Personal Recommendation Using Deep Recurrent Neural Networks in NetEase

Sai Wu $^{\#1}$, Weichao Ren $^{\#2}$, Chengchao Yu $^{\#3}$, Gang Chen $^{\#4}$, Dongxiang Zhang $^{\triangleleft 5}$, Jingbo Zhu $^{\diamond 6}$

$^{\#}$ *College of Computer Science and Technology, Zhejiang University, Hangzhou, China*
^{1,2,3,4} {wusai, weichaor, 21311200, cg}@zju.edu.cn

$^{\triangleleft}$ *School of Computer Science and Engineering, University of Electronic Science and Technology of China*
^{5} zhangdongxiang37@gmail.com

$^{\diamond}$ *NetEase (Hangzhou) Network Co., Ltd., Hangzhou, China*
^{6} zhujingbo@corp.netease.com

***Abstract*—Each user session in an e-commerce system can be modeled as a sequence of web pages, indicating how the user interacts with the system and makes his/her purchase. A typical recommendation approach, e.g., Collaborative Filtering, generates its results at the beginning of each session, listing the most likely purchased items. However, such approach fails to exploit current viewing history of the user and hence, is unable to provide a real-time customized recommendation service. In this paper, we build a deep recurrent neural network to address the problem. The network tracks how users browse the website using multiple hidden layers. Each hidden layer models how the combinations of webpages are accessed and in what order. To reduce the processing cost, the network only records a finite number of states, while the old states collapse into a single history state. Our model refreshes the recommendation result each time when user opens a new web page. As user's session continues, the recommendation result is gradually refined. Furthermore, we integrate the recurrent neural network with a Feedfoward network which represents the user-item correlations to increase the prediction accuracy. Our approach has been applied to Kaola (http://www.kaola.com), an e-commerce website powered by the NetEase technologies. It shows a significant improvement over previous recommendation service.**

I. Introduction

Collaborative Filtering (CF) recommendation [20][13] is widely used in e-commerce system which, based on the correlations between users and items, predicts the probability of a user buying a specific item. Its assumption is that users sharing the similar purchase history are likely to buy the same set of items. Although CF approach works well in some cases, we find that it is unable to provide a more accurate real-time recommendation, because its model is established using stale data, lacking customization options.

Figure 1 illustrates the scenario. When a user first enters the system, we can collect his/her basic information, such as which web browser he/she is using, his/her IP address or whether he/she is using a mobile phone to access our system. If this is a registered user, we can get a full profile about the user including gender, age, interest and his/her purchase history for the CF algorithm.

Besides those basic attributes, we also have a special type of dynamic attributes, e.g., the browsing history of current session, which update their values throughout the session. Before making his/her purchase, a user will view a long list of web pages (In Kaola, a user views 12 pages averagely before adding one item to the cart). This viewing history actually records how users interact with the system and perform their purchases, while previous CF recommendation approaches fail to exploit it. In Figure 1, the first page of the session is “index.html". As no specific information can be retrieved, we will return our recommendations using the CF algorithm as the other systems. Suppose the user is searching for men's wallets and kids' shoes. The next page to be opened is “men.html" and correspondingly, we may update our recommendations as Levis' jeans and Diesel's shirts. This is definitely not a correct recommendation, but it is the best guess given current viewing history. After the user opens pages “list.html?cat=101" (directory for men's wallets) and “2915823401.html? (boy's clog), we update the recommendations as wallets and crocs clog, which perfectly hit the hot spot. In other words, a more accurate real-time recommendation is required, which considers not only the purchase history of users, but also current browsing history and can adjust its recommended results during the session.

More specifically, based on current viewing history and the user's interest, we should provide a recommendation result which will be updated continuously by guessing what the user really wants to buy. This is a challenging task as:

- There are thousands of web pages in Kaola. If each web page represents a state, the input to our prediction model is a huge vector denoting which pages have been accessed. Moreover, each user session can be considered as a combination of a random number of states, making the learning job more complicated.
- As shown in Figure 1, pages are accessed in some specific orders, e.g., from a category page to a product page. When the user opens a new page, we will update our recommendations and use a widget in the page to show the new results. This requires our approach to be sensitive to the access order and efficient enough to generate real-time results for possibly millions of current user sessions.
- Finally, we can use the access log as our training set, since it tracks how users interacted with our system,

The work of Weichao Ren and Chengchao Yu was done in Netease. Gang Chen is the contract author.

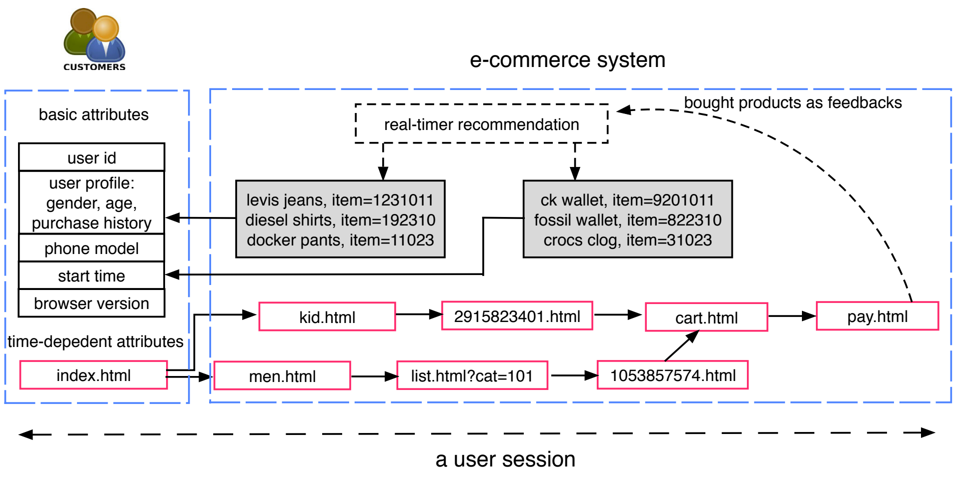


Fig. 1. Real-time Recommendations in E-commerce System

and it also has a record of which items the user finally bought when his/her session ended. However, the training set is keeping updated, because as long as the system is online, we can obtain new logs of more user sessions which can be further used to refine our model. In other words, our model is not the "train-once" model. It should be updated in real-time to reflect the new purchase patterns of users.

To address the above challenges, in this paper, we use a deep recurrent neural network (DRNN) to model user's browsing patterns and provide a real-time recommendation service. The DRNN consists of multiple hidden layers with temporal feedback loops in each layer. The bottom layer denotes the accessed web pages and the inner layer represents the combinations of pages. Our DRNN is similar to the one studied in [8]. However, there are three main differences. First, our DRNN is built to track the user's access patterns. The sequence of pages shows the path that leads a user to his/her desired product. The intuition of our DRNN model is to return a real-time recommendation to narrow down the gap between the user and his/her destination.

Second, conventional unrolled RNN is designed as a sliding window based approach which, at most, maintains a finite number of states. As the learning process continues, the old states will be replaced by new ones. However, it is not a trivial task to decide a proper window size. A larger window causes too much computation overhead, while a smaller window may affect the accuracy of prediction. In our DRNN, we summarize past states into a virtual history state. The history state works with current active states to generate the final prediction.

