{v} = \text{embed}(\mathbf{x})$ instead of $\mathbf{V}$. Since $\mathbf{x}$ is a binary input, $\mathbf{v}$ indicates the parameters involved in the forward propagation.

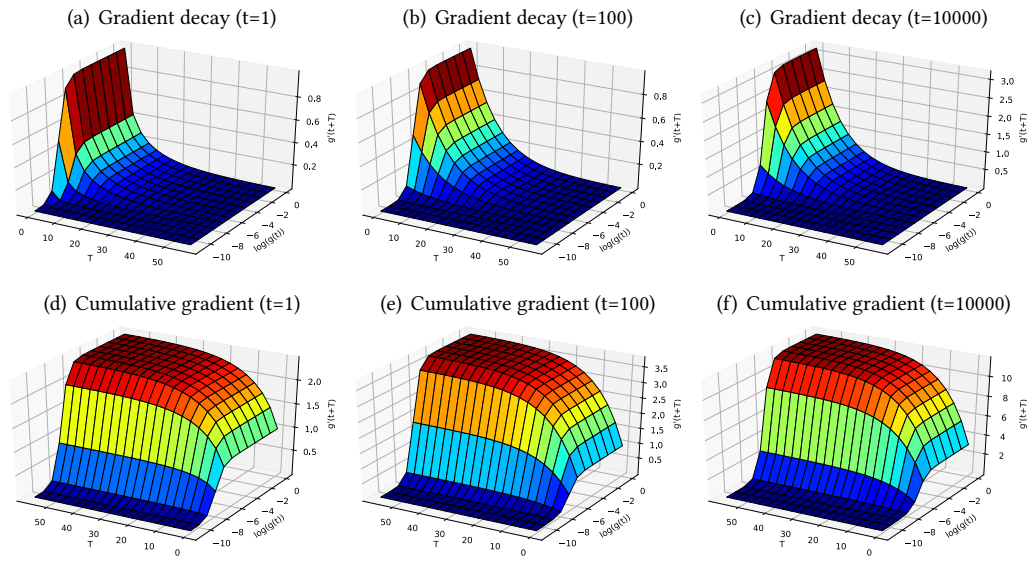

Fig. 9. Long-tailed effect of Adam gradient on sparse input. *Note:* In (a)-(c), the x- and y-axis represent logarithm of real gradient g_t and time window T, the z-axis represents the estimated gradient of Adam g'_{t+T} within a time window. In (d)-(f), the x- and y-axis are the same as (a)-(c), the z-axis represents the estimated gradient of Adam g'_{t+T} cumulated within time window T.

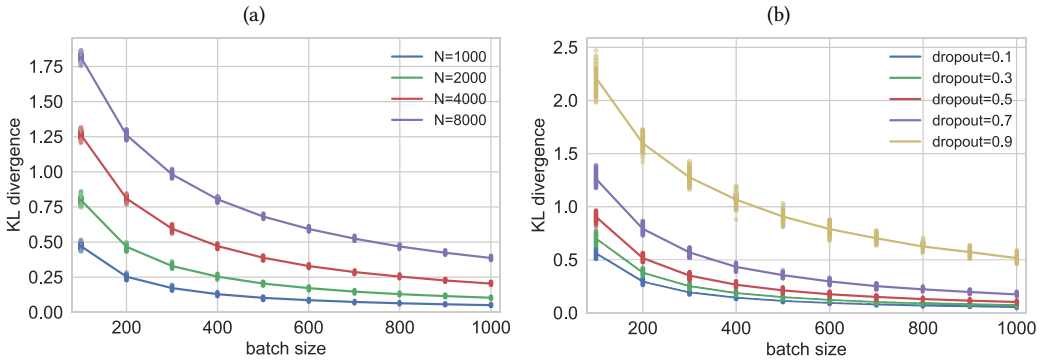

Fig. 10. A small mini-batch has a large bias in sparse data, and dropout amplifies this bias. *Note:* The x-axis means the mini-batch size, the y-axis means the KL-divergence of the estimated distribution from a mini-batch with respect to the ground truth distribution. In (a), N denotes the input dimension, and the data is sparser when N is larger. In (b), *dropout* denotes the probability of an input being dropped, and the input dimension is fixed to 1000.

4.4.2 Dropout. Dropout [41] is a technique developed for DNN, which introduces noise in training and improves robustness. However, in sparse data, a small mini-batch tends to have a large bias, and dropout may amplify this bias. We conduct a simple experiment to show this problem, and the results are shown in Fig. 10.

We generate a categorical dataset from a distribution $Q(X)$, $X \in \{x_1, x_2, \ldots, x_N\}$, where every sample has 10 values without replacement. For a mini-batch size bs, we draw bs samples as a batch and use this batch to estimate the distribution, $P(X = x_i) = freq(x_i)$. The evaluation metric is KL divergence, and we use a constant 10^{-8} for numerical stability. For each $N \in \{1000, 2000, 4000, 8000\}$, and $bs \in \{100, 200, \ldots, 1000\}$, we sample 100 batches and the results are shown in Fig. 10(a), where the mean values are calculated and plotted in lines. From this figure, we conclude: (i) a mini-batch tends to have a large bias on sparse data; (ii) this bias increases with data sparsity and decreases with batch size.

Then we test dropout on $N = 1000$. Denoting dropout rate as $drop$, we randomly generate a mask of length N with $drop$ elements being 0 while others being $1/drop$ to simulate the dropout process. Every sample is weighted by a mask before estimating the distribution. The results are shown in Fig. 10(b). This simple experiment illustrates the noise sensitivity of a mini-batch in sparse data. Thus, we turn to normalization techniques when a dataset is extremely sparse.

4.4.3 DNN Normalization. Normalization is carefully studied recently to solve internal covariate shift [20] in DNNs, and this method stabilizes activation distributions. Typical methods include batch normalization (BN) [20], weight normalization (WN) [38], layer normalization (LN) [3], self-normalizing network (SNN) [23], etc. In general, BN, WN, and LN use some statistics to normalize hidden layer activations, while SNN uses an activation function SELU, making activations naturally converge to the standard normal distribution.

Denote $\mathbf{x}$ as the input to a hidden layer with weight matrix $\mathbf{w}$, $\mathbf{x}_i$ as the i-th instance of a mini-batch, and x_i^j as the value of the j-th dimension of $\mathbf{x}_i$. As the following, we discuss BN, WN, LN, and SELU in detail.

Batch Normalization. BN normalizes activations $\mathbf{w}^\top \mathbf{x}$ using statistics within a mini-batch

$$\mathrm{BN}(\mathbf{w}^\top \mathbf{x}) = \frac{\mathbf{w}^\top \mathbf{x} - \mathrm{avg}_i(\mathbf{w}^\top \mathbf{x})}{\mathrm{std}_i(\mathbf{w}^\top \mathbf{x})} \mathbf{g} + \mathbf{b} \,, \tag{45}$$

where $\mathbf{g}$ and $\mathbf{b}$ scale and shift the normalized values. These parameters are learned along with other parameters and restore the representation power of the network. Similar parameters are also used in WN and LN.

BN solves internal covariate shift [20] and accelerates DNN training. However, BN may fail when the input is sparse, because BN relies on the statistics of a mini-batch. As shown in Fig. 10(a), a mini-batch tends to have a large bias when input data is sparse, but large mini-batch is not always practical due to the computation resource limit such as GPU video memory.

Weight Normalization. WN re-parametrizes the weight matrix $\mathbf{w}$, and learns the direction and scale of $\mathbf{w}$ separately

$$\mathrm{WN}(\mathbf{w}^\top \mathbf{x}) = \left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \mathbf{g} \right)^\top \mathbf{x} \,. \tag{46}$$

WN does not depend on mini-batch, thus can be applied to noise-sensitive models [38]. However, WN is roughly infeasible on high-dimensional data, because WN depends on the L2 norm of parameters, which results in even higher complexity than L2 regularization. Thus, WN meets similar complexity problem as L2 regularization when input space is extremely sparse.

