APG: Adaptive Parameter Generation Network for Click-Through Rate Prediction

Bencheng Yan[*], **Pengjie Wang**[*], **Kai Zhang, Feng Li, Hongbo Deng, Jian Xu, Bo Zheng**[†]
Alibaba Group
China
{bencheng.ybc,pengjie.wpj,victorlanger.zk,
adam.lf,dhb167148,xiyu.xj,bozheng}@alibaba-inc.com

Abstract

In many web applications, deep learning-based CTR prediction models (deep CTR models for short) are widely adopted. Traditional deep CTR models learn patterns in a static manner, i.e., the network parameters are the same across all the instances. However, such a manner can hardly characterize each of the instances which may have different underlying distributions. It actually limits the representation power of deep CTR models, leading to sub-optimal results. In this paper, we propose an efficient, effective, and universal module, named as Adaptive Parameter Generation network (APG), which can dynamically generate parameters for deep CTR models on-the-fly based on different instances. Extensive experimental evaluation results show that APG can be applied to a variety of deep CTR models and significantly improve their performance. Meanwhile, APG can reduce the time cost by 38.7% and memory usage by 96.6% compared to a regular deep CTR model. We have deployed APG in the industrial sponsored search system and achieved 3% CTR gain and 1% RPM gain respectively.

1 Introduction

Recently, deep CTR models have achieved great success in various web applications such as recommender systems, web search, and online advertising [4, 9, 31, 14]. Formally, a regular deep CTR model can be expressed as $y_i = \mathcal{F}_\Theta(\boldsymbol{x}_i)$ where $\boldsymbol{x}_i, y_i$ are the input features and the predicted CTR of the instance i respectively, Θ is the parameter, and $\mathcal{F}$ is usually implemented as a neural network.

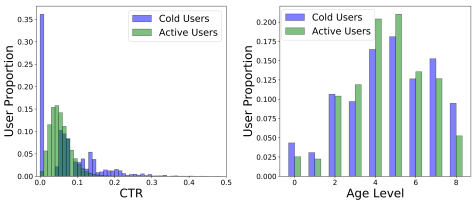


Figure 1: An example of feature distribution from different users (i.e., active vs. cold users). **Left:** the CTR distributions are varied from different user groups and a custom pattern should be considered. **Right:** A common pattern is welcomed to model the similar age distributions from different groups.

Improving the performance of deep CTR models has been a very hot topic in the research and industrial areas. Existing works can be broadly divided into two categories: (1) Focusing on $\boldsymbol{x}_i$, more and more elaborated information (e.g., user behavior features [37, 22], multimodal information [3, 11], knowledge graph [36, 30], etc) is introduced to enrich feature space (i.e., $\boldsymbol{x}_i$); (2) Focusing on $\mathcal{F}$, advanced architectures (including feature interaction modeling [4, 9, 31], automated architecture search [14, 26] and so on) are designed to improve the model performance.

[*] ∗ These authors contributed equally to this work and are co-first authors.
[†] † Corresponding author

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

However, few works focus on the improvement of the model parameters Θ, especially for the weight matrix $\boldsymbol{W} \in \mathbb{R}^{N \times M}$ used in hidden layers of deep CTR models [3]. It is another orthogonal aspect for the performance improvement. Actually, most of the existing works simply adopt a static manner, i.e., all the instances share the same parameters $\boldsymbol{W}$. We argue that such a manner is sub-optimal for pattern learning and limits the representation power of deep CTR models. On the one hand, although the common patterns among instances can be captured by the shared parameters $\boldsymbol{W}$, it is not friendly to custom pattern modeling. Specifically, taking the industrial sponsored search system as an example, the feature distribution can be varied from different users (e.g., active vs. cold users), different categories (e.g., clothing vs. medicine), and so on (see Figure 1 (a) as an example). Simply applying the same parameters across all the instances can hardly capture the characteristic of each instance from different distributions. On the other hand, the learned common pattern may not be suitable for each of the instances. For example, the shared parameters tend to be dominated by high-frequency features and may give a misleading decision for the long-tailed instances. This leads us to the following question: *Do we really need the same and shared parameters for all instances?*

Ideally, besides modeling the common pattern, the parameters should be more adaptive and can be dynamically changed for different instances to capture custom patterns at the same time. Then, the representation power (or model capacity) can be enhanced by the dynamically changed parameter space. To achieve this goal, we design a new paradigm for CTR prediction. The key insight is to propose an Adaptive Parameter Generation network (APG) to dynamically generate parameters depending on different instances. Firstly, we propose a basic version (Section 3.1) of APG which can be expressed as $y_i = \mathcal{F}_{\mathcal{G}(\boldsymbol{z}_i)}(\boldsymbol{x}_i)$ where $\mathcal{G}$ refers a neural network (e.g., MLP) and generates the adaptive parameters $\boldsymbol{W}_i \in \mathbb{R}^{N \times M}$ by the input-aware condition $\boldsymbol{z}_i \in \mathbb{R}^D$. However, the basic model suffers from two problems: (1) inefficient in time and memory. Directly generating the weight matrix $\boldsymbol{W}_i$ of a deep CTR model needs *$\mathcal{O}$(NMD)* cost in computation and memory storage, which is D times the cost in a regular deep CTR model and is costly especially for a web-scale application where N, M are usually set as a large value (e.g., *N=M=1,000*). The empirical results (Section 4.4) also show the basic model needs an extra $111\times$ training time and $31\times$ memory usage. (2) sub-effective in pattern learning. The parameter generation process is totally dependent on the condition $\boldsymbol{z}_i$, which may only capture the custom patterns and ignore the common patterns which contribute to understand users' behaviors (see Figure 1 (b)), leading to sub-effective pattern learning.

Then, we extend the above basic model to an efficient and effective version of APG: **(1) For the efficiency,** motivated by the low-rank methods [16, 1] which show that the weight matrix resides on a low intrinsic dimension, we parameterize the target weight matrix $\boldsymbol{W}_i$ as the production of three low-rank matrices $\boldsymbol{U}_i\boldsymbol{S}_i\boldsymbol{V}_i$ where $\boldsymbol{U}_i \in \mathbb{R}^{N \times K}, \boldsymbol{V}_i \in \mathbb{R}^{K \times M}, \boldsymbol{S}_i \in \mathbb{R}^{K \times K}$ and $K \ll min(M, N)$. In addition, the decomposed feed-forwarding is proposed to avoid the heavy computation of the weight matrix $\boldsymbol{W}_i$ reconstruction. Then we further take the center matrix $\boldsymbol{S}_i$ as the specific parameters which are dynamically generated to capture the custom patterns and the rest two matrices $\boldsymbol{U}, \boldsymbol{V}$ as the shared parameters which are randomly initialized and shared across instances to capture common patterns. In this way, the complexity of generating the specific parameters can be easily controlled by setting a small K. As a result, APG achieves *$\mathcal{O}$(KKD+NK+MK+KK)* time complexity and *$\mathcal{O}$(KKD+NK+MK)* memory complexity compared to that both have *$\mathcal{O}$(NM)* in a regular deep CTR model. We empirically find APG can speed up the training time by 38.7% and reduce memory usage by 96.6% relative to a regular deep CTR model (Section 4.4). **(2) For the effectiveness,** apart from the natural effectiveness of APG in custom pattern learning, to model the common pattern, the shared weights $\boldsymbol{U}$ and $\boldsymbol{V}$ are considered in the adaptive parameter generation. Then we further extend $\boldsymbol{U}$ and $\boldsymbol{V}$ to an over parameterization version which enriches the model capacity without any additional memory and time cost during inference. Besides, we also find there exists inherent similarity between different generated $\boldsymbol{S}_i$ which implicitly model the common information (Section 4.6).

In summary, the contributions of this paper are presented as follows: (1) We propose a new learning paradigm in deep CTR models where the model parameters are input-aware and dynamically generated to boost the representation power. It is orthogonal to many prior methods and is a universal module that can be easily applied in most existing deep CTR models. (2) We present APG which generates the adaptive parameters in an efficient and effective way, and theoretically analyze the

[3] For simplicity, in this paper, we mainly discuss the weight matrix W. Our method can also be easily applied to the parameters of other modules (e.g., transformer, attention network, etc) in deep CTR models since most of them can be regarded as a variety of MLP with a set of weight matrices [35].

computation and memory complexity. (3) Extensive experimental evaluation results demonstrate that the proposed method is a universal module and can improve the performance of most of the existing deep CTR models. we also provide a systematic evaluation of APG in terms of training time and memory consumption. Finally, we have developed APG in the industrial sponsored search system and achieved 3% CTR gain and 1% RPM gain respectively.