Third, we also build a feedforwarding neural network (FNN) to simulate the CF algorithm. Our FNN accepts the user's purchase history vector as input and generates the prediction for the probability of buying an item. FNN is forged with DRNN to produce the final results, while how the two networks are combined is learned from our training data.

The DRNN model has been implemented as a recommendation module of Kaola system. The module streams the real-time user log to a document database which reorganizes the log records into sessions and feeds each session to our DRNN model. The DRNN model updates the neural network accordingly. On the other hand, Kaola forwards the user's request of a web page to the DRNN model which will generate a list of new recommended items. Kaola then returns the web page which displays the new recommendation results in a customized widget.

To achieve a better performance, the DRNN model is tuned extensively. This is a tedious work, and in this paper, we propose an automatic tuning framework to optimize our DRNN model. It applies a genetic algorithm to improve the prediction accuracy of the model, which significantly reduces the overhead of deploying the DRNN for a real application.

Our experiments on the Kaola testing system show that the DRNN model achieves an order of magnitude better result than previous recommendation technique such as CF. The remaining of the paper is organized as follows. Section II gives an overview of the architecture of the recommendation module. Section III discusses the details of our DRNN model and its implementation details. Section IV introduces our automatic tuning framework. Section V presents the experiment results. We briefly review some related work in Section VI and conclude the paper in Section VII.

II. Overview of Recommendation Module

Personalized recommendation is a key feature in the e-commerce system (e.g., Kaola) to improve the user's experience. Previous in Kaola, we collected users' purchase history and applied the CF algorithm to generate a recommendation result for each user in an offline process. When a user logged in, we pushed the recommendation results to him/her. To catch up with new purchase trends, the CF algorithm was invoked periodically to update recommendation results using new log data. Unfortunately, the precision (the probability that a user finally buys a recommended item) is very low (see our experiment results), and the offline approach cannot discover the latest purchase pattern (e.g., a discount item attracts a sudden burst of sales).

To address the two issues, we develop a new recommenda-

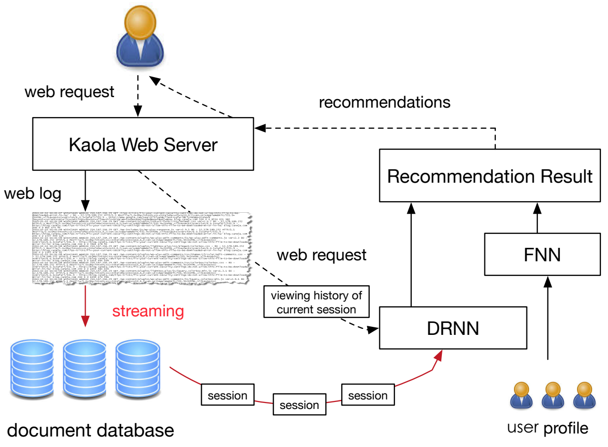


Fig. 2. Work Flow of Recommendation Module

tion module powered by our DRNN model to replace the old one. Figure 2 shows its architecture. User's request to a web page is sent to the Kaola web server, which is accumulated together to form a viewing session. The viewing session is used as the input for the DRNN model to generate a real-time prediction. DRNN model collaborates with a FNN model, which is used to simulate the CF algorithm. In other words, we consider both user's interest of current session and his/her past interest. Finally, Kaola server returns the requested web page to the user by demonstrating the recommendation results in a specific web widget. As the viewing session continues (more pages have been viewed), we gradually refine our prediction results. Users are expected to find their items from the recommendation results with a higher probability.

On the other hand, after a session completes, we actually get a ground truth result for our prediction (whether the user buys our recommended item). We can adjust our model using the new training sample. In particular, the http access log is streamed to a document database. Each log entry is saved as a JSON document. We create indexes on user ID, session ID and timestamp for each document. For a specific user u_i, we generate its session documents as follows:

- Let s_0 be current session ID of u_i. If we find a new log of u_i with session ID s_1 ($s_1 \neq s_0$), we group all JSON documents of s_0 into a session document.
- Let t_0 be the timestamp of the last log of u_i. If no new log document is received for u_i until t_0+T, we group all JSON documents of s_0 into a session document. T is a predefined a timeout threshold (e.g., 30 minutes in Kaola).

One session document consists of multiple log documents. So we have $D^s_i = \{D^l_0, D^l_1, ..., D^l_{k-1}\}$ where D^s_i and D^l_j represent a session document and log document respectively. The log documents are sorted by their timestamps. So we have $D^l_{j-1}.timestamp \leq D^l_j.timestamp$. Because each log document records the user's request to a specific URL, we can simplify D^s_i as $\{p_0, p_1, ..., p_{n-1}\}$, where p_j is the URL recorded in D^l_j. Session documents are used as the input for the DRNN model to adjust the weights and bias values of the neural network. To reduce the learning cost, DRNN starts its learning process only when we have a batch of M new sessions. M is a tunable parameter, configuring how fast the model can detect new purchase patterns.

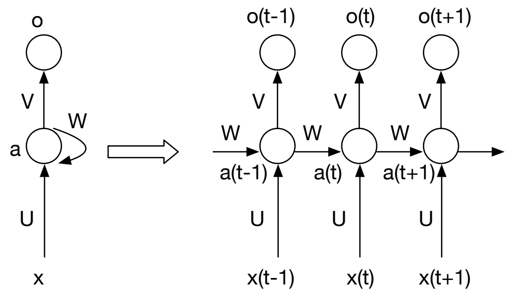


Fig. 3. Recurrent Neural Network

III. Deep Recurrent Neural Network

The idea of using RNN for real-time recommendation in our scenario is intuitive, because each user's session can be modeled as a series of web page visits and RNN is good for predicting temporal variables. RNN with one hidden layer can catch the order of page accesses, while RNN with multiple hidden layers can also discover the combinations of page accesses. In this section, we present our DRNN model and its implementation details.

A. Basic RNN Model

RNN (Recurrent Neural Network) is a neural network designed to process the sequential information. In this paper, we consider the discrete RNN, where the process is split into multiple states and each state is tagged with a timestamp. Figure 3 shows the basic idea of RNN. Let x and o represent the input and output respectively. We use a to denote the values in the hidden layer. We also have three transit matrixes U, V and W. The hidden layer has a self-link indicating that it will continuously update its values as time elapses.

For better illustration, the RNN can be unfolded into a full network as shown in Figure 3. Suppose we have three states at time $t-1$, t and $t+1$. $x(i)$ and $o(i)$ $(t-1 \leq t \leq t+1)$ denote the input and output at different states respectively. $a(i)$, the values of hidden layer at state i, will be updated based on $a(i-1)$, namely the values from previous state. Formally, we have:

$$a(i) = f(Ux(i) + Wa(i-1))$$

where f is a non-linear learning function such as tanh, ReLU and sigmoid. $o(i)$ is the prediction at state i, computed as:

$$o(i) = softmax(Va(i))$$

So we can get an output at every state.

Note that after we unfold the network, we still use the same transit matrixes U, V and W. In other words, we believe that the RNN performs the same set of computations at different states. This also significantly reduces the computation overhead of the RNN.