Layer Normalization. LN normalizes activations $\mathbf{w}^\top \mathbf{x}$ using statistics of different neurons within the same layer

$$\mathrm{LN}(\mathbf{w}^\top \mathbf{x}) = \frac{\mathbf{w}^\top \mathbf{x} - \mathrm{avg}_j(\mathbf{w}^\top \mathbf{x})}{\mathrm{std}_j(\mathbf{w}^\top \mathbf{x})} \mathbf{g} + \mathbf{b} \,. \tag{47}$$

LN stabilizes the hidden state dynamics in recurrent networks [3]. In our experiments, we apply LN on fully connected layers and inner/kernel product layers, and we apply fused LN in micro networks. Since LN does not work well in CNN [3], we exclude LN in CCPM.

Self-normalizing Network. SNN uses SELU as the activation function,

$$\text{SELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}. \tag{48}$$

Based on Banach fixed-point theorem, the activations that are propagated through many SELU layers will converge to zero mean and unit variance. Besides, SELU declares significant improvement in feed-forward neural networks on a large variety of tasks [23].

To summarize, we use (sparse) L2 regularization to penalize embedding vectors, and we use dropout, LN, and SELU to regularize DNNs. BN is not applied because of the mini-batch problem discussed above, and WN is not applied because of its high complexity. Corresponding experiments are in Section 5.2.

5 EXPERIMENTS

In Section 5.1, we present overall comparison. In Section 5.2, we discuss practical issues: complexity, initialization (Section 4.2), optimization (Section 4.3), and regularization (Section 4.4). In Section 5.3, we propose a visualization method to analyze feature interactions, corresponding to Section 3.1. And finally, in Section 5.4, we conduct a synthetic experiment to illustrate the deficiency of DNN, corresponding to Section 3.2.

5.1 Offline and Online Evaluations

In this section, we conduct offline and online evaluations to give a thorough comparison: (i) We compare KFM/NIFM with other latent vector-based models to verify the effectiveness of kernel product methods. (ii) We compare PNNs with other DNN-based models to verify the effectiveness of product layers. (iii) We also participate in the Criteo challenge and compete KFM with libFFM directly. (iv) We deploy PIN in a real recommender system.

5.1.1 *Datasets.*

Criteo. Criteo[7] contains one month of click logs with billions of data examples. A small subset of Criteo was published in Criteo Display Advertising Challenge, 2013, and FFM was the winning solution [21]. We select "day6-12" for training, and "day13" for evaluation. Because of the enormous data volume and serious label unbalance (only 3% samples are positive), we apply negative down-sampling to keep the positive ratio close to 50%. We convert the 13 numerical fields into categorical through bucketing (in Section 4.1). And we set the categories appearing less than 20 times as a dummy category "other".

Avazu. Avazu[8] was published in Avazu Click-Through Rate Prediction contest, 2014, and FFM was the winning solution [21]. We randomly split the public dataset into training and test sets at 4:1, and remove categories appearing less than 20 times to reduce dimensionality.

iPinYou. iPinYou[9] was published in iPinYou RTB Bidding Algorithm Competition, 2013. We only use the click data from season 2 and 3 because of the same data schema. We follow the data processing of [52], and we remove "user tags" to prevent leakage.

[7] http://labs.criteo.com/downloads/download-terabyte-click-logs/
[8] http://www.kaggle.com/c/avazu-ctr-prediction
[9] http://data.computational-advertising.org

Table 4. Dataset statistics.

Dataset	# instances	# categories	# fields	pos ratio
Criteo	1×10^8	1×10^6	39	0.5
Avazu	4×10^7	6×10^5	24	0.17
iPinYou	2×10^7	9×10^5	16	0.0007
Huawei	5.7×10^7	9×10^4	9	0.008

Huawei. Huawei [15] is collected from the game center of Huawei App Store in 2016, containing app, user, and context features. We use the same training and test sets as [15], and we use the same hyper-parameter settings to reproduce their results.

Table 4 shows statistics of the 4 datasets[10].

5.1.2 Compared Models. We compare 8 baseline models, including LR [25], GBDT [6], FM [36], FFM [21], FNN [51], CCPM [27], AFM [46] and DeepFM [15], all of which are discussed in Section 2 and Section 3. We use XGBoost[11] and libFFM[12] as GBDT and FFM in our experiments. We implement[13] all the other models with Tensorflow[14] and MXNet[15]. We also implement FFM with Tensorflow and MXNet to compare its training speed with other models on GPU. In particular, our FFM implementation (Avazu log loss=0.37805) has almost the same performance as libFFM (Avazu log loss=0.37803).

5.1.3 Evaluation Metrics. The evaluation metrics are **AUC**, and **log loss**. AUC is a widely used metric for binary classification because it is insensitive to the classification threshold and the positive ratio. If prediction scores of all the positive samples are higher than those of the negative, the model will achieve AUC=1 (separate positive/negative samples perfectly). The upper bound of AUC is 1, and the larger the better. Log loss is another widely used metric in binary classification, measuring the distance between two distributions. The lower bound of log loss is 0, indicating the two distributions perfectly match, and a smaller value indicates better performance.

5.1.4 Parameter Setting. Table 5 shows key hyper-parameters of the models.

For a fair comparison, on Criteo, Avazu, and iPinYou, we (i) fix the embedding size according to the best-performed FM (searched among {10, 20, 40, 80}), and (ii) fix the DNN structure according to the best-performed FNN (width searched in [100, 1000], depth searched in [1, 9]). In terms of initialization, we initialize DNN hidden layers with xavier [12], and we initialize the embedding vectors from uniform distributions (range selected from $\{\sqrt{c/Nk}, \sqrt{c/nk}, \sqrt{c/k}\}$, $c = \{1, 3, 6\}$, as discussed in Section 4.2.). For Huawei, we follow the parameter settings of [15].

With these constraints, all latent vector-based models have the same embedding size, and all DNN-based models additionally have the same DNN classifier. Therefore, all these models have similar amounts of parameters and are evaluated with the same training efforts. We also conduct parameter study on 4 typical models, where grid search is performed.

5.1.5 Overall Performance. Table 6 shows the overall performance. Underlined numbers are best results of baseline models, and bold numbers are best results of all.

[10]Datasets: https://github.com/Atomu2014/Ads-RecSys-Datasets and https://github.com/Atomu2014/make-ipinyou-data.
[11]https://xgboost.readthedocs.io/en/latest/
[12]https://github.com/guestwalk/libffm
[13]Code: https://github.com/Atomu2014/product-nets-distributed
[14]https://www.tensorflow.org/
[15]https://mxnet.apache.org/

Table 5. Parameter settings.

Param	Criteo	Avazu	iPinYou	Huawei
General	bs=2000, opt=Adam, lr=10^{-3}	bs=2000, opt=Adam, lr=10^{-3}	bs=2000, opt=Adam, lr=10^{-3}, l2=10^{-6}	bs=1000, opt=Adam, lr=10^{-4}, l2=10^{-4}
LR	-	-	-	-
GBDT	depth=25, # tree=1300	depth=18, # tree=1000	depth=21, # tree=700	depth=6, # tree=600
FFM	k=4	k=4	k=4	k=4
FM, KFM	k=20,	k=40,	k=20,	k=10,
AFM	t=0.01, h=32, l2_a=0.1	t=1, h=256 l2_a=0	t=1, h=256, l2_a=0.1	t=0.0001, h=64, l2_a=0
NIFM	sub-net=[40,1]	sub-net=[80,1]	sub-net=[40,1]	sub-net=[20,1]
CCPM	k=20, kernel=7×256, net=[256×3,1]	k=40, kernel=7×128, net=[128×3,1]	k=20, kernel=7×128, net=[128×3,1]	k=10, kernel=5×512, net=[512×3,1], drop=0.1
FNN, DeepFM, IPNN, KPNN	k=20, LN=T, net=[700×5,1]	k=40, LN=T, net=[500×5,1]	k=20, LN=T, net=[300×3,1]	k=10, LN=F, net=[400×3,1], drop=0.1
PIN	sub-net=[40,5]	sub-net=[40,5]	sub-net=[40,5]	sub-net=[20,1]