2 Related Work

Deep CTR Models. A traditional CTR prediction method usually adopts a deep neural network to capture the complex relations between users and items. Most of them focus on (1) introducing abundant useful informations [37, 22, 3, 11, 36, 30] to improve the model understanding, (2) designing advanced architecture [4, 9, 31, 14, 26] to achieve better performance. All of them adopt a static parameters manner which limits the model capacity of these methods, leading to sub-optimal performance

Coarse-grained Parameter Allocating. There are some research areas that bring a coarse-grained parameter allocating strategy that may be related to our goals, including multi-domain learning [25, 15] and multi-task learning [19, 21]. Both of them maintain and allocate different parameters to different domains or tasks manually. However, such a coarse-grained parameter allocating can hardly be extended to a fine-grained (e.g., user, item, or instances sensitive) manner. It not only costs too much memory to maintain and store a large number of parameters when considering the fine-grained modeling but also lacks flexibility and generalization since the parameter allocation is manually pre-defined. We also conduct experiments to compare this kind of methods (Section 4.5).

Dynamic Deep Neural Networks. Our method is related to recent works of dynamic neural networks used in computer vision and natural language processing, in which the model parameters [13, 32, 7, 23, 10, 20, 29] and architecture [17, 34] can be dynamically changed. In our paper, we take the first step to bring the idea about dynamic networks into deep CTR models and develop it in real applications where the efficiency in computation and memory is extremely needed and the common and custom pattern learning are also required.

3 Method

In this paper, we denote scalars, vectors and tensors with lower-case (or upper-case), bold lower-case, and bold upper-case letters, e.g., n (or N), $\boldsymbol{x}$, $\boldsymbol{X}$, respectively.

3.1 Basic Model

In this section, we introduce the basic version of APG. Generally speaking, the basic idea of APG is to dynamically generate parameters $\boldsymbol{W}_i$ by different condition $\boldsymbol{z}_i$, i.e., $\boldsymbol{W}_i = \mathcal{G}(\boldsymbol{z}_i)$ where $\mathcal{G}$ refers to the adaptive parameter generation network. Then the generated parameters are applied to the deep CTR models, i.e., $y_i = \mathcal{F}_{\mathcal{G}(\boldsymbol{z}_i)}(\boldsymbol{x}_i)$. Next, we present (1) how to design the condition $\boldsymbol{z}_i$ for different instance i and (2) how to implement $\mathcal{G}$.

3.1.1 Condition Design

In this paper, we propose three kinds of strategies (including group-wise, mix-wise, and self-wise) to design different kinds of $\boldsymbol{z}_i$.

Group-wise. The group-wise strategy tries to generate different parameters depending on different instance groups. The purpose is that, sometimes, the instances can be divided into different groups [2] and instances in the same group may have similar patterns. Thus $\boldsymbol{z}_i$ is the representation identifying different groups. An example can be found in Appendix A.1.

Mix-wise. To further enrich the expression power and flexibility of the adaptive parameters, a mix-wise strategy is designed to take multiple conditions into consideration. Specifically, given k condition embeddings $\{\boldsymbol{z}_i^j \in \mathbb{R}^d | j \in \{0, 1, .., k-1\}\}$ of the instance i, we propose two aggregation policies to consider different conditions at the same time: (1) *Input Aggregation.* This policy firstly aggregates different condition embeddings and then feeds the aggregated embeddings into APG to obtain the mix-wise based parameters. The aggregation functions can be (but are not limited to):

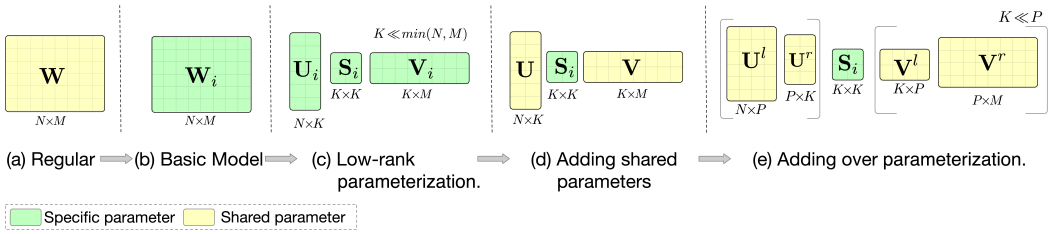


Figure 2: The comparison of different versions of APG.

Concatenation, Mean, and Attention. (2) *Output Aggregation* This policy firstly feeds different $\boldsymbol{z}_i^j$ into different APG and obtains the corresponding parameters $\boldsymbol{W}_i^j$ respectively. Then the aggregation is applied to these adaptive parameters $\boldsymbol{W}_i^j$. Similarly, the aggregation function includes Concatenation, Mean, and Attention. Examples can be found in Appendix A.2.

Self-wise. The above two strategies need additional prior knowledge to generate parameters. Self-wise strategy tries to use a simple and easily obtained knowledge (i.e., self-knowledge) to guide parameters generation, i.e., the network parameters are generated by their own input. For example, for the 0-th hidden layer of a deep CTR model, we can set $\boldsymbol{z}_i = \boldsymbol{x}_i$. For l-th hidden layer, $\boldsymbol{z}_i = \boldsymbol{h}^{\boldsymbol{l-1}}$ where $\boldsymbol{h}^{\boldsymbol{l-1}}$ is the input of the l-th hidden layer.

3.1.2 Parameters Generation

After obtaining the conditions, we adopt a multilayer perceptron[4] to generate parameters depending on these conditions (see Figure 2 (b)), i.e.,

$$\boldsymbol{W}_i = reshape(MLP(\boldsymbol{z}_i)) \tag{1}$$

where $\boldsymbol{W}_i \in \mathbb{R}^{N \times M}$, $\boldsymbol{z}_i \in \mathbb{R}^D$, and the operation $reshape$ refers to reshaping the vectors produced by the MLP into a matrix form. Then a deep CTR model with APG can be expressed as:

$$y_i = \sigma(\boldsymbol{W}_i \boldsymbol{x}_i) \tag{2}$$

where σ is the activation function. Here we take a deep CTR model with an MLP layer as an example, and other deep CTR models can also be easily extended since most of them can be regarded as a variety of MLP with a set of weight matrices [35].

3.2 Effective and Efficient Adaptive Parameter Generation Network

As introduced in Section 1, the above basic model has two problems: (1) inefficient in time and memory and (2) sub-effective in pattern learning. To address the above two problems, we propose some extensions including low-rank parameterization, decomposed feed-forwarding, parameter sharing, and over parameterization to parameterize the weight matrix in an efficient and effective way.

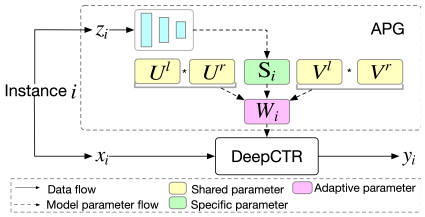


Figure 3: The framework of APG. For simplicity, the process of decomposed feed-forwarding is omitted in this figure.

Low-rank parameterization. Inspired by the recent success of the low-rank methods [16, 1] which have demonstrated that strong performance can be achieved by optimizing a task in a low-rank subspace, we hypothesize that the adaptive parameters also have a low "intrinsic rank". To this end, we propose to parameterize the weight matrix $\boldsymbol{W}_i \in \mathbb{R}^{N \times M}$ as a low-rank matrix, which is the product of three sub-matrices $\boldsymbol{U}_i \in \mathbb{R}^{N \times K}, \boldsymbol{S}_i \in \mathbb{R}^{K \times K}, \boldsymbol{V}_i \in \mathbb{R}^{K \times M}$ and the rank $K \ll min(N, M)$ (see Figure 2 (c)). Formally, the weight generation process can be expressed as:

$$\boldsymbol{U}_i, \boldsymbol{S}_i, \boldsymbol{V}_i = reshape(MLP(\boldsymbol{z}_i)) \tag{3}$$

[4]In this paper, we take MLP as an example, and other implementations of $\mathcal{G}$ can also be considered.