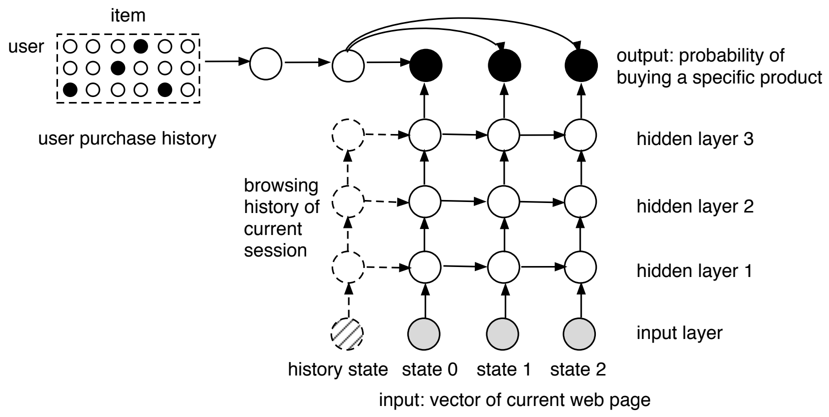

item
user
user purchase history
output: probability of buying a specific product
hidden layer 3
browsing history of current session
hidden layer 2
hidden layer 1
input layer
history state
state 0
state 1
state 2
input: vector of current web page

Fig. 4. Illustration of Deep RNN model

B. Deep RNN Model with Infinite States

In Figure 3, we show a RNN with a single hidden layer, while in real applications, we can build more hidden layers to improve the prediction accuracy.

In our scenario, the input is a sequence of web pages $\{p_0, p_1, ..., p_{n-1}\}$ denoting a specific user session, where user accesses p_t before p_{t+1}. We consider each web page as a state of user session. The RNN is turned into a state machine of the e-commerce system, simulating how users interact with the system. Suppose we have unlimited memory, so all states can be modeled by the RNN.

The input layer of our RNN consists of n states. Each state refers to an accessed web page. More specifically, for page p_t, we generate an M-element vector V_t (M is the number of web pages). $V_t[p_t]$ is set to 1 and all the other elements are set to 0. Vector V_t is used as the input at the $t+1$th state in the input layer.

Each state of RNN includes L hidden layers and each layer has E neurons. Suppose we have M web pages and N items in the e-commerce system. E will be set between M and N, namely $N << E << M$. Note that it is possible to tune the number of neurons individually for each layer (see Section IV for details). To simply the representation, we first assume that all layers are using the same number of neurons. Inside a state, neurons in the ith layer connect to the neurons in the $i+1$th layer. Between two consecutive states, neurons in the ith layer of state t connects to the neurons in the ith layer of state $t+1$.

Finally, the output layer is a probability vector $V_{out} = \{v_0, v_1, ..., v_{N-1}\}$. v_i represents the probability of purchasing the ith item. E.g., the ith item will be purchased in current session with a probability of v_i. The last hidden layers will connect to the output layer in each state. So we can return a real-time prediction when a new state is accepted by the system. Normally, the recommendation service will return k items with maximally probabilities to users as our prediction.

Let $a_i(t)$ denote the state of the ith layer at state t. Its update function is set as:

$$a_i(t) = \begin{cases} f(W_i a_i(t-1) + Z_i(a_{i-1}(t) + b_i(t))) & \text{if } i > 1 \\ f(W_i a_i(t-1) + Z_i(V_t + \theta(p_t))) & \text{if } i = 1 \end{cases}$$

In the above function, W_i represents the weights of connections from state $t-1$ and Z_i denotes the weights of connections from the lower hidden layer or input layer at the same state. $b_i(t)$ is the bias and note that for input layer, we use $\theta(p_t)$ (the time that user stays at page p_t) as the bias.

We illustrate the network in Figure 4. We have three states and three hidden layers. The bottom layer captures the sequence of web pages that users access. The next layer models how two consecutive web pages are accessed. In this way, the kth layer actually summarizes how $k-1$ web pages can be accessed in an arbitrary sequence.

C. Deep RNN Model with History States

In real systems, we do not have enough memory to maintain all possible states. Instead, previous RNN only models a finite number of states. For example, the network in Figure 4 can accept three continuous states. If a new state is received, the old one will be discarded from the model. This limitation affects both the training process and prediction process. Suppose only n states are supported. During the training process, if a user session involves more than n pages, we only keep the last n pages as our input. On the other hand, in the prediction process, we maintain a sliding window of n states. When the user opens a new page, it will be added into the window and if the window is already full, the oldest page will be removed. Only pages inside the window will be used as input for the network to generate a prediction result. This may dramatically reduce our prediction accuracy, because the viewing history shows a complete path of how users reach their items.

However, it is challenging to set a proper n. A larger n definitely increases the computation overhead in both training and prediction process. Based on the observation from the user log of Kaola system, we find that some user may browse more than 50 different pages before making a decision, indicating that n should be large enough to provide an accurate result.

To address the problem, we introduce a history state which collapses all past states into one. Let $\{p_0, p_1, ..., p_x\}$ represent all pages that the user has accessed in his/her session. If $x \leq n$, history state is not used. Otherwise, the input for the history

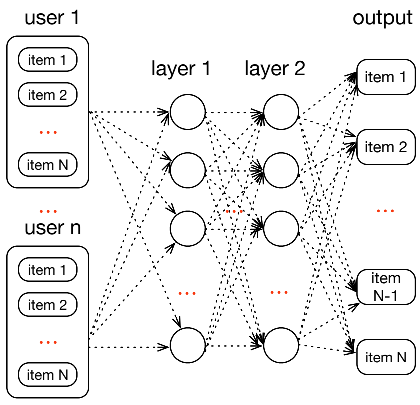


Fig. 5. FNN for CF

state is computed as:

$$\bar{V} = \sum_{i=0}^{x-n} \epsilon_i V_i$$

V_i is the vector for page p_i and ϵ_i is the aging factor for old states, which is defined as:

$$\epsilon_i = \frac{\theta(p_i)}{\sum_{j=i}^{x-n} \theta(p_j)}$$

For the history state, we use the same update function as the normal state for its hidden layers. History state does not produce any output. Instead, it affects the decision of other states as shown in Figure 4. Suppose we have accessed x pages. The first state in the RNN maintains the information about page p_{x-n+1}. The state also accepts recurrent connections from the history state. Note that history state is just an approximate representation of the pass states to balance the trade-off between computation overhead and accuracy.

D. Collaborations with Collaborative Filtering

CF (Collaborative Filtering) approach shows its effectiveness in many real applications. It catches the correlations between users and items, revealing the common interests of users. Users sharing the same interests are likely to buy the same set of items. Our RNN model is a good supplement for the CF approach. CF approach generates a good recommendation if users follow their old purchase patterns, while RNN model can effectively predict the unexpected purchase of a specific user. So in this paper, we build a forward-feedback neural network (FNN) to simulate the CF approach and link it with our RNN. Previous work [19] also adopted the FNN to build the CF model. In this paper, we simplify its learning process.

Figure 5 illustrates our FNN. In the input layer, we create U neurons, one for each user. Its input is a 0-1 bitmap, indicating whether the user has bought an item or not. The whole input layer is actually a matrix representing users' purchase history, simulating the input of CF algorithm.