Note: bs=Batch Size, opt=Optimizer, lr=Learning Rate, l2=L2 Regularization on Embedding Layer, k=Embedding Size, kernel=Convolution Kernel Size, net=DNN Structure, sub-net=Micro Network, t=Softmax Temperature, l2_a= L2 Regularization on Attention Network, h=Attention Network Hidden Size, drop=Dropout Rate, LN=Layer Normalization (T: True, F: False)

Table 6. Overall performance. (Left-Right: Criteo, Avazu, iPinYou, Huawei)

Model	AUC (%)	Log Loss	AUC (%)	Log Loss	AUC (%)	Log Loss	AUC (%)	Log Loss
LR	78.00	0.5631	76.76	0.3868	76.38	0.005691	86.40	0.02648
GBDT	78.62	0.5560	77.53	0.3824	76.90	0.005578	86.45	0.02656
FM	79.09	0.5500	77.93	0.3805	77.17	0.005595	86.78	0.02633
FFM	79.80	0.5438	78.31	0.3781	76.18	0.005695	87.04	0.02626
CCPM	79.55	0.5469	78.12	0.3800	77.53	0.005640	86.92	0.02633
FNN	79.87	0.5428	78.30	0.3778	77.82	0.005573	86.83	0.02629
AFM	79.13	0.5517	78.06	0.3794	77.71	0.005562	86.89	0.02649
DeepFM	79.91	0.5423	78.36	0.3777	77.92	0.005588	87.15	0.02618
KFM	79.85	0.5427	78.40	0.3775	76.90	0.005630	87.00	0.02624
NIFM	79.80	0.5437	78.13	0.3788	77.07	0.005607	87.16	0.02620
IPNN	80.13	0.5399	78.68	0.3757	78.17	0.005549	87.27	0.02617
KPNN	80.17	0.5394	78.71	0.3756	78.21	0.005563	87.28	0.02617
PIN	**80.21**	**0.5390**	**78.72**	**0.3755**	**78.22**	**0.005547**	**87.30**	**0.02614**

Comparing KFM/NIFM with FM, FFM, and AFM, FFM is the best baseline model on Criteo, Avazu, and Huawei, and AFM is the best baseline model on iPinYou. KFM and NIFM achieve even better performance than FFM and AFM on all datasets. The scores of KFM and NIFM successfully verify the effectiveness of kernel product methods. A more detailed discussion about feature interactions is in Section 5.3. Comparing GBDT with FNN, we find GBDT performs no better than FNN. A possible reason is the enormous feature space makes GBDT hard to explore all possible combinations. Comparing PNNs with other DNN-based models, in general, DeepFM is the best baseline model on 4 datasets. PNNs consistently outperform DeepFM, and PIN achieves the best results on all datasets. The performance of PNNs verifies the effectiveness of product layers.

Table 7. Adaptive embedding.

Model	Embed Size	AUC	Log Loss
KFM	40	0.7840	0.3775
	$\min(4\log(N_i), 40)$	0.7850	0.3769
NIFM	40	0.7813	0.3788
	$\min(4\log(N_i), 40)$	0.7819	0.3786

Since kernel product has the ability to learn adaptive embeddings, we study adaptive embeddings on KFM and NIFM, as suggested in Section 3.1. We use $k_i = \min(c\log(N_i), K)$ as the embedding size for field i with N_i categories. We choose Avazu to compare adaptive embeddings and fixed size embeddings, because Avazu is relatively small and has balanced positive/negative samples. As shown in Table 7, adaptive embeddings further improve the performance of KFM and NIFM. Note that adaptive embeddings are harder to parallelize, thus are trained much slower.

Table 8. Significance test.

p-value	KFM, NIFM	IPNN, KPNN, PIN
FM, FFM	$< 10^{-6}$	$< 10^{-6}$
FNN, CCPM, DeepFM	$< 10^{-6}$	$< 10^{-6}$

Table 8 presents the significance test. We use the Wilcoxon signed-rank test to check if the results of our proposed models and baseline models are generated from the same distribution. The p-values validate that the improvements of our models are significant. To test model robustness, we train PIN on Criteo for 10 times with different random seeds. The standard deviation of AUC is 0.0085 (average AUC 80.2), and the standard deviation of log loss is 0.0002 (average log loss 0.539).

Besides, we participate in the Criteo Display Advertising Challenge, libFFM being the winning solution[16]. We download the winners' code and data to repeat their results and generate the training files. libFFM achieves log loss = 0.44506/0.44520 on the private/public leaderboard, and achieves 0.44493/0.44508 after calibration. We train KFM with the same training files as libFFM on one 1080Ti. KFM achieves 0.44484/0.44492 on the private/public leaderboard, and achieves 0.44484/0.44491 after calibration[17]. Besides, KFM uses less memory (64M parameters) than libFFM (272M parameters), and uses similar training time (3.5h)[18] to libFFM (3.5h).

[16]https://github.com/guestwalk/kaggle-2014-criteo. This repository has 2 branches, and we use the "master" branch.
[17]Our solution: https://github.com/Atomu2014/product-nets-distributed
[18]An acceleration trick is used, see Section 5.2.1.

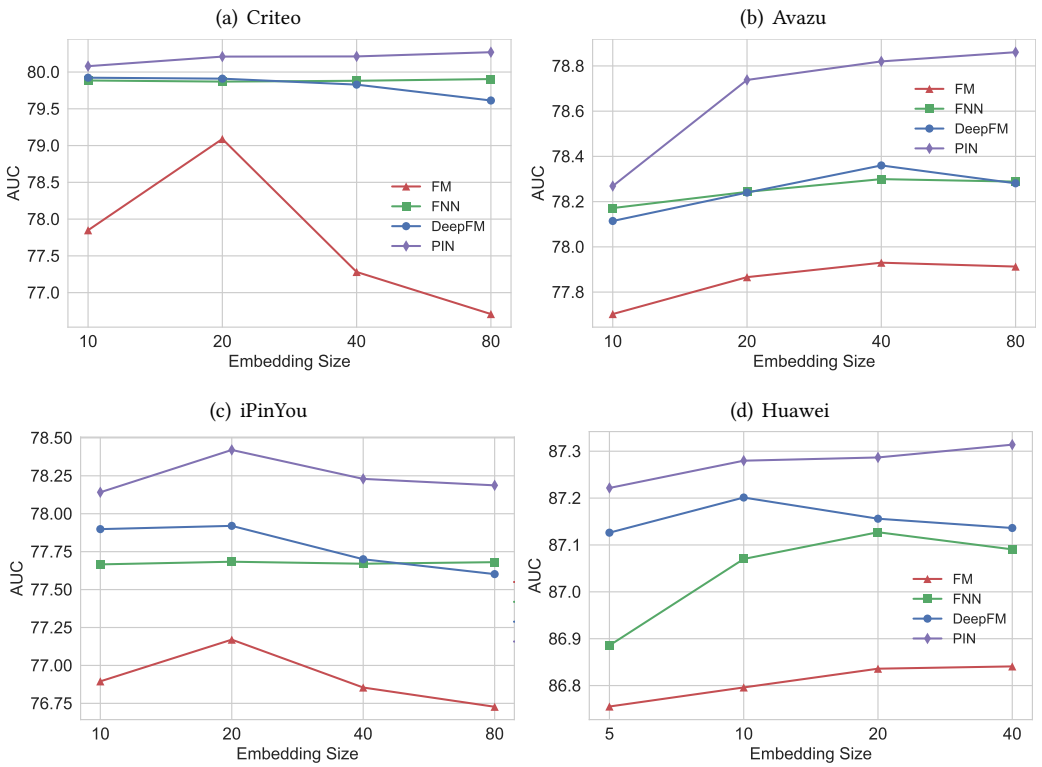

Fig. 11. Embedding size parameter study. *Note:* The network structures of DNN models are fixed according to Table 5.