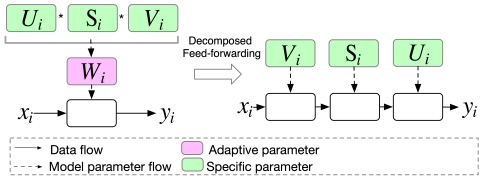


Figure 4: An example of the decomposed feed-forwarding.

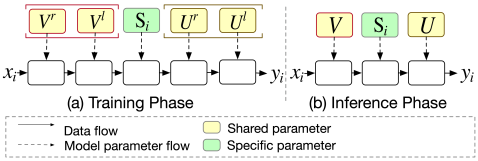


Figure 5: An example of over parameterization.

Intuitively, we can set a small value of K to control the time and memory cost. Meanwhile, ignoring the substantial storage and computation cost, K can also be set to a higher value to enlarge the adaptive parameter space (see detailed discussion in Appendix E). Overall, through the low-rank parameterization, we can significantly reduce dimensionality in the parameter space, i.e., $K \ll min(N, M)$, to enable a more compact model. As a result, the computation complexity of the specific parameter generation process can be reduced to *$\mathcal{O}$((NK+MK+KK)D)* (see Table 1). The memory cost is also reduced to *$\mathcal{O}$((NK+MK+KK)D)* (see Table 1).

Decomposed Feed-forwarding. After low-rank parameterization, the Eq 2 can be written as

$$y_i = \sigma(\boldsymbol{W}_i \boldsymbol{x}_i) = \sigma((\boldsymbol{U}_i \boldsymbol{S}_i \boldsymbol{V}_i) x_i) = \sigma(\boldsymbol{U}_i(\boldsymbol{S}_i(\boldsymbol{V}_i \boldsymbol{x}_i))) \tag{4}$$

Here instead of directly reconstructing the weight matrix $\boldsymbol{W}_i$ by the sub-matrix production, we design a decomposed feed-forwarding and apply $\boldsymbol{x}_i$ to each sub-matrix sequentially (see Figure 4). Such a design helps us avoid the heavy computation of the sub-matrix production which costs *$\mathcal{O}$(NKK+NMK)*. Actually, this design benefits from the low-rank parameterization which naturally supports the decomposed feed-forwarding. Since the computation complexity in Eq 4 is *$\mathcal{O}$(NK+KK+MK)*, the total time cost of a deep CTR model with APG per layer is reduced to *$\mathcal{O}$((NK+MK+KK)(D+1))* (see Table 1).

Parameter sharing. In this section, we present our common pattern modeling strategy. Thanks to decomposing the weight matrix into $\boldsymbol{U}_i$, $\boldsymbol{S}_i$, and $\boldsymbol{V}_i$, it allows us to be more flexible. Consequently, we divide these three matrices into (1) *specific parameters* that capture custom patterns from different instances; (2) *shared parameters* that are shared across instances to characterize the common patterns. Specifically, we define $\boldsymbol{S}_i$ as the specific parameters since the matrix scale is totally controlled by K and is more efficient for the generation process. $\boldsymbol{U}$ and $\boldsymbol{V}$ are regarded as the shared parameters (see Figure 2 (d)). Then we can rewrite Eq 3 and 4 as:

$$\boldsymbol{S}_i = reshape(MLP(\boldsymbol{z}_i)) \tag{5}$$

$$y_i = \sigma(\boldsymbol{U}(\boldsymbol{S}_i(\boldsymbol{V}\boldsymbol{x}_i))) \tag{6}$$

Furthermore, such a design also contributes the efficiency. Since the size of the generated specific parameter is reduced to $K \times K$, the computation complexity of the specific parameter generation can be further reduced to *$\mathcal{O}$(KKD)* in Eq 5, and the total time cost of a deep CTR model with APG is reduced to *$\mathcal{O}$(KKD+NK+MK+KK)* (see Table 1). Meanwhile the memory cost is reduced to *$\mathcal{O}$(KKD+NK+MK)* where *NK* and *MK* refers to the storage of the shared parameters $\boldsymbol{U}$ and $\boldsymbol{V}$ respectively (see Table 1).

Over Parameterization Compared with the shared weight matrix $\boldsymbol{W} \in \mathbb{R}^{N \times M}$ in a regular deep CTR model, $\boldsymbol{U}$ and $\boldsymbol{V}$ in APG can be hardly scaled to large matrices due to the efficiency constraint with $K \ll min(N, M)$, leading to a possible performance drop. To address this problem and further enlarge the model capacity, we follow the idea about the over parameterization [1, 5] to enrich the number of the shared parameters (see Figure 2 (e)). Specifically, we replace shared parameters in Eq 6 with two large matrices, i.e.,

$$\boldsymbol{U} = \boldsymbol{U}^l \boldsymbol{U}^r, \ \boldsymbol{V} = \boldsymbol{V}^l \boldsymbol{V}^r \tag{7}$$

where $\boldsymbol{U}^l \in \mathbb{R}^{N \times P}$,$\boldsymbol{U}^r \in \mathbb{R}^{P \times K}$,$\boldsymbol{V}^l \in \mathbb{R}^{K \times P}$,$\boldsymbol{V}^r \in \mathbb{R}^{P \times M}$ and $P \gg K$. Although $\boldsymbol{U}$ (or $\boldsymbol{V}$) is exactly equal to $\boldsymbol{U}^l \boldsymbol{U}^r$ (or $\boldsymbol{V}^l \boldsymbol{V}^r$) at the mathematical perspective, replacing $\boldsymbol{V}$ with $\boldsymbol{V}^l \boldsymbol{V}^r$ can contribute into two folds: (1) Since $P \gg K$, more shared parameters are introduced to enlarge the model capacity [5, 6]; (2) The form as the multiple matrix production can result in an implicit regularization and thus enhance generalization [1].

No Additional Inference Latency and Memory Cost. During the training phase, we can set P to a large value to enlarge the model representation power. Remarkably, when $P > max(N, M)$, the shared

Table 1: The computation and memory complexity per layer of different versions of APG. SPG refers to the time or memory cost in the process of specific parameter generation. R-Wi is the time cost to reconstruct $\boldsymbol{W}_i$. FF refers to the time cost in the feed-forwarding process of a regular deep CTR model. SPS refers to the memory cost to store the shared parameters. The total computation cost is the sum of SPG, R-Wi, and FF. The total memory cost is the sum of SPG and SPS. Since over parameterization dose not introduce any additional time cost, it is not depicted here.

Versions	Computation complexity per layer				Memory complexity per layer		
	SPG	R-Wi	FF	Total Cost	SPG	SPS	Total Cost
$\boldsymbol{W}\boldsymbol{x}_i$	-	-	$\mathcal{O}(NM)$	$\mathcal{O}(NM)$	-	$\mathcal{O}(NM)$	$\mathcal{O}(NM)$
$\boldsymbol{W}_i\boldsymbol{x}_i$	$\mathcal{O}(NMD)$	-	$\mathcal{O}(NM)$	$\mathcal{O}(NMD+NM)$	$\mathcal{O}(NMD)$	-	$\mathcal{O}(NMD)$
$(\boldsymbol{U}_i\boldsymbol{S}_i\boldsymbol{V}_i)\boldsymbol{x}_i$	$\mathcal{O}((NK+MK+KK)D)$	$\mathcal{O}(NKK+NMK)$	$\mathcal{O}(NM)$	$\mathcal{O}((NK+MK+KK)D+NKK+NMK+NM)$	$\mathcal{O}((NK+MK+KK)D)$	-	$\mathcal{O}((NK+MK+KK)D)$
$\boldsymbol{U}_i(\boldsymbol{S}_i(\boldsymbol{V}_i\boldsymbol{x}_i))$	$\mathcal{O}((NK+MK+KK)D)$	-	$\mathcal{O}(NK+MK+KK)$	$\mathcal{O}((NK+MK+KK)(D+1))$	$\mathcal{O}((NK+MK+KK)D)$	-	$\mathcal{O}((NK+MK+KK)D)$
$\boldsymbol{U}(\boldsymbol{S}_i(\boldsymbol{V}\boldsymbol{x}_i))$	$\mathcal{O}(KKD)$	-	$\mathcal{O}(NK+MK+KK)$	$\mathcal{O}(KKD+NK+MK+KK)$	$\mathcal{O}(KKD)$	$\mathcal{O}(NK+MK)$	$\mathcal{O}(KKD+NK+MK)$

parameter space can be larger than that in a regular deep CTR model. During the inference phase, we explicitly pre-compute and store $\boldsymbol{V}, \boldsymbol{U}$, and use these two matrices for inference (see Figure 5). Critically, this guarantees that we can introduce abundantly shared parameters without any additional latency and memory cost during the inference phase.