For simplicity, all users share the same set of neurons in the hidden layers, which also helps us identify the common purchase patterns of users. Suppose each hidden layer has $\bar{E}$ neurons. For the jth neuron at the ith hidden layer, its state $\bar{a}_j^{(i)}$ is updated as:

$$\bar{a}_j^{(i)} = f(\sum_{x=0}^{\bar{E}-1} w_j^{(i-1)} \bar{a}_x^{(i-1)} + b_x^{(i-1)})$$

where $w_j^{(i-1)}$ is the weight to this neuron from the last layer and $b_x^{(i-1)}$ is the corresponding bias.

The output layer of FNN contains N neurons. Each neuron represents a specific item and its output value is the probability that current user selects the item for purchase.

We observe that the FNN shares the same output layer as the RNN. So we will merge their output layers to produce a unified result. Suppose we have L_0 layers of RNN and L_1 layers of FNN. For the prediction at state t, the probability of buying an item i ($P(i)$) is computed as:

$$\frac{f(\sum_{x=0}^{E-1}(w_i^{L_0} a_{L_0}(t) + b_{L_0}(t)) + \sum_{x=0}^{\bar{E}-1}(\bar{w}_i^{L_1} \bar{a}_x^{(L_1)} + b_x^{(L_1)}))}{\sum_x f(\sum_{x=0}^{E-1}(w_i^{L_0} a_{L_0}(t) + b_{L_0}(t)) + \sum_{x=0}^{\bar{E}-1}(\bar{w}_i^{L_1} \bar{a}_x^{(L_1)} + b_x^{(L_1)}))}$$

$P(i)$ estimates the probability of buying the ith item at state t. We use $w_i^{L_0}$ and $\bar{w}_i^{L_1}$ to represent the weights from the last hidden layers of RNN and FNN respectively. As the probability should be less than 1, we normalize it using the weights of all items.

Note that instead of predefining some parameters for RNN and FNN to merge their results, how RNN and FNN are combined is tuned by the weights which will be trained using SGD (Stochastic Gradient Descent). In this way, we do not need to explicitly specify which model (CF model and RNN model) is more important.

E. Implementation Details

1) Generating Training Data: For a user session $S = \{p_0, p_1, ..., p_{n-1}\}$, where p_i denotes the ith page that the user accesses, we can generate a set of training samples for our DRNN model. In particular, let I be the vector representing the items purchased in session S. I is used as the ground truth to perform the SGD training process. If our DRNN model can maintain k states, we generate a training sample for S as:

$$h, p_{n-k}, ..., p_{n-1} \rightarrow I$$

h is the accumulative history state computed using p_0,..., p_{n-k-1}. If $n-1 < k$, we do not have enough states. The training sample is represented as:

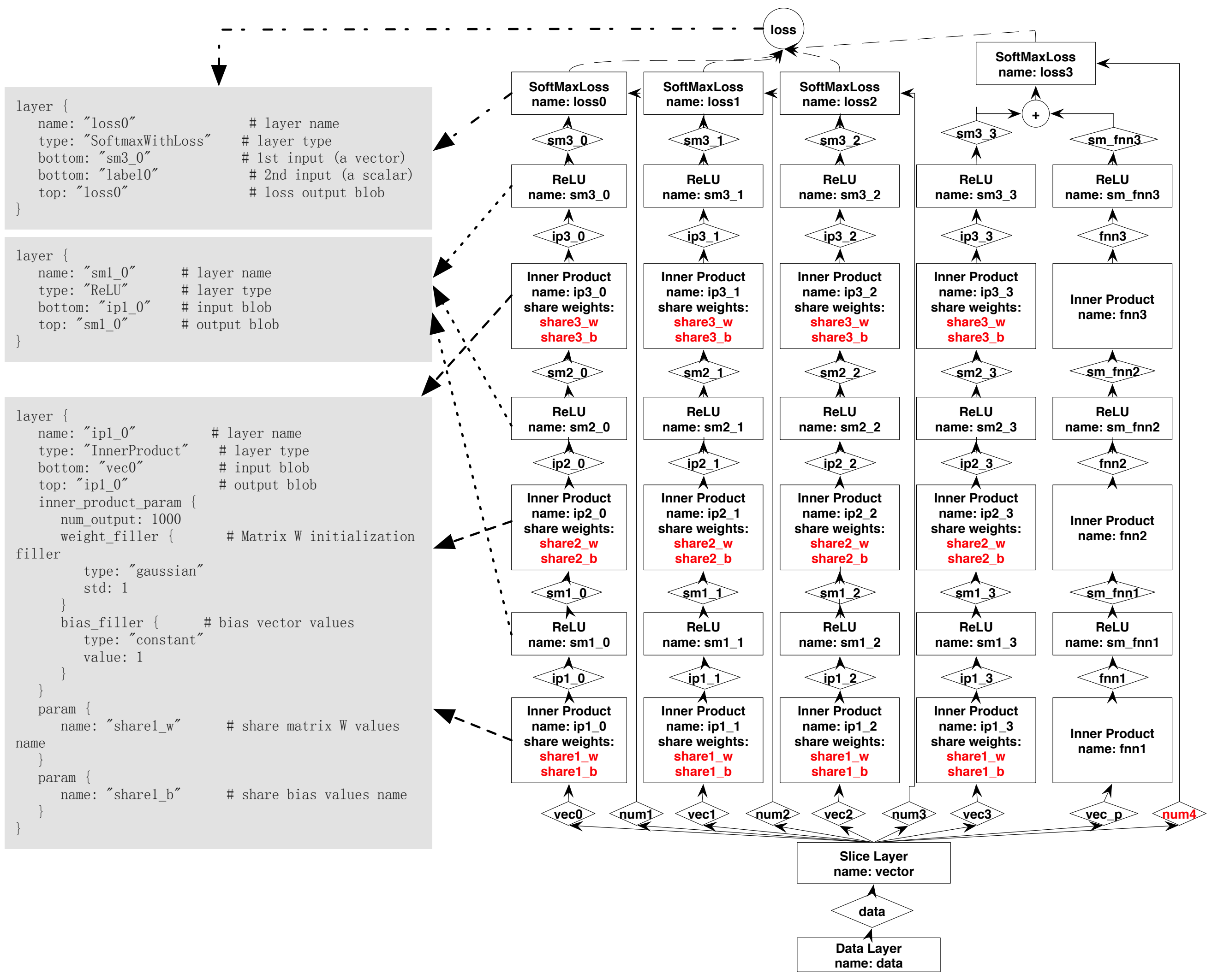
$$p_0, ..., p_{n-1} \rightarrow I$$

However, remember that our goal is to reduce the number of pages that users have to access before making their final purchase. The ideal case should be

$$p_0 \rightarrow I$$

Namely, after viewing the first page, the user has already decided what to buy. We can also have a set of possibly better rules than the real case, such as:

$$p_0, p_1 \rightarrow I$$


```
layer {
   name: "loss0"               # layer name
   type: "SoftmaxWithLoss"    # layer type
   bottom: "sm3_0"            # 1st input (a vector)
   bottom: "label0"            # 2nd input (a scalar)
   top: "loss0"                # loss output blob
}
```

```
layer {
   name: "sm1_0"      # layer name
   type: "ReLU"       # layer type
   bottom: "ip1_0"    # input blob
   top: "sm1_0"       # output blob
}
```

```
layer {
   name: "ip1_0"          # layer name
   type: "InnerProduct"    # layer type
   bottom: "vec0"          # input blob
   top: "ip1_0"            # output blob
   inner_product_param {
      num_output: 1000
      weight_filler {       # Matrix W initialization
filler
         type: "gaussian"
         std: 1
      }
      bias_filler {      # bias vector values
         type: "constant"
         value: 1
      }
   }
   param {
      name: "share1_w"      # share matrix W values
name
   }
   param {
      name: "share1_b"      # share bias values name
   }
}
```