5.1.6 Parameter Study. In this section, we study embedding size, network width, and network depth on FM, FNN, DeepFM, and PIN.

We test embedding size = {10, 20, 40, 80} on Criteo, Avazu, and iPinYou. From Fig. 11, we find 20 (Criteo), 40 (Avazu), 20 (iPinYou) are the best for FM, which is consistent with Table 5. Since Huawei is relatively low-dimensional, we test embedding size = {5, 10, 20, 40}, based on the parameter study of [15]. An interesting phenomenon is FM and DeepFM are easier to overfit with large embedding sizes. A possible reason is DNN has higher capacity than FM.

As for network structure, we first fix the embedding size and the depth according to Fig. 11 and Table 5, and test network width = {100, 200, 400, 800, 1600} on Criteo, Avazu, and iPinYou, {100, 200, 400, 800} on Huawei. The results are shown in Fig. 12(a)-(d). As for network depth, we choose the embedding size and the width according to the above results, and test network depth = {1, 3, 5, 7} on all the datasets. The results are shown in Fig. 12(e)-(h). In general, we find: (i) PIN consistently outperforms FNN and DeepFM. (ii) When the network is small, the DNN capacity is restricted, yet PIN performs even better than FNN and DeepFM. This means PIN learns more expressive feature representations. (iii) When the network is large, FNN and DeepFM are easier to overfit, which means PIN is more robust.

Finally, the best scores and parameters are shown in Table 9.

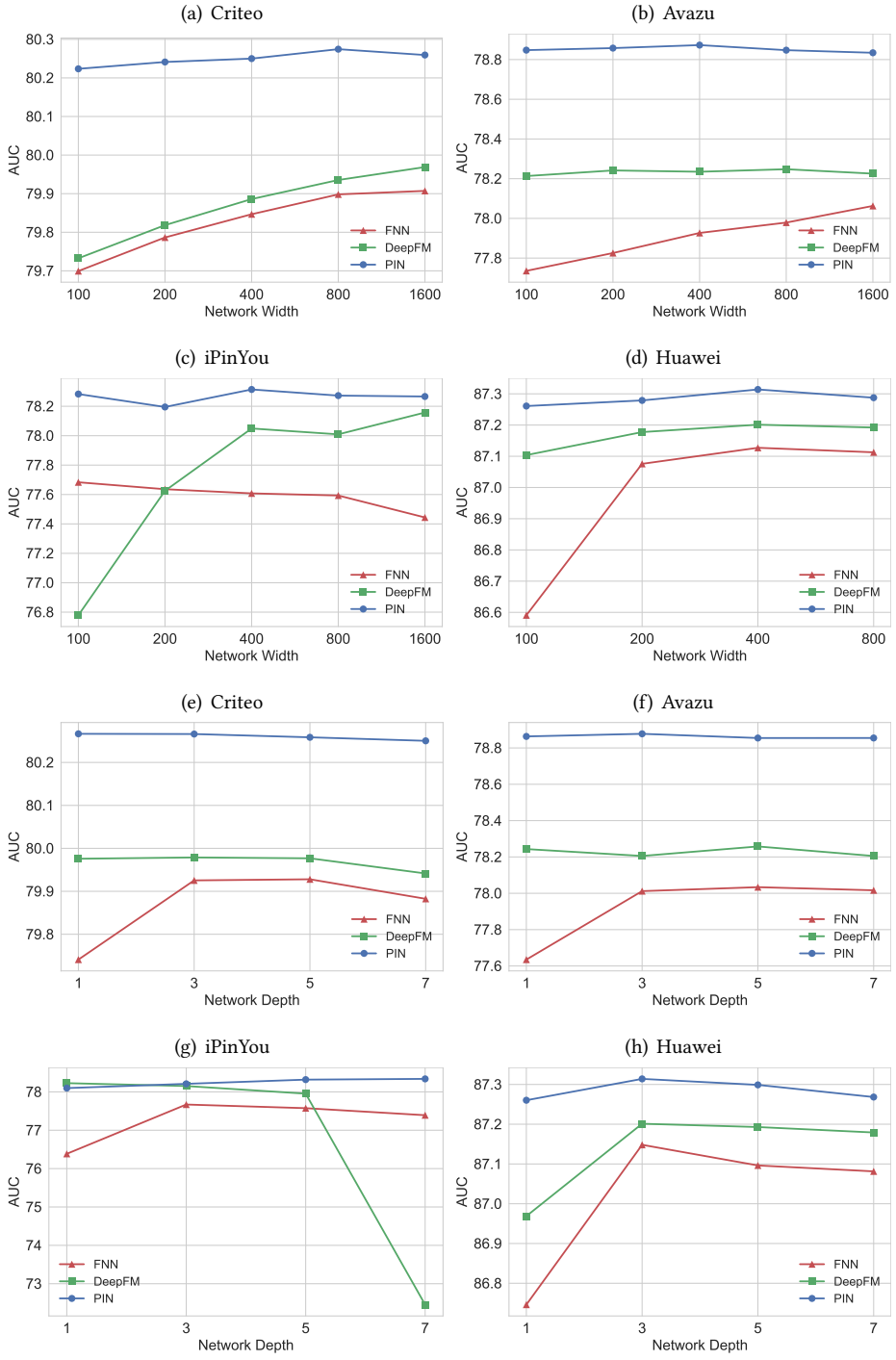

Fig. 12. Network structure parameter study. *Note:* In (a)-(d), the network depths are fixed according to Table 5, and the embedding sizes are chosen based on the best performance of Fig. 11. In (e)-(h), the embedding sizes and network widths are chosen according to Fig. 11 and (a)-(d).

Table 9. Best performance in parameter study. *Note:* k=Embedding Size, net=DNN structure.

	Criteo	Avazu	iPinYou	Huawei
FM	k=20 AUC=79.09 log loss=0.5500	k=40 AUC=77.93 log loss=0.3805	k=20 AUC=77.17 log loss=0.005595	k=40 AUC=86.84 log loss=0.02628
FNN	k=20, net=1600*5 AUC=79.91 log loss=0.5425	k=40, net=500*5 AUC=78.30 log loss=0.3778	k=20, net=300*3 AUC=77.68 log loss=0.005585	k=20 net=400*3 AUC=87.15 log loss=0.02619
DeepFM	k=10, net=1600*3 AUC=79.98 log loss= 0.5419	k=40, net=500*5 AUC=78.36 log loss=0.3777	k=20, net=1600*1 AUC=78.23 log loss=0.005562	k=10, net=400*3 AUC=87.20 log loss=0.02616
PIN	k=80, net=800*5 AUC=**80.27** log loss=**0.5385**	k=80, net=400*5 AUC=**78.88** log loss=**0.3745**	k=20, net=300*3 AUC=**78.42** log loss=**0.005546**	k=40, net=400*3 AUC=**87.31** log loss=**0.02616**

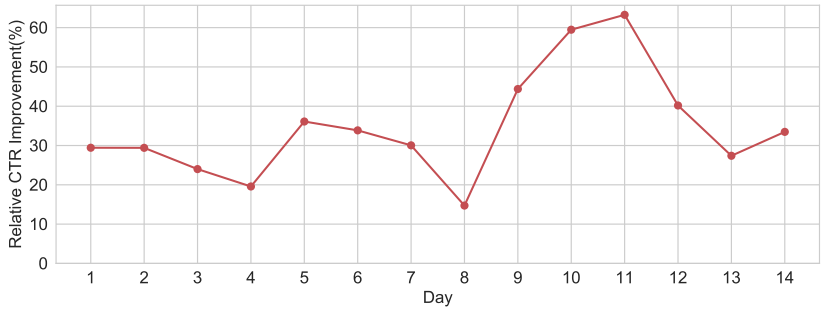

Fig. 13. Relative CTR improvements of PIN, calculated by $\frac{CTR of PIN - CTR of FTRL}{CTR of FTRL}$.