3.3 Complexity

In this section, we detailedly analyze the proposed model complexity including the memory and computation complexity during the inference phase. For analysis, the parameters generation network is implemented as a single perceptron layer, and the per layer costs in a regular deep CTR model and an adaptive deep CTR model are compared. Summarization can be found in Table 1.

Memory Complexity. For a regular deep CTR model, the memory cost is $\mathcal{O}(NM)$ per layer, i.e., storing the shared weight matrix. For APG, the memory cost comes from two parts: (1) The memory cost of generating $\boldsymbol{S}_i$ in Eq 5 is $\mathcal{O}(KKD)$; (2) The shared parameters $\boldsymbol{U}, \boldsymbol{V}$ cost $\mathcal{O}((N + M)K)$ during the inference phase. Then the total memory complexity of APG is $\mathcal{O}(KKD+NK+MK)$ per layer.

Computation Complexity. A regular deep CTR model needs $\mathcal{O}(NM)$ per layer for the feed-forward computation. APG needs $\mathcal{O}(KKD)$ in Eq 5 to calculate the specific parameters. Meanwhile, the feed-forwarding computation of a deep CTR model is $\mathcal{O}(NK+KK+MK)$ by the decomposed feed-forwarding in Eq 8. In total, the computation complexity is $\mathcal{O}(KKD+NK+KK+MK)$.

In summary, APG has $\mathcal{O}(KKD+NK+MK)$ in memory cost and $\mathcal{O}(KKD+NK+KK+MK)$ in computation cost. Since $K \ll min(N, M)$ and D is usually set smaller than N, M, the memory and computation cost of APG can even be much smaller than that (i.e., $\mathcal{O}(NM)$) in a regular deep CTR model. The experimental results in Section 4.4 also show the efficiency of APG.

4 Experiments

4.1 Experimental Settings

The detailed settings including datasets, baselines, and training details are presented in Appendix B.

Datasets. Four real-world datasets are used including **Amazon**, **MovieLens**, **IAAC**, and **IndusData**. The first three are public datasets and the last is an industrial dataset.

Baselines. Here, we compare our method with two kinds of methods (1) *Existing CTR prediction methods* include WDL[4], PNN[24], FIBINET[12], DIFM[18], DeepFM[9], DCN[31], and AutoInt[27]; (2) *Coarse-grained parameter allocating methods.* multi-task learning: MMoE [19] and multi-domain learning: Star [25].

Training Details. See Appendix B.3 for the detailed introduction.

4.2 Performance Evaluation with Existing Deep CTR Models

Results on public datasets. We first apply APG on CTR prediction tasks on public datasets. APG is a universal module and can be applied to most of the existing deep CTR models. Hence, to evaluate the performance of APG, we apply APG to various existing deep CTR models, and report the results of the original model (denoted as Base) and the model with APG. Here, AUC (%) [8] score is reported as the metric [5]. The results are shown in Table 2. We can find that with the help of APG, all of

[5]Note 0.1% absolute AUC gain is regarded as significant for the CTR task [37, 28, 4].

Table 2: The AUC (%) results of Click-Through Rate (CTR) prediction on different datasets. Note Base refers to the original results of the corresponding methods and Base+APG refers to the results with the help of APG. Ave is the average results across all cases. Δ refers the improvement of Base+APG compared to Base.

Data	Method	WDL	PNN	FIBINET	DIFM	DeepFM	DCN	AutoInt	Ave
MovieLens	Base	79.21	79.5	79.78	79.84	79.3	79.29	79.36	79.46
	Base+APG	**79.73**	**79.67**	**79.82**	**79.94**	**79.60**	**79.62**	**79.64**	**79.70**
	Δ	+0.39	+0.17	+0.04	+0.10	+0.30	+0.33	+0.28	+0.24
Amazon	Base	69.15	69.16	68.88	69.17	69.1	68.98	68.96	69.06
	Base+APG	**69.43**	**69.37**	**69.19**	**69.23**	**69.43**	**69.42**	**69.38**	**69.34**
	Δ	+0.22	+0.21	+0.31	+0.06	+0.33	+0.44	+0.42	0.28
IAAC	Base	65.17	65.3	65.15	65.76	65.64	64.78	64.99	65.26
	Base+APG	**65.94**	**65.87**	**66.15**	**66.42**	**66.17**	**66.39**	**66.21**	**66.15**
	Δ	+0.77	+0.57	+1.0	+0.66	+0.53	+1.61	+1.22	+0.91

Table 3: The settings of different APG versions.

	Version	Annotation
Base	$\boldsymbol{W}\boldsymbol{x}_i$	WDL [4] as the baseline and the backbone
v1	$\boldsymbol{W}_i\boldsymbol{x}_i$	Basic model (Section 3.1)
v2	$(\boldsymbol{U}_i\boldsymbol{S}_i\boldsymbol{V}_i)\boldsymbol{x}_i$	+ Low-rank parameterization
v3	$\boldsymbol{U}_i(\boldsymbol{S}_i(\boldsymbol{V}_i\boldsymbol{x}_i))$	+ Decomposed feed-forwarding
v4	$\boldsymbol{U}(\boldsymbol{S}_i(\boldsymbol{V}\boldsymbol{x}_i))$	+ Parameter sharing
v5	$(\boldsymbol{U}^l\boldsymbol{U}^r)(\boldsymbol{S}_i((\boldsymbol{V}^l\boldsymbol{V}^r)x_i))$	+ Over parameterization

Table 4: The AUC results of the evaluation of different versions of APG. Δ refers the difference relative to Base.

	MovieLens	Amazon	IAAC	Ave(AUC)	Ave(Δ)
Base	79.21	69.15	65.17	71.18	—
v1	79.51	69.33	65.52	71.45	+0.27
v2	79.49	69.24	65.61	71.45	+0.27
v4	79.61	69.36	65.78	71.58	+0.40
v5	79.73	69.43	65.94	71.70	+0.52

the methods achieve a significant improvement on all datasets. For example, the gains of DCN is 0.33% $\sim$ 1.61% (other methods also can obtain similar improvement). It demonstrates that (1) giving adaptive parameters for models can enrich the parameter space and learn more useful patterns for different instances; (2) the proposed APG is a universal framework that can boost the performance of many other methods. Such nice property encourages APG to be applied to various scenarios and various methods.

Results on industrial application. We also develop APG in the industrial sponsored search system, and achieve 0.2% AUC gain on the industrial dataset, 3% CTR gain and 1% RPM (Revenue Per Mile) gain during online A/B test. Detailed analysis are presented in Appendix F.

4.3 Effectiveness Evaluation

In this section, we implement various versions of APG (see Table 3) and conduct experiments to detailedly evaluate the effectiveness of APG. The AUC results are reported in Table 4. Note since the design of decomposed feed-forwarding does not influence the AUC performance of APG, we do not compare version v3 in this section.

The impact of the basic model. Compared with the base, v1 achieves significant improvements on AUC results overall datasets. It demonstrates the effectiveness to introduce specific parameters to give a custom understanding of different instances.

The impact of the low-rank parameterization. Since the parameter matrix resides on a low intrinsic dimension [16, 1], it encourages a low-performance drop when adopting a low-rank based method to reduce the computation and memory cost. Considering the effectiveness, v1 and v2 do not provide much performance difference. Both of them achieve high performance and perform much better than Base. More importantly, by low-rank parameterization, we can generate adaptive parameters in a more efficient way (see Section 4.4 for details).

The impact of the parameter sharing. In APG, we introduce the shared parameters to characterize common patterns. Comparing the performance of versions v2, and v4 can further improve the performance by introducing the shared parameters, demonstrating the effectiveness of APG in common pattern modeling. Furthermore, sharing parameter also contributes to the efficiency, due to fewer specific parameters generated (see Section 4.4).