loss
SoftMaxLoss name: loss0
SoftMaxLoss name: loss1
SoftMaxLoss name: loss2
SoftMaxLoss name: loss3
+
sm3_0
sm3_1
sm3_2
sm3_3
sm_fnn3
ReLU name: sm3_0
ReLU name: sm3_1
ReLU name: sm3_2
ReLU name: sm3_3
ReLU name: sm_fnn3
ip3_0
ip3_1
ip3_2
ip3_3
fnn3
Inner Product name: ip3_0 share weights: share3_w share3_b
Inner Product name: ip3_1 share weights: share3_w share3_b
Inner Product name: ip3_2 share weights: share3_w share3_b
Inner Product name: ip3_3 share weights: share3_w share3_b
Inner Product name: fnn3
sm2_0
sm2_1
sm2_2
sm2_3
sm_fnn2
ReLU name: sm2_0
ReLU name: sm2_1
ReLU name: sm2_2
ReLU name: sm2_3
ReLU name: sm_fnn2
ip2_0
ip2_1
ip2_2
ip2_3
fnn2
Inner Product name: ip2_0 share weights: share2_w share2_b
Inner Product name: ip2_1 share weights: share2_w share2_b
Inner Product name: ip2_2 share weights: share2_w share2_b
Inner Product name: ip2_3 share weights: share2_w share2_b
Inner Product name: fnn2
sm1_0
sm1_1
sm1_2
sm1_3
sm_fnn1
ReLU name: sm1_0
ReLU name: sm1_1
ReLU name: sm1_2
ReLU name: sm1_3
ReLU name: sm_fnn1
ip1_0
ip1_1
ip1_2
ip1_3
fnn1
Inner Product name: ip1_0 share weights: share1_w share1_b
Inner Product name: ip1_1 share weights: share1_w share1_b
Inner Product name: ip1_2 share weights: share1_w share1_b
Inner Product name: ip1_3 share weights: share1_w share1_b
Inner Product name: fnn1
vec0
num1
vec1
num2
vec2
num3
vec3
vec_p
num4
Slice Layer name: vector
data
Data Layer name: data

Fig. 6. Implementation of Recurrent Neural Network

$$h, p_{n-k-1}, ..., p_{n-2} \rightarrow I$$

Generally, we can have n rules as training samples, which are used as input for our DRNN. In other words, we intentionally shorten the access path of users by guessing which items they will buy aggressively and making it as a rule.

As mentioned before, a session consists of multiple access log documents. We merge those documents to produce training samples for a specific user. The samples are then buffered in levelDB[1], a key-value store developed by Google, by using the user's ID as the key. After they have been fed to the neural network, we discard them from the database.

2) DRNN Implementations: We implement our RNN model on top of Caffe [9], an open source deep learning framework. We use Caffe 1.0 which does not support RNN explicitly. So we transform the normal convolutional network into a RNN by sharing the weights among different network layers. Figure 6 shows our network structure and the corresponding layer definition in Caffe. For simplicity, layers of the history states are not shown.

The bottom layer is the data layer retrieving training samples from levelDB. We formalize the training samples into two vectors, representing the accessed web page (vec) and ground truth (num) result respectively. The first vector is used as the input for the neural network, while the second one is used in the SGD process. The network in Figure 6 consists of one RNN(the first four columns) and one FNN(the last column). Both RNN and FNN use three inner product layers. The RNN model can maintain k states ($k = 4$ in the figure), where k is a tunable parameter.

Each inner product layer is actually translated into three

[1]http://leveldb.org

layers of Caffe. Two layers represent the neurons before and after the transformation. One layer between them defines how the transformation is performed. In Figure 6, we use ip$i$_j to denote the neuron layers of the ith level at the jth state. We show Caffe script of ip1_0 in the figure. All other neuron layers follow the same implementation. The layer is defined to use inner product transformation and we use gaussian distribution to generate the initial values. All bias values are set as 1 initially. To simulate the RNN, we share the same weight and bias matrixes among the neuron layers in the same level. E.g., layer ip1_0, ip1_1, ip1_2 and ip1_3 use the same weight and bias matrixes during their learning processes.

Between the neuron layers, we define the computation layer to use the ReLU function as our activation function. Other functions, such as sigmoid and tanh, can be used instead. But ReLU shows a better convergence rate in our experiments, and it is faster to compute ReLU using GPU. So our current implementation uses ReLU as our default activation function.

Finally, the top layer in the figure is the loss layer, where we can start the SGD to adjust the weights and biases. We use softmax function as our loss function and compare the prediction results with the ground truth (the num vector). The results of all loss layers are aggregated together to produce the final prediction results, a vector that represents the probability of a specific item bought by current user.

IV. Model Optimizations

The optimization techniques significantly affect the performance of the deep neural network [18]. In this section, we present our automatic optimization framework, which helps build a DRNN that can provide a good prediction accuracy and is fast to learn. Note that optimizing the deep learning process is extremely complicated. No theoretic rules can be applied and the detailed theoretic analysis is also beyond the scope of this paper. So our target is to estimate the performance heuristically and generate a good enough solution.

A. Automatic Code Generation

There are many parameters in our DRNN model, such as the number of states, the number of layers, the type of activation functions, the number of neurons and the loss function. In Caffe, to adjust those parameters, we need to update the definitions in our script or even completely rewrite the script. This makes it impossible to test the performance of different configurations. To address the problem, we build a code generator, which accepts a configuration file defining the values of parameters and outputs the Caffe script for the corresponding DRNN model.

Specifically, the generator first classifies the parameters into *basic parameters* (e.g., loss function and learning rate) and *network structure parameters* (e.g., the number of neurons per layer). Changing basic parameters only needs to reset their values, while updating network structure parameters will completely change the Caffe code. Currently, we support three-dimensional structure adjustment as shown in Figure 7.

The width of the network refers to the number of states that our DRNN model can maintain (except the history state). For each state, we will create a set of neural network layers and

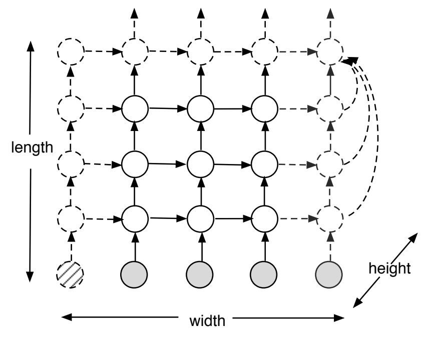


Fig. 7. Adjustment of Network Structure

Algorithm 1 CodeGen(int w, int l, int h)

```
1: Layer[] input = genInputLayer()
2: for $i = 0$ to $w - 1$ do
3:     Layer top = null, Layer btm =null
4:     for $j = 0$ to $l - 1$ do
5:         if btm $\neq$ null then
6:             btm = genNeuronLayer()
7:         top = genNeuronLayer()
8:         connectLayer(btm, top)
9:         btm = top
10:    Layer loss = genLossLayer()
11:    connectLayer(btm, loss)
12: Layer history_top = null, Layer history_btm =null
13: for $j = 0$ to $l - 1$ do
14:    if history_btm $\neq$ null then
15:        btm = genNeuronLayer()
16:    history_top = genNeuronLayer()
17:    connectLayer(history_btm, history_top)
18:    history_btm = history_top
19: for $i = 0$ to $w - 1$ do
20:    Layer loss = getLossLayer($i$)
21:    Layer next = input[$j$].getTopLayer()
22:    for $j = 0$ to $h - 1$ do
23:        connectLayer(next, loss)
24:        next = next.getTopLayer()
```

connect them to the layers of the consecutive state. Normally, more states lead to a more accurate model, but the maintenance overhead and learning cost increase dramatically. For our case, if the average size of a user session is X, the number of states in the network should be slightly less than X. Otherwise, the RNN may introduce noises for short sessions.