5.1.7 Online Evaluation. Beside offline evaluations, we perform an online A/B test in the game center of Huawei App Market. The compared model is FTRL which has been incrementally updated for several years. We train PIN on the latest 40-day CTR logs. Both models share the same feature map. After a 14-day A/B test, we observe an average of 34.67% relative CTR improvement (maximum 63.26%, minimum 14.72%). The improvements are shown in Fig. 13.

5.2 Practical Issues

5.2.1 Space and Time Complexity. In order to facilitate calculation, we choose Criteo (input dimension 1×10^8) to compare memory usage and training speed. Except for LR and GBDT, all the other models have embedding layers. To compare memory usage, we set the embedding size $k = 1$ for FFM and $k = 20$ for the other models, and we keep other parameters same as Table 5. To compare the training speed, we train these models with 10 million instances (batch size=2000) on an NVidia 1080Ti GPU. The results are shown in Table 10.

As for space complexity, LR has the least parameters, since it has no embedding layer. FFM consumes more memory, because the space complexity of FFM is $O(Nnk)$, where $N = 10^6$ is the input dimension, $n = 39$ is the number of fields, and $k = 1$ is the embedding size. Except for LR and FFM, the space complexities of other models are near $O(Nk)$. Among DNN-based models, CCPM

Table 10. Parameter number and training speed. *Note:* "# params" includes all trainable weights.

Model	# params (10^6)	time (min)	Model	# params (10^6)	time (min)
LR	1	1	FNN	22.51	2
FM	21	3	CCPM	20.23	2
AFM	21	13	DeepFM	23.51	3
FFM	≥ 40	5	IPNN	23	3
KFM	21.3	12	KPNN	23.3	13
NIFM	22.22	6	PIN	26.48	6

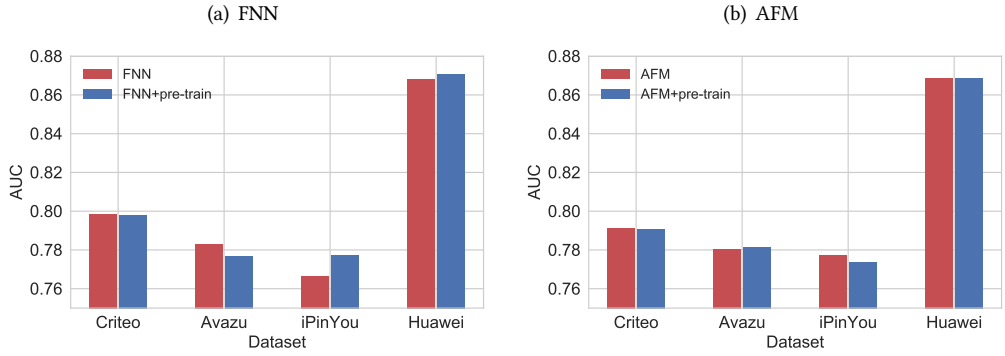

Fig. 14. Comparison of FM pre-training and random initialization.

uses the least memory due to parameter sharing in convolutional layers. PNNs use extra feature extractors, thus require more memory than FNN.

In terms of training speed, we find AFM, KFM, and KPNN are relatively slow. The interactions in FM have a complexity of $O(nk)$, the attention network in AFM has a complexity of $O(n^2kh)$, and the kernel products in KFM and KPNN have a complexity of $O(n^2k^2)$. The complexity of kernel products is quadratic to the embedding size, which slows down training if we use large embedding vectors. In the Criteo Challenge, we use a trick which can reduce the complexity of kernel products to $O(n^2k)$: we replace the kernel matrices of size $k \times k$ with kernel vectors of size k. This trick saves training time dramatically in the contest.

5.2.2 FM Pre-training. Section 4.2 lists several initialization methods for the embedding vectors. In [46, 51], the authors pre-trained the embedding vectors with FM. We compare pre-training with random initialization on FNN and AFM, the results are in Fig. 14. From this figure, we find FM pre-training does not always produce better results compared with random initialization. Thus we believe the initialization of embedding vectors depends on datasets, which we leave as future work.

5.2.3 Optimization. The data in user response prediction is usually sparse and unbalanced (positive samples are usually much fewer than negative ones). A sparse dataset is hard to train because the input categories usually follow long-tailed distributions. An unbalanced dataset is also hard to train because it may produce extremely small gradients when updating the model parameters. Section 4.3 discusses the behaviors of Adam on sparse input. In this section, we mainly focus on gradient sensitivity in Section 4.3.1.

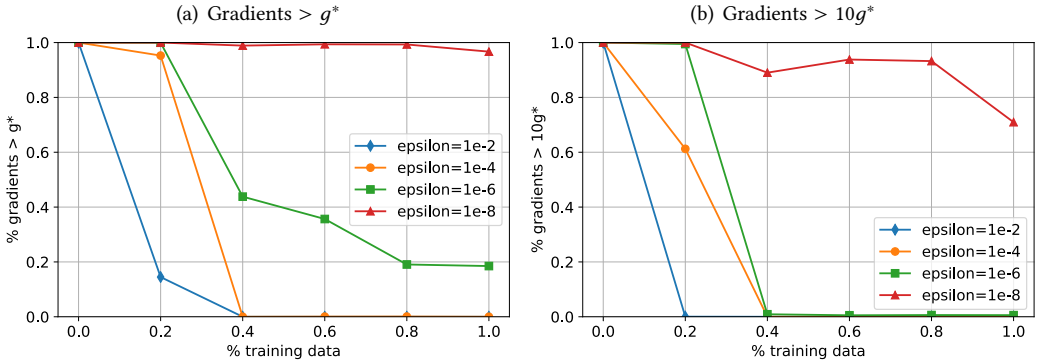

Fig. 15. Gradient decays with respect to ϵ of Adam. *Note:* The x-axis means the portion of training examples fed in a model. The y-axis means the portion of gradients with magnitude larger than a threshold. g^* refers to Eq. (42), which influences the model convergence.

Table 11. Training Adam on iPinYou dataset with different ϵ. *Note:* "-" means model does not converge.

AUC	ϵ=10^{-2}	ϵ=10^{-4}	ϵ=10^{-6}	ϵ=10^{-8}	Log Loss	ϵ=10^{-2}	ϵ=10^{-4}	ϵ=10^{-6}	ϵ=10^{-8}
LR	-	76.38	75.31	74.84	LR	-	0.005691	0.005705	0.005719
FM	-	75.28	77.17	75.69	FM	-	0.005674	0.005595	0.005655
FFM	-	76.18	75.24	74.29	FFM	-	0.005695	0.005697	0.005709
CCPM	-	77.37	77.53	76.83	CCPM	-	0.005584	0.005640	0.005622
FNN	76.55	77.82	77.26	76.66	FNN	0.005597	0.005573	0.005568	0.005620
AFM	-	75.30	77.71	75.72	AFM	-	0.005658	0.005562	0.005654
DeepFM	74.96	77.92	77.36	76.46	DeepFM	0.006189	0.005588	0.005581	0.005603
KFM	62.62	72.77	76.90	75.13	KFM	0.005997	0.005776	0.005630	0.005675
NIFM	59.57	75.25	77.07	76.19	NIFM	0.005994	0.005664	0.005607	0.005622
IPNN	75.80	78.17	77.12	75.90	IPNN	0.005703	0.005549	0.005577	0.005608
KPNN	76.10	78.21	77.43	76.91	KPNN	0.005647	0.005563	0.005582	0.005611
PIN	76.71	**78.22**	77.39	76.81	PIN	0.005602	**0.005547**	0.005587	0.005600

Fig. 15 shows the gradient changes of FNN on iPinYou. We collect gradients (absolute values, by default) of the first hidden layer in FNN at different training steps and calculate the ratio of the gradients greater than g^* or $10g^*$. From this figure, the gradients are always greater than the threshold when $\epsilon = 10^{-8}$, but cross the threshold at different training steps when $\epsilon = 10^{-2}, 10^{-4}, 10^{-6}$. Thus we guess ϵ has a large impact on model convergence.