Table 5: The inference time and memory cost for different versions of APG. Δ is the relative difference with respect to Base.

Method	[393,64,32,16]		[393,128,64,32]		[393,256,128,64]		[393,512,256,128]		[393,1024,512,256]	
	Cost	Δ	**Cost**	Δ	**Cost**	Δ	**Cost**	Δ	**Cost**	Δ
Time (s)										
Base	4.52	—	5.27	—	5.89	—	7.12	—	13.04	—
v1	16.52	265.5%	50.31	854.6%	126.27	2043.8%	420.33	5803.5%	1462.78	11117.6%
v2	13.83	206.0%	39.53	682.8%	90.43	1435.3%	363.07	4999.3%	1065.35	8069.9%
v3	6.31	39.6%	5.42	7.3%	5.71	-3.1%	6.31	-11.4%	11.23	-13.9%
v4	4.49	-0.7%	4.78	-5.3%	5.46	-7.3%	5.89	-17.3%	7.99	-38.7%
Memory (M)										
Base	0.50	—	0.89	—	1.93	—	4.61	—	12.91	—
v1	11.21	2160.1%	24.20	2613.0%	56.32	2818.1%	144.84	3041.9%	419.11	3146.4%
v2	0.74	50.0%	0.91	1.7%	1.21	-37.3%	1.98	-57.0%	3.22	-75.1%
v4	0.27	-45.0%	0.29	-68.0%	0.31	-84.1%	0.35	-92.4%	0.44	-96.6%

The impact of the over parameterization. Comparing the performance with or without over parameterization (i.e., v4 vs. v5), it shows that v5 performs better than v4 in all cases, which indicates adding more shared parameters can enrich the model capacity and lead to better performance.

In addition, we also give a detailed analysis about the impact of the condition design in Appendix D and C, and the impact of the hyper-parameters in Appendix E. Furthermore, the influence to different frequency instances are also discussed in Appendix F

4.4 Efficiency Evaluation

In this section, we evaluate the time and memory efficiency of our proposed method. To this end, we train different versions (see Table 3) of APG on the dataset IAAC and analyze the influence of each extension introduced in Section 3.2. Since over parameterization does not introduce any cost, it is not considered here. The decomposed feed-forwarding does not bring extra memory cost, it is not discussed in the memory usage. For all versions, we set $K = 4$ and the backbone (also the base) is WDL with 3 hidden layers. We gradually increase the number of hidden units from [64,32,16] to [1024,512,256] with a fixed input_shape=393 to evaluate the memory and time cost in different model scales. In Table 5, we report the training time per epoch, memory usage[6] of each version.

For the basic model (v1), although it achieves high performance (see Table 4), it is time expensive (265.5% $\sim$ 11117.6% relative to Base) and memory costly (2160.1% $\sim$ 3146.6% relative to Base). Such an inefficient model can hardly be accepted in web-scale applications.

When introducing the low-rank parameterization (see v2 in Table 5), compared to the basic model, the inefficiency problem in v2 is addressed across all cases. Moreover, compared with Base, v1 can reduce memory usage (e.g., -75.1% in the large scale model). Note when we set N, M to a small value (e.g., [64,32,16] or [128,64,32]), the memory cost, theoretically in $\mathcal{O}((NK+MK)D)$, is more sensitive with K and D, leading to a little increasement. For the time efficiency, the contribution of the low-rank parameterization can be summarized as follows: (1) Although v2 still needs a high time requirement due to the weight matrix $\boldsymbol{W}_i$ reconstruction, compared with v1, the overall time cost of v2 is decreased. (2) The low-rank parameterization naturally contributes to the decomposed feed-forwarding which plays a key role in efficient learning.

For version v3 which adopts decomposed feed-forwarding, it is free from the high computation of reconstructing the weight matrix $\boldsymbol{W}_i$ and achieves great improvement by -13.9% in time cost when considering a large scale model. In addition, since v3 does not introduce any extra memory cost, it has the same memory usage as v2. Note in the small model, the computation is far from the GPU bottleneck and time cost is insensitive with GFlops, leading to less improvement.

[6]Since in this paper we mainly focus on the improvement of the hidden layers, we only count the memory cost of these hidden layers and the other parts (e.g., embedding layers) are not included here.

Table 6: The comparison with coarse-grained parameter allocating methods. Mem refers to the memory cost (M) of the hidden layers.

	MovieLens			**Amazon**			**IAAC**		
	Setting	AUC	Mem	Setting	AUC	Mem	Setting	AUC	Mem
Base	—	79.21	1.29	–	69.15	1.51	—	65.17	1.93
Base+MMoE$_{fg}$	movie	79.31	123.42	item	69.17	315.93	item	65.22	332.97
Base+MMoE$_{cg}$	movie gender	79.28	32.91	item category	69.14	5.78	item brand	65.20	70.53
Base+Star$_{fg}$	movie	79.28	3784.23	item	69.20	11034.02	item	65.36	13753.71
Base+Star$_{cg}$	movie gender	79.25	311.56	item category	69.20	42.84	item brand	65.28	2866.88
Base+APG$_{fg}$	movie	**79.58**	0.24	item	**69.35**	0.26	item	**65.76**	0.38
Base+APG$_{cg}$	movie gender	79.46	0.24	item category	69.30	0.26	item brand	65.59	0.38

When introducing the sharing parameters, v4 only needs to generate a small weight matrix, where time and memory cost (i.e., $\mathcal{O}(KKD)$) in the generation process are free from the scales of a model. Thus, the memory requirement of v4 is best among all cases, reducing memory by -45% to -96.6% relative to Base. v4 also speeds up training time substantially, by -0.7% to -38.7% relative to Base. It also supports the theoretical complexity analysis in Section 3.3.

Overall, such nice properties in terms of high performance, efficient memory usage, and low time requirement of our proposed model are welcomed for web-scale applications.

4.5 Comparison with coarse-grained parameter allocating methods

In this section, we compare the performance with coarse-grained parameter allocating methods (CGPMs), including Star and MMoE. Specifically, we conduct two settings (see Table 6), i.e., coarse-grained (cg) and fine-grained (fg), for each method. For the former, methods adopt coarse-grained parameter modeling strategies, e.g., developing different parameters for different movie genders in MovieLens. For the latter, methods adopt fine-grained parameter modeling strategies, e.g., developing different parameters for different movies in MovieLens. The results are presented in Table 6. We can find, compared with Star and MMoE, APG shows superiority in efficiency and effectiveness. The reasons are that (1) the specific parameters of APG are dynamically generated on-the-fly, without any need to store these specific parameters, and the architecture of APG is also designed efficiently to further save memory. For Star and MMoE, they have to store these specific parameters due to the manually allocating strategies used in CGPMs. Even worse, when comes to a fine-grained setting, more parameters are maintained. (2) APG has better generalization than Star and MMoE, leading to better performance. Actually, APG adopts parameter generation manner which provides a potential opportunity for generalization, and it has the ability to imply the inherent similarity between different specific parameters (see Section 4.6 for additional experiments). Especially for the fine-grained setting where there may be a lack of enough instances to train the specific parameters, the inherent connections among different specific parameters are helpful for parameter learning.

4.6 Visualization

In this section, we visualize the generated specific parameters. Specifically, we take the group-wise strategy on Amazon as an example. The item category is set as the condition, and there is a total of 10 different categories. Then we plot the generated specific parameters (i.e., $\boldsymbol{S}_i$) into a 2-D space by PCA [33]. The visualization is presented in Figure 6. Each point refers to the specific parameters generated for one specific category. Interestingly, the observed groupings (i.e, in the same dashed circle) correspond to similar categories. This shows that the learned specific parameters by APG are meaningful and can capture the relations among different specific parameters implicitly.

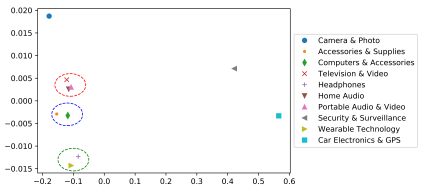


Figure 6: The visualization of the generated specific parameters.