The length of the network is the number of layers used in the DRNN model. In other words, it decides how deep the neural network is. Although neural network with two layers can handle most applications, using more layers can fit the training samples more closely. However, we also need to address the “overfit" problem [23].

The height of the network is the number of short links between the hidden layers and the loss layer. Instead of following all the hidden layers to the loss layer, we can link a hidden layer directly to the loss layer to increase its impact on the learning results.

Algorithm 1 shows the workflow of our code generator. It first creates all neuron layers for current states (line 2-11). Then, it sets up the history state without the loss layer (line 13-18). Finally, we selectively connect neuron layers to their loss layers to create the shortcuts (line 19-24). To

Algorithm 2 GenTune(int w, int l, int h)

```
1:  $\tau = 0$
2:  $P(\tau)$ = GeneratePopulation()
3:  maxFit = maximum fitness of chromosomes in $P(\tau)$
4:  best = chromosome with maximum fitness in $P(\tau)$
5:  while $\tau < N$ and user does not interrupt do
6:      randomShuffle($P(\tau)$)
7:      for $i$ = 1 to $\frac{P(\tau).size}{2}$ do
8:          $c_1$= $P(\tau)[i * 2 - 1]$
9:          $c_2$ = $P(\tau)[i * 2]$
10:         $r_1$, $r_2$ and $r_3$ are random values in [0, 1]
11:         if $r_1 < p_c$ then
12:             crossover($c_1, c_2$)
13:         if $r_2 < p_m$ then
14:             mutate($c_1$)
15:         if $r_3 < p_m$ then
16:             mutate($c_2$)
17:         if $r_1 < p_c$ or $r_2 < p_m$ then
18:             maxFit = Max(getFitness($c_1$, maxFit)
19:             best = chromosome with maximum fitness in new $P(\tau)$
20:         if $r_1 < p_c$ or $r_3 < p_m$ then
21:             maxFit = Max(getFitness($c_2$, maxFit)
22:             best = chromosome with maximum fitness in new $P(\tau)$
23:     $\tau$++
```

simplify the task, we have predefined layer templates for different types of layers. The *genNeuronLayer* function selects the proper template and fills in the values of basic parameters to materialize those layers.

B. Model Tuning

Even for an experienced user, it is a challenging job to find the optimal parameter setting for a deep neural network. There are too many parameters and the effect of their combinations on the performance of the network is unknown. Moreover, it is impossible to build a mathematic model for the training process. Previous work [12][25] adopted the genetic algorithm to tune the structure of the neural network. They only focus on the number of neurons in each layer. In our DRNN, we also use the genetic algorithm as a heuristic tool to tune the network. However, we try to optimize all basic parameters and structure parameters.

We illustrate the basic idea in Algorithm 2. $P(\tau)$ is the chromosome set in the τth iteration. Each chromosome represents a configuration of C parameters, denoting as $(w, l, h, a_1, a_2, ..., a_L, ...)$. The first three elements define the width, length and height of the DRNN. The next L elements ($l < L$) are used to tune the number of neurons in each layer. Note that a_i ($l < i$) is actually not used. The remaining $C - L - 3$ elements are the basic parameters, describing the learning rate, loss function and etc. We use the *fit* value to evaluate the performance of a chromosome, which is defined as:

$$fit = accuracy + \frac{1}{1 + loss}$$

$accuracy$ is the prediction accuracy, if the chromosome is applied to the neural network. $loss$ is the loss rate. fit can only be computed, if we establish a network using the chromosome and run the learning process. However, we find that we can get a quite good estimation for the $accuracy$ and $loss$ after a few rounds of learning, if the model finally converges (e.g., Figure 10 and Figure 12).

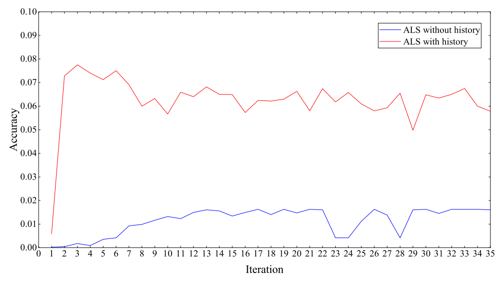


Fig. 8. Performance of ALS

So we run the learning process using the chromosome for at most R epochs (see experiment section for details). If the $accuracy$ is bounded by a predefined threshold π, we return the estimated fit value and terminate the learning process immediately. If $accuracy$ cannot be bounded in R epochs, we set fit value as 0.

Algorithm 2 tries to find the chromosome with the maximal fit value by randomly evolving all chromosomes for N times. Chromosomes are grouped into pairs and we apply *crossover* and *mutate* functions to update them. Specifically, *crossover* exchanges and merges values of two chromosomes to generate better genes. *mutate* randomly updates some genes of a chromosome.

The intuition of Algorithm 2 and other genetic algorithms is not to find the optimal chromosome. Instead, after a process of evolvement, those algorithms will reach a local optimal solution, which is good enough for complicated problems that are hard to model theoretically.

V. Experiments

To evaluate the proposed DRNN model, we deploy it on the Kaola testing system. The system repeats the same web log of June 1st, 2015, which consists of 232,326 records and can be grouped into 27,985 sessions. There are 37667 unique users in the log, who bought 1584 different items. The log is streamed to a document database (MongoDB[2] in current implementation) which is split into sessions and fed to the DRNN model. The DRNN model is trained on a GPU server equipped with two Xeon 2.6G 8-core CPUs, 64GB memory and one GeForce GTX Titian Z GPU (12GB DDR5). Caffe is configured to run on the GPU mode. During the training process, 60% sessions are used for training; 20% are used for validation; and the rest are used for testing. Normally, it takes 4 to 6 hours before a model completely converges.