We then test the parameter sensitivity of ϵ on iPinYou for this dataset is sparse and unbalanced. The results are shown in Table 11. From this table, we conclude: (i) Shallow models are more sensitive to ϵ, and in some cases, these models do not converge after sufficient training steps. (ii) An unbalanced dataset is sensitive to ϵ, and sometimes empirical value $\epsilon = 10^{-8}$ is not a good choice. For these reasons, when presenting the overall performance in Table 6, we use $\epsilon = 10^{-8}$ on Criteo and Avazu, $\epsilon = 10^{-5}$ on Huawei, and we use different ϵ (usually 10^{-4} or 10^{-6}) on different models on iPinYou.

5.2.4 Regularization. In Section 4.4, we discuss several regularization methods. In this section, we study L2 regularization, dropout, LN, and SELU. We choose FNN to perform the parameter study, since FNN is the most typical network structure.

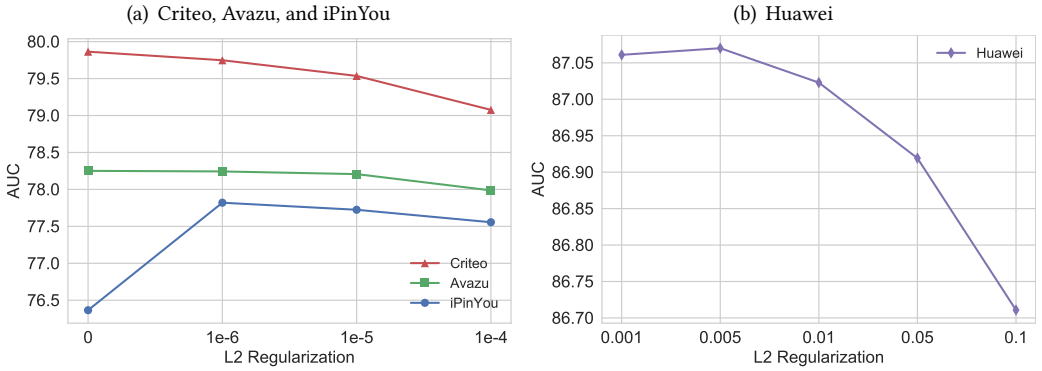

Fig. 16. FNN performance with respect to L2 regularization.

Fig. 16 shows the L2 regularization results. In Huawei, L2 regularization controls overfitting well, and keeps the embedding distribution stable during training. In Criteo, Avazu, and iPinYou, we use sparse L2 regularization and LN instead, because the input dimension is too large to apply L2 regularization. According to Fig. 16, we use $L2 = 0, 0, 10^{-6}$, and 0.005 on Criteo, Avazu, iPinYou and Huawei respectively.

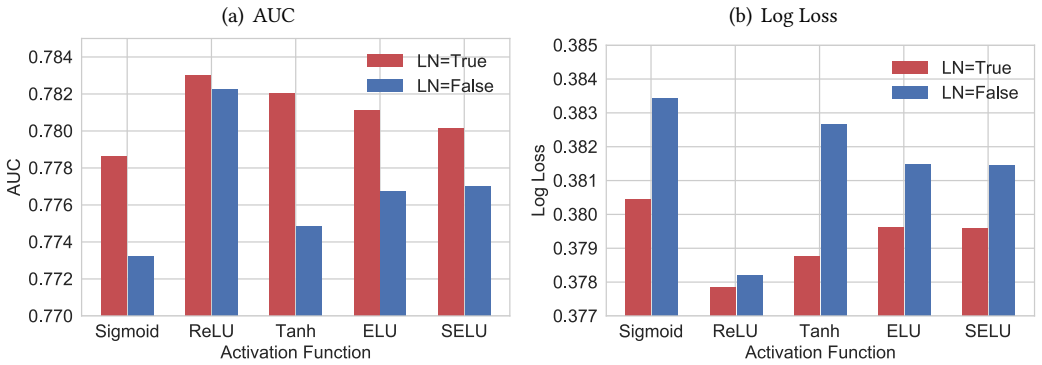

Fig. 17. FNN performance on Avazu with respect to different activation functions. *Note:* LN means layer normalization.

We compare SELU with other activation functions on FNN, Avazu. The results are shown in Fig. 17. We can observe that ReLU has the best performance, while SELU is similar to ELU. This may be because ReLU has more efficient gradient propagation.

We find the performance of dropout depends on data sparsity. For sparse data, small mini-batch has a large bias, and dropout amplifies this bias, as shown in Fig. 10. We study dropout on FNN, Avazu. From Fig. 18 we find: (i) Dropout decreases AUC. This becomes even worse when batch size is small. (ii) LN stabilizes dropout with different batch sizes. According to this result and the parameter study in [15], we use LN on Criteo, Avazu, and iPinYou, and we use dropout value 0.1 (without LN) on Huawei.

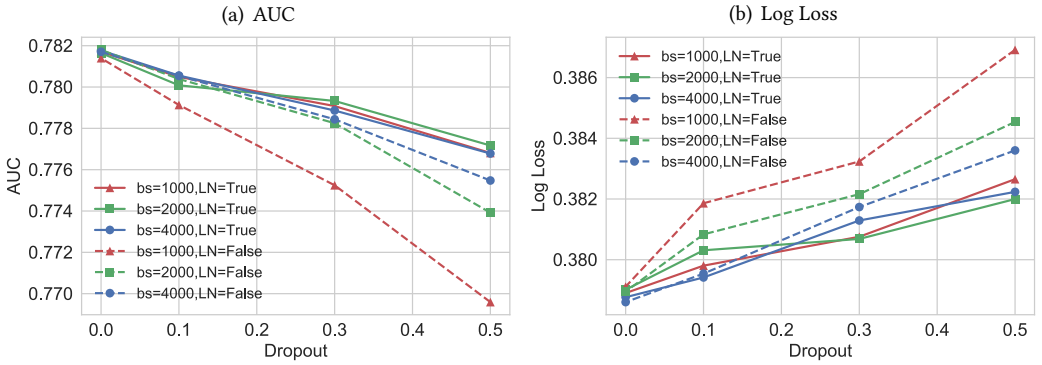

Fig. 18. FNN performance on Avazu with respect to dropout and layer normalization. *Note:* dropout rate means the probability of a neuron being disabled when training a mini-batch.

5.3 Feature Interaction Visualization

In Section 3.1, we analyze the weaknesses of FM and FFM, and refine feature interactions to field-aware feature interactions. In this section, we propose a technique to visualize the learned feature interactions. Recall the cross term $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ in FM: (i) When the inner product is positive, this term pushes up the prediction to 1. (ii) When negative, this term pulls down the prediction to 0. (iii) When near 0, this term does not contribute to the prediction.

5.3.1 Mean Embedding. Different fields have different numbers of categories, ranging from tens (e.g., device) to millions (e.g., IP address), resulting in difficulties for commonly-used unsupervised methods (e.g., PCA, t-SNE [29]). Taking PCA as an example, the objective of PCA is defined as the summed reconstruction errors, $Err = \sum_i \sum_{j=1}^{N_i} err(\mathbf{v}_i^j; \theta)$, where $\mathbf{v}_i^j$ is the embedding vector of the j-th category of field i, N_i is the field size of field i. Recall the example in Table 1, there are 1009 embedding vectors, 7 for WEEKDAY, 2 for GENDER, and 1000 for CITY. Thus the principal components will be dominated by CITY, and the interactions between CITY and GENDER are no longer mentained.

Therefore, we turn to field-level analysis. The simplest way is using mean embeddings. A mean embedding is the center of the embedding vectors within a field, $\bar{\mathbf{v}}_i = \sum_{j=1}^{N_i} \mathbf{v}_i^j / N_i$.