5 Conclusion

In this paper, we propose an efficient, effective, and universal module to adaptively generate parameters for different instances. In this way, the model can carefully characterize the patterns for different instances by adopting different parameters. Experimental results show that with the help of APG, all of the existing deep CTR models can make great improvements, which also encourages a wide application for APG. Furthermore, the effectiveness and efficiency of APG are also analyzed in detail. Currently, APG requires users to set some hyper-parameters, e.g., condition strategies, K, P, and etc. In the future, we will attempt to automatically implement APG with different settings for different situations.

References

[1] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. Intrinsic dimensionality explains the effectiveness of language model fine-tuning. *arXiv preprint arXiv:2012.13255*, 2020.

[2] Liangliang Cao, Jiebo Luo, Andrew Gallagher, Xin Jin, Jiawei Han, and Thomas S Huang. Aworldwide tourism recommendation system based on geotaggedweb photos. In *ICASSP*, pages 2274–2277. IEEE, 2010.

[3] Xu Chen, Hanxiong Chen, Hongteng Xu, Yongfeng Zhang, Yixin Cao, Zheng Qin, and Hongyuan Zha. Personalized fashion recommendation with visual explanations based on multimodal attention network: Towards visually explainable recommendation. In *SIGIR*, pages 765–774, 2019.

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

[5] Xiaohan Ding, Yuchen Guo, Guiguang Ding, and Jungong Han. Acnet: Strengthening the kernel skeletons for powerful cnn via asymmetric convolution blocks. In *ICCV*, pages 1911–1920, 2019.

[6] Xiaohan Ding, Xiangyu Zhang, Jungong Han, and Guiguang Ding. Diverse branch block: Building a convolution as an inception-like unit. In *ICCV*, pages 10886–10895, 2021.

[7] Qingnan Fan, Dongdong Chen, Lu Yuan, Gang Hua, Nenghai Yu, and Baoquan Chen. Decouple learning for parameterized image operators. In *ECCV*, pages 442–458, 2018.

[8] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[9] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: a factorization-machine based neural network for ctr prediction. *arXiv preprint arXiv:1703.04247*, 2017.

[10] David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. *CoRR*, abs/1609.09106, 2016.

[11] Li He, Hongxu Chen, Dingxian Wang, Shoaib Jameel, Philip Yu, and Guandong Xu. Click-through rate prediction with multi-modal hypergraphs. In *CIKM*, pages 690–699, 2021.

[12] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. Fibinet: combining feature importance and bilinear feature interaction for click-through rate prediction. In *RecSys*, pages 169–177, 2019.

[13] Xu Jia, Bert De Brabandere, Tinne Tuytelaars, and Luc V Gool. Dynamic filter networks. *NIPS*, 29, 2016.

[14] Manas R Joglekar, Cong Li, Mei Chen, Taibai Xu, Xiaoming Wang, Jay K Adams, Pranav Khaitan, Jiahui Liu, and Quoc V Le. Neural input search for large scale recommendation models. In *KDD*, pages 2387–2397, 2020.

[15] Hoyeop Lee, Jinbae Im, Seongwon Jang, Hyunsouk Cho, and Sehee Chung. Melu: Meta-learned user preference estimator for cold-start recommendation. In *KDD*, pages 1073–1082, 2019.

[16] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*, 2018.

[17] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, pages 2736–2744, 2017.

[18] Wantong Lu, Yantao Yu, Yongzhe Chang, Zhen Wang, Chenhui Li, and Bo Yuan. A dual input-aware factorization machine for ctr prediction. In *IJCAI*, pages 3139–3145, 2020.

[19] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *KDD*, pages 1930–1939, 2018.

[20] Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, and James Henderson. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. *arXiv preprint arXiv:2106.04489*, 2021.

[21] Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch networks for multi-task learning. In *CVPR*, pages 3994–4003, 2016.

[22] Qi Pi, Guorui Zhou, Yujing Zhang, Zhe Wang, Lejian Ren, Ying Fan, Xiaoqiang Zhu, and Kun Gai. Search-based user interest modeling with lifelong sequential behavior data for click-through rate prediction. In *CIKM*, pages 2685–2692, 2020.

[23] Emmanouil Antonios Platanios, Mrinmaya Sachan, Graham Neubig, and Tom Mitchell. Contextual parameter generation for universal neural machine translation. In *EMNLP*, pages 425–435, 2018.

[24] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. Product-based neural networks for user response prediction. In *ICDM*, pages 1149–1154. IEEE, 2016.

[25] Xiang-Rong Sheng, Liqin Zhao, Guorui Zhou, Xinyao Ding, Qiang Luo, Siran Yang, Jingshan Lv, Chi Zhang, and Xiaoqiang Zhu. One model to serve all: Star topology adaptive recommender for multi-domain ctr prediction. *CIKM 2021*, 2021.

[26] Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. Towards automated neural interaction discovery for click-through rate prediction. In *KDD*, pages 945–955, 2020.

[27] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *CIKM*, pages 1161–1170, 2019.

[28] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *CIKM*, pages 1161–1170, 2019.

[29] Yi Tay, Zhe Zhao, Dara Bahri, Don Metzler, and Da-Cheng Juan. Hypergrid transformers: Towards a single model for multiple tasks. 2021.

[30] Hongwei Wang, Miao Zhao, Xing Xie, Wenjie Li, and Minyi Guo. Knowledge graph convolutional networks for recommender systems. In *WWW*, pages 3307–3313, 2019.

[31] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *ADKDD'17*, pages 1–7. 2017.

[32] Ze Wang, Zichen Miao, Jun Hu, and Qiang Qiu. Adaptive convolutions with per-pixel dynamic filter atom. In *ICCV*, pages 12302–12311, 2021.

[33] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[34] Zhihang Yuan, Bingzhe Wu, Guangyu Sun, Zheng Liang, Shiwan Zhao, and Weichen Bi. S2dnas: Transforming static cnn model for dynamic inference via neural architecture search. In *ECCV*, pages 175–192. Springer, 2020.

[35] Weinan Zhang, Jiarui Qin, Wei Guo, Ruiming Tang, and Xiuqiang He. Deep learning for click-through rate estimation. *IJCAI 2021 (Survey Track)*, 2021.

[36] Jun Zhao, Zhou Zhou, Ziyu Guan, Wei Zhao, Wei Ning, Guang Qiu, and Xiaofei He. Intentgc: a scalable graph convolution framework fusing heterogeneous information for recommendation. In *KDD*, pages 2347–2357, 2019.

[37] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *KDD*, pages 1059–1068. ACM, 2018.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 5
   (c) Did you discuss any potential negative societal impacts of your work? [No] Not applicable.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] The theories in this paper can be deduced in a straightforward manner.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] All data, instructions and hyper-parameters are explicitly written in the main paper and/or in the Suppl (see Appendix B). The code is currently proprietary.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 4.1 and Appendix B
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] But we average the results with different random seeds after running experiments multiple times
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix B
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A] We provide the URLs of the public datasets
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Examples for different condition designs

A.1 Group-wise

Example 1 (Thousand users with Thousand Models (a.k.a. Personalized Parameters)) *In a traditional recommendation model, although some personalized signals (e.g., user-item interactions) are introduced, the shared parameters cannot significantly characterize the personality of each user especially for long-tailed users. In our method, with the help of the group-wise strategy, we can simply give the prior knowledge about users to the model parameters to capture personalized patterns and achieve "Thousand users with Thousand Models". Specifically, given the embedding of a user m as $\boldsymbol{u}_i^{\boldsymbol{m}} \in \mathbb{R}^d$ and an involved instance i of this user, we can directly set $\boldsymbol{z}_i = \boldsymbol{u}_i^{\boldsymbol{m}}$. In other words, by the prior knowledge $\boldsymbol{u}_i^{\boldsymbol{m}}$, we explicitly group the instances by users and allow different users to enjoy different parameters.*

A.2 Mix-wise

Example 2 (Real-time Personalized Parameters) *In practice, user interests may be dynamically changed. For a mix-wise strategy, we can add prior knowledge about users' latest behaviors to allow the parameters sensitive to real-time interests. Specifically, given an instance i associated with a user m, we denote $\boldsymbol{z}_i^{\boldsymbol{0},\boldsymbol{m}} \in \mathbb{R}^d$ as the user m embedding and $\boldsymbol{z}_i^{\boldsymbol{1},\boldsymbol{m}} \in \mathbb{R}^d$ as the latest behaviors embedding of user m [7]. Then these two conditions $\boldsymbol{z}_i^{\boldsymbol{0},\boldsymbol{m}}, \boldsymbol{z}_i^{\boldsymbol{1},\boldsymbol{m}}$ are both considered to generate parameters. In this way, the generated parameters are specific for different users and can be adjusted in real-time by different user interests.*