By default, the DRNN model maintains four states and one history state. We use three hidden layers. The first two layers have 1000 neurons and the last layer only has 100 neurons. The default settings are listed in Table I. After tuning, we may update some of those values to achieve a better performance. The prediction accuracy is employed as our main metric. For each training session, we split it into multiple training samples, which share the same ground truth result, namely, whether the user purchased the item or not. For each sample, the DRNN outputs exactly one recommendation item. If the user finally

[2]https://www.mongodb.org

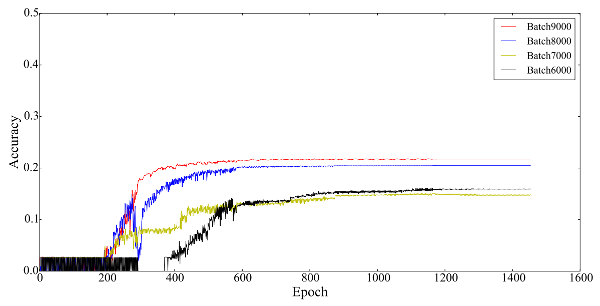


Fig. 9. Effect of Batching without Tuning

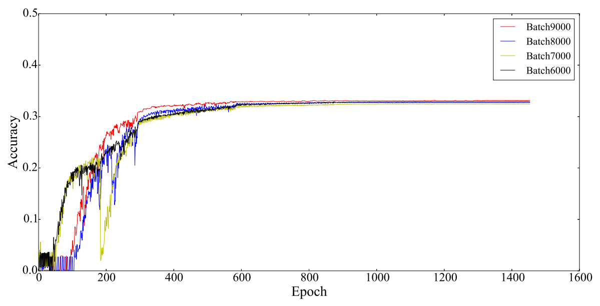


Fig. 10. Effect of Batching with Tuning

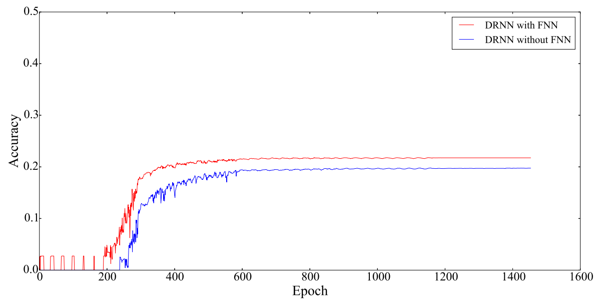


Fig. 11. Effect of FNN without Tuning

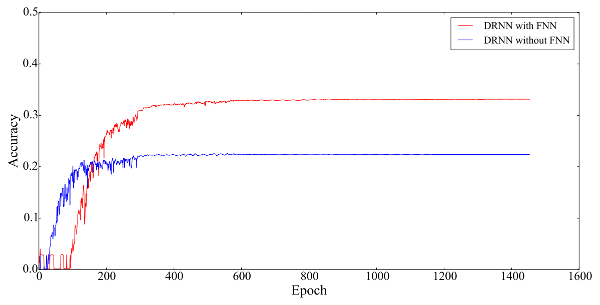


Fig. 12. Effect of FNN with Tuning

purchases the item in the session, our prediction is correct. So the accuracy is computed as:

$$\frac{f(S)}{|S|}$$

where S denotes all the training samples and $f(S)$ returns the number of samples with correct prediction.

The model is trained iteratively. In each iteration, we update the model using all our training samples. Each iteration is called an *epoch* in our experiment, and it may take hundreds of epochs before the model converges. For comparison, we also show the results of using ALS(Alternating Least Squares) [29] approach, one widely used collaborative filtering algorithm. We test two ALS approaches. The first one only uses the accessed pages in current session for its prediction, while the second one also considers all the access history of the corresponding user. Previously, Kaola runs the ALS on Spark [27] to generate the personalized recommendations. However, as shown in Figure 8, the accuracy is very low (less than 1% and 6% for ALS with/without history), even we have extensively tried different configurations. This is because the probability of users buying non-popular items is not significantly correlated with what they have bought before, invalidating the assumption of CF algorithm.

A. *Effect of Batching*

During the learning process, M samples are grouped into a batch, which are trained together. The batch size affects the accuracy of the model as shown in Figure 9 and Figure 10. Figure 9 uses the default settings for parameters, while Figure 10 applies our automatic tuning process. In either case, the performance is much better than the ALS approach, almost approaching 30%. This verifies the effectiveness of using RNN to catch the purchase patterns for each individual session. In the experiment, with a larger batch size, we normally get a better accuracy. But on the other hand, larger batches incur higher memory overhead. Our GPU server can support a batch of 10,000 samples at most. One interesting observation is that after the tuning process, the effect of batch size is not significant any more, which verifies that the tuning process can help optimize the model.

TABLE I. EXPERIMENT SETTINGS

Parameter	**Default Value**
Number of Hidden Layers in RNN	3
Number of Hidden Layers in FNN	3
Number of States in RNN	4
Number of Neurons in RNN	1000,100
Number of Neurons in FNN	1000
Activation Function	ReLU
Loss Function	Softmax
Initial Learning Rate	0.1
Batch Size in RNN	9000
Batch Size in FNN	5000
Weight Initialization Function	Xavier

B. *Effect of FNN*

Our model combines the results of a RNN and a FNN. The RNN is used to model the behaviors of current session, while the FNN describes the purchase habits of the user and his/her interest. In Figure 11 and Figure 12, we show the effect of FNN with/without the tuning process respectively. Using FNN can significantly improve the prediction accuracy, especially