5.3.2 Visualization. We choose FM, FFM and KFM to visualize feature interactions. The parameter settings follow Table 5, where the embedding size of FM and KFM is 40 and the embedding size of FFM is 4. Fig. 19 shows the heatmaps generated from FM, FFM and KFM. The x- and y-axis of Fig. 19 both represent the 24 fields of Avazu, and the value of grid (i, j) is the inner/kernel product of the mean embeddings of field i and field j. The diagonal elements are set to 0, since a field does not interact with itself.

We assume feature interactions should be sparse. (i) In category level, m-order feature combinations are roughly $C_n^m \bar{N}_i^m$, where C_n^m denotes the combination number, $\bar{N}_i$ denotes the average field size. Compared with the enormous feature combinations, feature interactions should be sparse in practice, due to the rare positive responses. (ii) In field level, if a field has strong interactions with all other fields, this field provides little interactive information. Thus inter-field interactions should also be sparse.

According to the colorbar, we focus on bright (yellow or orange) and dark (dark blue) points because they have large absolute values, and we neglect light blue points because they have values

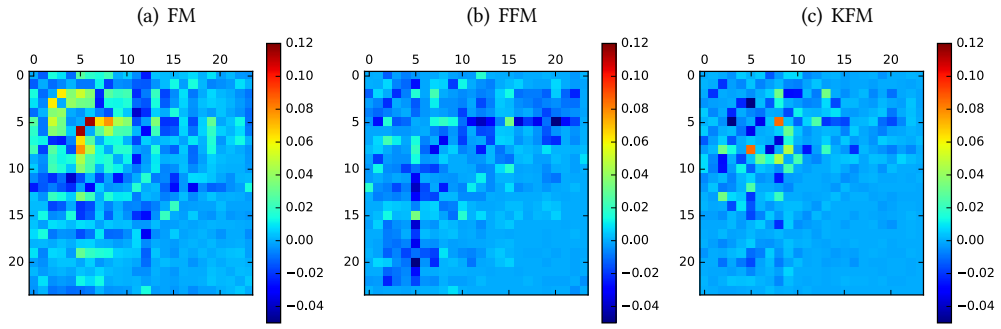

Fig. 19. Feature interaction heatmaps of FM, FFM, and KFM on Avazu. *Note:* The x- and y-axis both represent fields in Avazu, these 24 fields are shown in Table 12.

Table 12. Fields of Avazu.

Field 1-13	Field 14-24
C1, banner_pos, site_id, site_domain, site_category, app_id, app_domain, app_category, device_id, device_ip, device_model, device_type, device_conn_type	C14, C15, C16, C17, C18, C19, C20, C21, day, hour, weekday

near 0. If the interaction between field i and field j is large, there will be a bright/dark point at grid (i, j). If the interactions between field i and other fields are all large values, there will be a bright/dark bar in the i-th row/column. With the sparse assumption of feature interactions, we expect isolated bright/dark points instead of bright/dark bars in a heatmap.

In Fig. 19(a), the bright/dark bars in the left-top corner indicate these fields are strongly correlated. Therefore, we conclude FM has the coupled gradient issue, as discussed in Section 3.1. In Fig. 19(b), FFM learns clearer patterns than FM, since the bright/dark bars are shorter than those of FM. However, the short bars are still not expected, showing that FFM is limited by the small embedding size, as discussed in Section 3.1. In Fig. 19(c), we find that most of the bars disappear, and clearly isolated points are displayed. This figure proves that kernel product successfully solves the coupled gradient issue.

We find most useful feature interactions appear in the left-top corner (field 1 - field 13), which can guide feature engineering and model design. For example, we can use more complex models to capture these fields, or design higher-order cross features among these fields. For the right-bottom corner (field 14-24), these fields provide less interactive information. And a model can be compressed if we remove interactive parameters of these fields. The 24 Avazu fields are listed in Table. 12.

5.4 Training Difficulty of Gradient-based DNN

In Section 3.2, we discuss the insensitive gradient issue of gradient-based DNN from a theoretical view. In this section, we conduct a synthetic experiment to support the discussion. This synthetic dataset is generated from a poly-2 function, where the bi-linear terms are analogous to interactions between categories. Based on this dataset, we investigate (i) the impact of data sparsity on DNN, and (ii) the ability of DNN in fitting poly-2 functions.

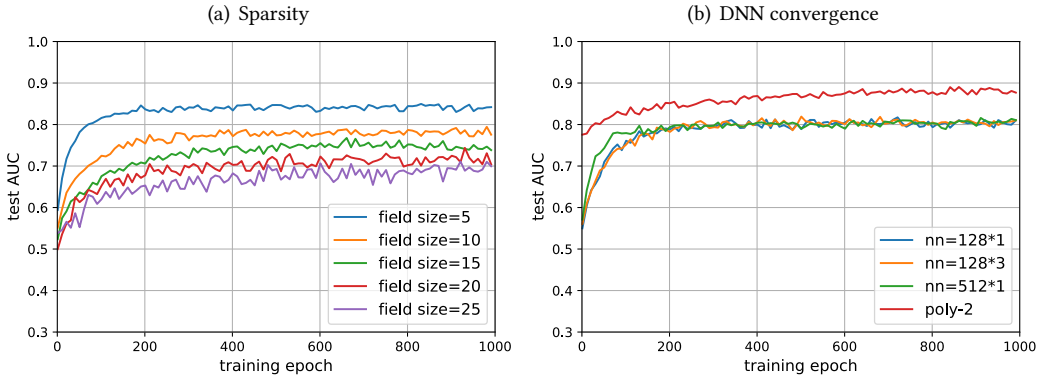

Fig. 20. DNN training curves of the synthetic experiments. *Note:* Plot (a) verifies the influence of data sparsity. Plot (b) verifies DNN as not a perfect function approximator in such a problem.

The input $\mathbf{x}$ of this dataset is randomly sampled from N categories of n fields, where each field size N_i is randomly selected. The output y is binary labeled depending on the sum of linear terms and bi-linear terms

$$y = \delta\Big(\sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} v_{i,j} x_i x_j + b + \epsilon\Big) \tag{49}$$

$$\delta(z) = \begin{cases} 1, & \text{if } z \geq \text{threshold} \\ 0, & \text{otherwise} \end{cases} . \tag{50}$$

The data distribution $p(\mathbf{x})$ and $\mathbf{w}, \mathbf{v}, b$ are randomly sampled and fixed, and the data pairs $\{\mathbf{x}, y\}$ are i.i.d. sampled to build the training and validation datasets. We also add a small random noise ϵ to the sampled data. We use DNNs with different depths and widths to fit the synthetic data. We use AUC to evaluate these models on the validation dataset.

First, we study the impact of data sparsity by changing field sizes. We sample a data instance from $p(\mathbf{x})$ and determine its label through Eq. (49), with each field one-hot encoded. It is a reciprocal relationship between field size and data sparsity. The size of each field is a random number between 1 and $2N/n$ (so that the average field size is N/n), where $N = \{200, 400, 600, 800, 1000\}$, $n = 40$. We use a DNN with 3 hidden layers of 128 neurons to fit the training dataset with different N values, and the evaluation curves are shown in Fig. 20(a). From this figure, we conclude DNN training is more difficult if input data is sparser.

Second, we choose field size $= 10$, $N = 400$ to test DNN convergence. Fig. 20(b) compares three types of DNNs: 1 hidden layer of 128 neurons, 1 hidden layer of 512 neurons, and 3 hidden layers of 128 neurons. We use Eq. (49) to fit this data, namely poly-2 regression (for short, poly-2). Because the ground-truth is a poly-2 function, we treat poly-2 as the performance upper bound, shown as the red curve. From this figure we can see, there is a consistent gap between DNNs and poly-2. And unfortunately, increasing width or depth does not improve the performance of DNN. In this experiment, we also try DNN with {1, 3, 5} hidden layers of {128, 512, 1024} neurons, but the performance changes slightly in different settings. This figure indicates that, in spite of universal approximation property, DNN cannot fit a simple poly-2 function perfectly through gradient descent, thus gradient-based DNN may not be a perfect function approximator for user response prediction.