Example 3 (Thousand instances with Thousand Models (a.k.a. Instance-aware Parameters) *We can consider more conditions to allow the model parameters sensitive to the instance level. To achieve this goal, the conditions are required to identify each instance. Specifically, assuming in a recommendation application, for each instance, we can take the embedding of the associated user, item, and context as $\boldsymbol{z}_i^j$.*

B The detailed experimental setting

B.1 Datasets

Four real-world datasets are used: (1)**Amazon** [8] is collected from the electronics category on Amazon. There are total 1,292,954 instances, 1,157,633 users. (2) **MovieLens.** [9] is a review dataset and is collected from the MovieLens web site. There are total 1,000,209 instances, 6,040 users. (3) **IJCAI2018 Advertising Algorithm Competition (IAAC)**[10] is a dataset collected from a sponsored search in E-commerce. Each record refers to whether a user purchases the displayed item after clicking this item. There is a total of 478,138 records, 197,694 users, and 10,075 items. (3) **Industrial dataset (IndusData)** is used for industrial evaluation (see Appendix F). Each instance refers to a user who searches a query in this platform, and the platform returns an item to this user. The label is defined as whether the user clicks this item. There are a total of 4 billion instances, and 100 million users. The statistics of the data sets are summarized in Table 7.

B.2 Baselines.

Here, we compare our method with two kinds of methods

Existing CTR prediction methods: To show the effectiveness of the proposed APG, we apply it to various existing deep CTR models (1) WDL[4] adopts wide and deep parts to memorize and generalize patterns of instances. (2) PNN[24] explicitly introduces product operation to explore the interactions of categorical data in multiple fields. (3) FIBINET[12] designs a squeeze-excitation

[7] The latest behaviors embedding can be obtained by averaging the embedding of the latest clicked items of this user

[8] https://www.amazon.com/

[9] https://grouplens.org/datasets/movielens/

[10] https://tianchi.aliyun.com/competition/entrance/231647/introduction

Table 7: The statistic of datasets.

	#Data	#User ID	#Item ID
Amazon	1,292,954	1,157,633	9,560
MovieLen	1,000,209	6,040	3,706
IAAC	478,138	197,694	10,075
IndusData	4 billion	100 million	80 million

Table 8: The AUC (%) results of APG with different kinds of condition strategy, including Group-wise, Mix-wise, and Self-wise. Note Base refers to the results of the method without APG. U, I, and C refers to the embedding of users, items, and contexts respectively.

Strategy	z_i	MovieLens	Amazon	IAAC
Base		79.21	69.15	65.17
Group-wise	U	79.61	69.28	65.80
	I	79.58	69.35	65.76
Mix-wise	U,I,C	79.45	69.31	65.90
Self-wise	x_i	**79.73**	**69.43**	**65.94**

network to dynamically learn the feature importance and use a bilinear-interaction layer to learn the interactions among features. (4) DIFM[18] brings the idea of the transformer and learns vector-wise and bit-wise interactions among features. (5) DeepFM[9] takes the linear part of WDL with an FM network to better represent low-order features. (6) DCN[31] learns low-order and high-order features simultaneously and needs low computation cost. (7) AutoInt[27] learns the feature interactions automatically vim self-attention neural networks.

Coarse-grained parameter allocating methods: We also try to compare the proposed APG with the coarse-grained parameter allocating methods: (1) Multi-task learning: MMoE [19] keeps multiple parameters by adopting multiple network branches for different tasks; (2) Multi-domain learning: Star [25] allocates multiple parameters for different scenarios.

B.3 Training Details

The embedding dimension is set 32 for all methods. The number and units of hidden layers are set $\{256, 128, 64\}$ for all methods by default. Other hyper-parameters of different methods are set by the suggestion from original papers. The backbone of the deep CTR models is set as WDL by default. For APG, we set the self-wise strategy as the default condition strategy. $\mathcal{G}$ is implemented as an MLP with a single layer by default. We set the hyper-parameters $K \in \{2, 4, 6, 8\}$ and $P \in \{32, 64, 128, 256, 512\}$ and perform grid search over K and P. We include the results for different values of K and P in Appendix E. We use the Adam optimizer with a learning rate of 0.005 for all methods. The batch size is 1024 for all datasets. Each dataset is randomly split into 80% train, 10% validation, and 10% test sets. For the public datasets, methods are trained on a single V100S GPU. For the industrial dataset, methods are trained in an internal cluster equipped with V100S GPU and SkyLake CPU. We run all experiments multiple times with different random seeds and report the average results.

C Evaluation of Condition Design

In this section, we analyze the influence of the condition design. Specifically, we evaluate different condition strategies on CTR prediction tasks and report the AUC results. We take WDL as the backbone of APG. For the mix-wise strategy, the input aggregation with attention function is used and the effect of the aggregation method of the mix-wise strategy is detailedly analyzed in Appendix D. Besides we also provide the results of WDL (defined as Base) for comparison.

The effect of different condition strategies. We first analyze the effect of different condition strategies, including the group-wise, the mix-wise, and the self-wise strategies. From Table 8, we can find that (1) Compared with Base, all of these condition strategies can obtain better performance. It indicates the effectiveness of APG which can dynamically generate the parameters for better pattern learning; (2) The self-wise strategy achieves the best performance among other strategies in all cases. One of the possible reasons is that the prior knowledge of the self-wise strategy is directly from the hidden layers' input which is a more immediate signal for the current layer and may lead to better parameter generation.

The effect of different prior knowledge in the same strategy. We also evaluate the effect of different prior knowledge in the same strategy. Specifically, considering the group-wise strategy, we compare the performance between the prior knowledge about the user embedding and the item embedding. The results are reported in Table 8. We can conclude that (1) Different prior knowledge in the same strategy can get competitive performance; (2) Designing a proper prior knowledge may achieve better performance. For example, for group-wise strategy, taking the item embedding as the prior knowledge obtains higher AUC on Amazon. While the prior knowledge about the user embedding performs better on IAAC and MovieLens.

D Evaluation of the Different Aggregation Functions

In this section, we conduct experiments to evaluate the performance when using different aggregation functions for the mix-wise strategy. All of the cases use the embedding of users, items, and the context as the condition. The results are reported in Table 9. We can observe that (1) Compared with other functions, the Attention function achieves the best performance in most cases due to the high representation power of the attention function; (2) Compared with Output Aggregation, Input Aggregation performers better. One of the possible reasons is that the relations among different conditions can be implicitly modeled through $\mathcal{G}$ while output aggregation simply summarizes different specific parameters.

Table 9: The results of using different aggratetion function for the mix-wise strategy.

AUC (%)	**Function**	**MovieLens**	**Amazon**	**IAAC**
Base		79.21	69.15	65.17
Input Aggregation	Mean	79.33	69.28	65.44
	Concat	79.38	**69.36**	65.73
	Attention	**79.45**	69.31	**65.90**
Output Aggregation	Mean	79.30	69.34	65.44
	Concat	79.33	**69.36**	65.57
	Attention	79.41	69.25	65.70