Poly-2 and libFFM are also studied in [21], and FFM achieves promising results. Thus, it is promising to add a feature extractor to explore interactive patterns in a DNN model. This synthetic experiment validates the discussion in Section 3.2, and highlights the necessity of extracting feature interactions from the sparse input, as a complement of DNN classifiers.

6 CONCLUSION

In this paper, we study user response prediction over multi-field categorical data. We find a coupled gradient issue of latent vector-based models and propose to learn field-aware feature interactions instead. We propose kernel product methods to solve this problem as well as solving the memory bottleneck of FFM. We also find an insensitive gradient issue of DNN-based models, and we propose several PNN models to solve this problem. The improvement of PNNs over other DNN-based models proves the necessity of product layers. With both expressive feature extractors and powerful DNN classifiers, PNNs consistently outperform 8 baselines and achieve the state-of-the-art performance on 4 industrial datasets. Besides, PIN makes great CTR improvements in online A/B test. In future work, we will study different kernels and micro networks, and explore the generalization ability of DNN-based models in information systems.

ACKNOWLEDGMENTS

The work is sponsored by Huawei Innovation Research Program. The corresponding author Weinan Zhang thanks the support of National Natural Science Foundation of China (61632017, 61702327, 61772333), Shanghai Sailing Program (17YF1428200).

REFERENCES

[1] Deepak Agarwal, Rahul Agrawal, Rajiv Khanna, and Nagaraj Kota. 2010. Estimating rates of rare events with multiple hierarchies through scalable log-linear models. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 213–222.

[2] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. 2006. Learning user interaction models for predicting web search result preferences. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 3–10.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[4] Bokai Cao, Lei Zheng, Chenwei Zhang, Philip S Yu, Andrea Piscitello, John Zulueta, Olu Ajilore, Kelly Ryan, and Alex D Leow. 2017. DeepMood: Modeling Mobile Phone Typing Dynamics for Mood Detection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 747–755.

[5] Olivier Chapelle and Ya Zhang. 2009. A dynamic bayesian network click model for web search ranking. In *Proceedings of the 18th international conference on World wide web*. ACM, 1–10.

[6] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[7] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. ACM, 7–10.

[8] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 191–198.

[9] Ying Cui, Ruofei Zhang, Wei Li, et al. 2011. Bid landscape forecasting in online ad exchange marketplace. In *SIGKDD*. ACM, 265–273.

[10] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

[11] Andries P Engelbrecht, Ap Engelbrecht, and A Ismail. 1999. Training product unit neural networks. (1999).

[12] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 249–256.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[14] Thore Graepel, Joaquin Q Candela, Thomas Borchert, et al. 2010. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoft's bing search engine. In *ICML*. 13–20.

[15] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. *arXiv preprint arXiv:1703.04247* (2017).

[16] Xiangnan He and Tat-Seng Chua. 2017. Neural Factorization Machines for Sparse Predictive Analytics. *SIGIR* (2017).

[17] 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*. International World Wide Web Conferences Steering Committee, 173–182.

[18] Xinran He, Junfeng Pan, Ou Jin, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. ACM, 1–9.

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

[20] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*. 448–456.

[21] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-aware factorization machines for CTR prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 43–50.

[22] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[23] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-Normalizing Neural Networks. *arXiv preprint arXiv:1706.02515* (2017).

[24] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009).

[25] Kuang-chih Lee, Burkay Orten, Ali Dasdan, et al. 2012. Estimating conversion rate in display advertising from past erformance data. In *SIGKDD*. ACM, 768–776.

[26] Min Lin, Qiang Chen, and Shuicheng Yan. 2013. Network in network. *arXiv preprint arXiv:1312.4400* (2013).

[27] Qiang Liu, Feng Yu, Shu Wu, et al. 2015. A Convolutional Click Prediction Model. In *CIKM*. ACM, 1743–1746.

[28] Chun-Ta Lu, Lifang He, Weixiang Shao, Bokai Cao, and Philip S Yu. 2017. Multilinear factorization machines for multi-task multi-view learning. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 701–709.

[29] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, Nov (2008), 2579–2605.

[30] H Brendan McMahan, Gary Holt, David Sculley, et al. 2013. Ad click prediction: a view from the trenches. In *SIGKDD*. ACM, 1222–1230.

[31] Aditya Krishna Menon, Krishna-Prasad Chitrapura, Sachin Garg, et al. 2011. Response prediction using collaborative filtering with hierarchies and side-information. In *SIGKDD*. ACM, 141–149.

[32] Claudia Perlich, Brian Dalessandro, Rod Hook, et al. 2012. Bid optimizing and inventory scoring in targeted online advertising. In *SIGKDD*. ACM, 804–812.

[33] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based neural networks for user response prediction. *ICDM* (2016).

[34] J Ross Quinlan. 1996. Improved use of continuous attributes in C4. 5. *Journal of artificial intelligence research* 4 (1996), 77–90.

[35] Kan Ren, Weinan Zhang, Yifei Rong, Haifeng Zhang, Yong Yu, and Jun Wang. 2016. User Response Learning for Directly Optimizing Campaign Performance in Display Advertising. In *CIKM*.

[36] Steffen Rendle. 2010. Factorization machines. In *ICDM*. IEEE, 995–1000.

[37] Matthew Richardson, Ewa Dominowska, and Robert Ragno. 2007. Predicting clicks: estimating the click-through rate for new ads. In *WWW*. ACM, 521–530.

[38] Tim Salimans and Diederik P Kingma. 2016. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*. 901–909.

[39] Shai Shalev-Shwartz, Ohad Shamir, and Shaked Shammah. 2017. Failures of gradient-based deep learning. In *International Conference on Machine Learning*. 3067–3075.

[40] Ying Shan, T Ryan Hoens, Jian Jiao, Haijing Wang, Dong Yu, and JC Mao. 2016. Deep Crossing: Web-scale modeling without manually crafted combinatorial features. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 255–262.

[41] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[43] Anh-Phuong Ta. 2015. Factorization machines with follow-the-regularized-leader for CTR prediction in display advertising. In *IEEE BigData*. IEEE, 2889–2891.
[44] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. 2015. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1235–1244.
[45] Xuejian Wang, Lantao Yu, Kan Ren, Guanyu Tao, Weinan Zhang, Yong Yu, and Jun Wang. 2017. Dynamic attention deep model for article recommendation by learning human editors' demonstration. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2051–2059.
[46] Jun Xiao, Hao Ye, Xiangnan He, Hanwang Zhang, Fei Wu, and Tat-Seng Chua. 2017. Attentional factorization machines: Learning the weight of feature interactions via attention networks. *arXiv preprint arXiv:1708.04617* (2017).
[47] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*. 2048–2057.
[48] Gui-Rong Xue, Hua-Jun Zeng, Zheng Chen, Yong Yu, Wei-Ying Ma, WenSi Xi, and WeiGuo Fan. 2004. Optimizing web search using web click-through data. In *CIKM*.
[49] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. 2017. Deep sets. arXiv preprint. *arXiv preprint arXiv:1703.06114* (2017).
[50] Shuai Zhang, Lina Yao, and Aixin Sun. 2017. Deep learning based recommender system: A survey and new perspectives. *arXiv preprint arXiv:1707.07435* (2017).
[51] Weinan Zhang, Tianming Du, and Jun Wang. 2016. Deep Learning over Multi-field Categorical Data: A Case Study on User Response Prediction. *ECIR* (2016).
[52] Weinan Zhang, Shuai Yuan, and Jun Wang. 2014. Optimal real-time bidding for display advertising. In *SIGKDD*. ACM, 1077–1086.
[53] Yuyu Zhang, Hanjun Dai, Chang Xu, et al. 2014. Sequential click prediction for sponsored search with recurrent neural networks. *arXiv preprint arXiv:1404.5772* (2014).