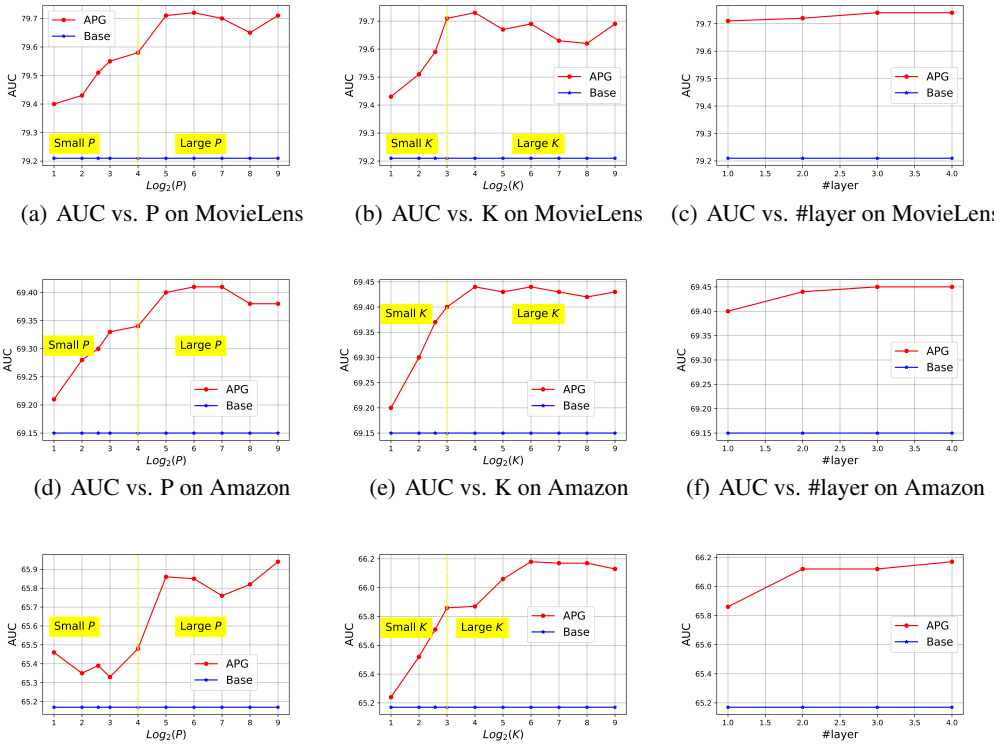
E Evaluation of Hyper-parameters

In this section, we conduct experiments to analyze the effect of the hyper-parameters including P, K, and the number of MLP layers in APG.

E.1 The effect of different P

The hyper-parameter P is introduced to add more shared parameters. Thus P is required to be much larger than K (i.e., $P \gg K$). Here we keep $K = 8$ for all cases, and set P to $\{32, 64, 128, 256, 512\}$ respectively and evaluate the performance of APG. The results are plotted in the right part (i.e., Large P) in Figure 7 (a)(d)(g). Note we also provide the results of the Base for comparison. We can find that setting different large values of P can give a similar better performance compared with Base. It indicates that when adding more parameters by setting a large P, the model can be stably improved.

Does a small P works? We further manually set P as a small value (e.g., $\{2, 4, 6, 8, 16\}$). From the left part (i.e., Small P) in Figure 7 (a)(d)(g), we can find, that although APG still performs better than Base, there exists performance gap between Large P and Small P. It shows the importance of introducing sufficient shared parameters by over parameterization.



(a) AUC vs. P on MovieLens (b) AUC vs. K on MovieLens (c) AUC vs. #layer on MovieLens

(d) AUC vs. P on Amazon (e) AUC vs. K on Amazon (f) AUC vs. #layer on Amazon

(g) AUC vs. P on IAAC (h) AUC vs. K on IAAC (i) AUC vs. #layers on IAAC

Figure 7: Evaluation of the hyper-parameters.

E.2 The effect of different K

In this part, we evaluate the effect of different K. Specifically, we set $P = 32$ and the hidden layers of the deep CTR model as $\{256, 128, 64\}$ for all cases. Since K is required to be much smaller than $min(N, M)$, we firstly set K to $\{2, 4, 6, 8\}$ respectively. The results are reported in the left part (i.e., Small K) in Figure 7 (b)(e)(h). Note we also provide the results of the Base for comparison. We can find that (1) APG performs better than Base in all cases which shows the effectiveness of APG; (2) With the increasing of K, APG can also achieve better performance. It indicates that it is important to give sufficient specific parameters to characterize the custom patterns of different instances.

Does a large K helps? Similar to the purpose of over parameterization, ignoring the heavy cost of storage and computation, we can also set K to a large value (e.g., $\{16, 32, 64, 128, 256, 512\}$) to see whether the model can obtain further improvements. The results are presented in the right part (i.e., Large K) in Figure 7 (b)(e)(h). Some observations are summarized as follows: (1) When we set a large value of K, compared with the small one, APG can further improve the model performance in most cases. It indicates increasing K to a large value does help to model the custom patterns; (2) When K is set to extremely large values (e.g., 256 or 512), APG only achieves similar performance with the cases where $K \in \{16, 32, 64, 128\}$; It shows enlarge the specific parameters does not always give a positive contribution and it is wiser to set a suitable value of K.

E.3 The effect of the different number of layers

Here, we increase the number of the MLP layers in APG to evaluate the performance. We also report the results of Base for comparison. The results are depicted in Figure 7 (c)(f)(i). Some observations are summarized as follows (1) Compared with Base, APG with a different number of layers can always achieve significant improvement; (2) Compared with the performance of a single layer, increasing the number of layers can make a slight improvement.

Table 10: The results of the severing efficiency. v1 refers to the basic model of APG.

	Base	**APG**	**v1**
RT(ms)	14.5	14.8	57.1
PVR(%)	0	-0.01	-33.2
Memory (M)	138.03	16.14	4812.7

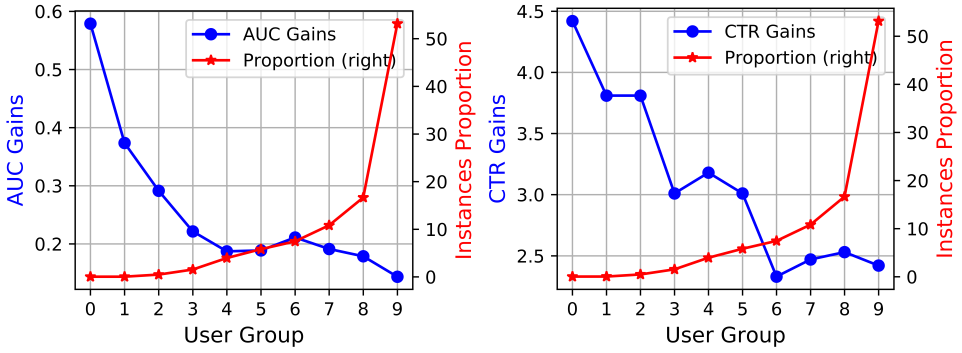


Figure 8: The AUC (**Left**) and CTR (**Right**) gains in different user groups.

F Performance in Industrial Sponsored Search System

Here, we show the performance of APG on industrial applications. Specifically, we firstly train APG with the industrial dataset and then develop it in the sponsored search system. Since December 2021, APG is developed and served as the main traffic of our system.

Overall Gains. Compared with the online model, it achieves 0.2% gains in AUC. During the online A/B test, we observe a 3% CTR gain and 1% RPM (Revenue Per Mile) gain respectively. Note this is a significant improvement in the industrial sponsored search system.

Severing Efficiency. We further evaluate the severing efficiency of APG. The average Response Time (RT) and Page View Rate (PVR) of the model inference online are evaluated when a user search queries. We also present the memory cost of the hidden layers in each model. Not PVR is influenced by the request timeout. The results are reported in Table 10. Considering the memory efficiency, the memory cost of APG is 8$\times$ smaller than Base. Considering the time efficiency, APG does not achieve much improvement and has similar RT and PVR compared to Base. The reason is that, in online serving, the inference time may not have a direct positive relation with GFlops (or theoretical complexity) since the calculation is not always the bottleneck due to the powerful distributed environment and other factors (e.g., I/O, cpu-gpu communication and etc.) also have a great impact on the inference time. In addition, compared with v1, APG achieves a significant improvement in time cost and memory usage, which demonstrates the efficiency of the proposed extensions in APG.

The effect on low-frequency users. As described in Introduction Section, adding the specific parameters can capture the custom patterns for different instances, especially for the long-tailed instances, since without the specific parameters the model may be easily dominated by the hot instances. Thus, we detailedly analyze the impact on the different instances when considering the specific parameters. Specifically, we divide the users into 10 groups by their frequency. The frequency is increased from group 0 to group 9 and the number of users is set the same in different groups. Then we evaluate the improvement (including AUC and CTR) of different groups respectively. The results are reported in Figure 8. We can find that (1) Since group 9 refers to the users with the highest frequency, although it has only 10% users, this group produces more than 50% instances (see the red line in Figure 8.); (2) The specific parameters contribute more for low-frequency users since it achieves more gains of AUC and CTR in low-frequency users (e.g., group 0). It demonstrates that specific parameters do allow low-frequency instances to better represent their features, leading to better performance.