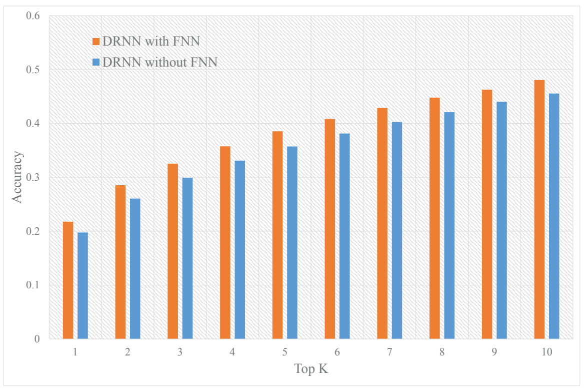


Fig. 13. FNN Top-K Result without Tuning

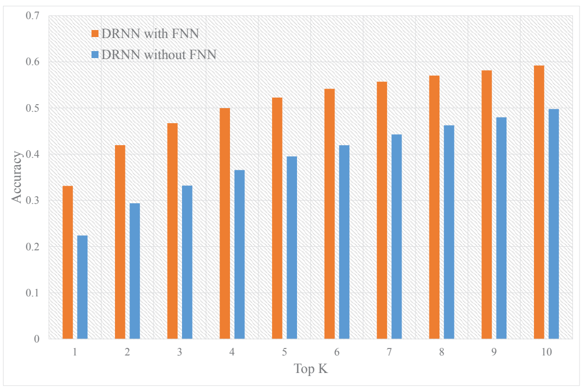


Fig. 14. FNN Top-K Result with Tuning

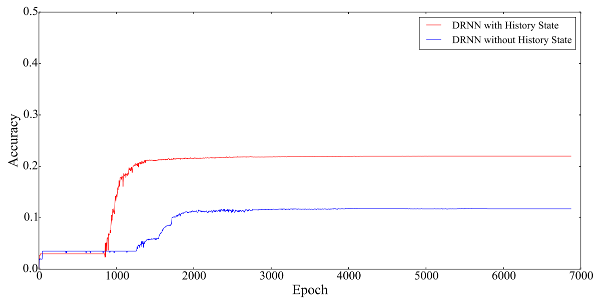


Fig. 15. Effect of History State without Tuning

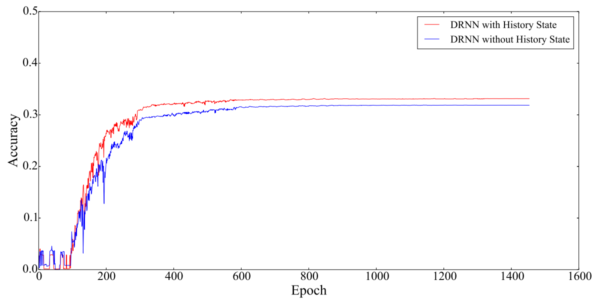


Fig. 16. Effect of History State with Tuning

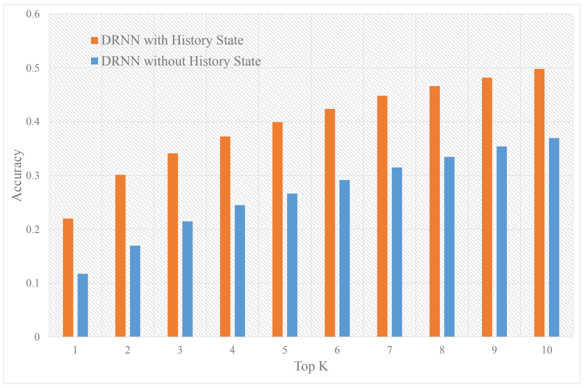


Fig. 17. History State Top-K Result without Tuning

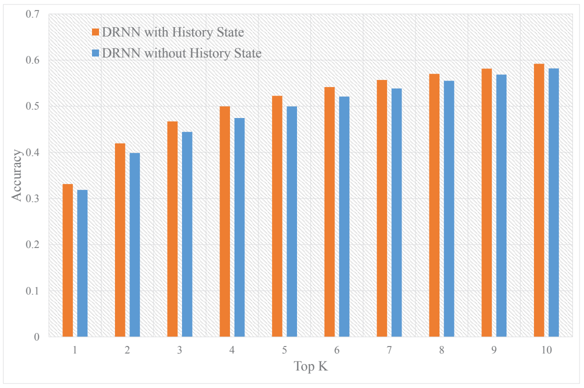


Fig. 18. History State Top-K Result with Tuning

after the tuning process. In Figure 12, the accuracy increases for about 10% by integrating the FNN into our model. Merging two neural networks significantly increases the complexity of our model. However, we find that it does not affect the convergence rate. In other words, the FNN does not incur more computation overheads.

By default, our model only returns one item as its prediction. In real systems such as Kaola, we can show k items in the recommendation widget. This may significantly increase the probability of a recommended item being purchased by a user. Figure 13 and Figure 14 show the accuracies when we increase the number of recommended items. In this experiment, the accuracy is computed as $\frac{g(S)}{|S|}$, where $g(S)$ returns the number of samples whose ground truth results are contained by our top-k recommendation. If $k = 10$, we can get an accuracy above 50% using our self-tuning process. This is a surprisingly good result for a personalized recommend problem. Sometimes, the DRNN model may be overfit for the training samples, but it is still a very promising technique, far superior than the CF algorithm.

C. *Effect of History State*

As mentioned before, the RNN can only maintain a limited number of states. To address the problem, we aggregate the information of stale states into one history state. Suppose the RNN can maintain m states. The history state is not valid for a user session which contains less than m web pages. In this experiment, we set m to 4 and only keep the sessions that have more than 5 states. The tuning process is also prohibited from changing the width of the neural network. In this case, there are 7,203 sessions left and used in the training process. Figure 15 and Figure 16 show the results of history state. Without the tuning process, maintaining the history state can improve the accuracy by almost 10%. On the contrary, if we tune the training process carefully, we can only achieve a 2% higher accuracy via using the history state. This observation shows the effectiveness of the tuning process which can reduce the impact of different network structures. Similarly, we also show the top-k results in Figure 17 and Figure 18. We get a similar trend as the experiments on FNN. Using tuning process partially neutralizes the effect of history state.

D. *Convergence Rate*

We find that the convergence rate is mainly correlated with the learning rate. By default, our initial learning rate ϕ is 0.1. After every 100 epochs, $\phi = \frac{\phi}{5}$. We lower the learning rate to avoid the fluctuation of the model. So the model can converge faster. Figure 19 illustrates the effect of initial learning rate on the convergence. Because our model actually contains two neural network, one RNN and one FNN, we say that the model converges only when both models are stable. We show the results with/without the FNN network. It is not surprising that without the FNN, the model converges faster. On the other hand, using a larger learning rate can reduce the time for converging, since the model will try to fit the samples more aggressively. However, note that using a larger learning rate, in fact, reduces the prediction accuracy, because the model may converge to a suboptimal point.

Finally, we show the effect of our recommendation in Table II. Let L_1 be the number of accessed pages in a session before the user adds an item t to his/her cart. Let L_2 be the number of accessed pages before our approach makes a correct prediction for t. We compute the path compression ratio as $\frac{L_2}{L_1}$.

TABLE II. PATH COMPRESSION RATIO

Method	Accuracy	Compression Ratio
DRNN without FNN	0.222638	0.771280
DRNN with FNN	0.331312	0.724127
ALS without History	0.016243	0.991366
ALS with History	0.063745	0.965895

VI. RELATED WORK

A. *Recurrent Neural Network*

The main difference between RNN (Recurrent Neural Network) and FNN (Feedforward Neural Network) is that the connection topology of RNN contains cycles, which are used to support temporal activation dynamics. In this way, RNN can use its “internal memory" to process arbitrary sequences of inputs. It becomes an effective tool for processing non-linear time series applications [10][17], such as handwriting recognition [6] and speech recognition [7]. It was shown that under proper assumptions, RNN can be used to simulate many dynamical systems [5].

However, RNN is hard to train using the classical gradient-descent-based methods. Sometimes, the network fails to converge [14], because gradual change of network parameters during the iterative learning processes may cause the network bifurcations [4]. A number of training approaches have been proposed to improve the learning process. Readers can refer to [2] for a survey of classical gradient descent RNN training methods.

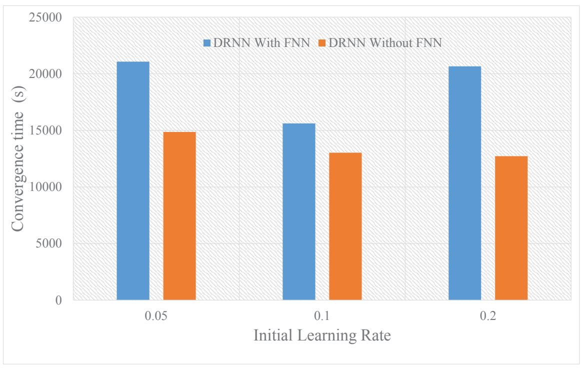


Fig. 19. Convergence Rate

Optimizing gradient-descent-based methods involves many tuning processes, requiring substantial skill and experience to be successfully applied. Alternatively, a fundamental new training technique, Reservoir Computing, was introduced [16][24][14]. The initial RNN is called the *reservoir*, which maintains the input signals in its states as history information. The output is generated as a linear combination of the neural signals of the reservoir using linear regression. Reservoir computing has been widely adopted to support the real-time systems modeled with bounded resources [15] and shows a higher accuracy than the gradient-descent-based approach. Our approach in this paper uses some ideas form the reservoir computing to improve our prediction accuracy.

B. *Personalized Recommendation*

Personalized recommendation can significantly improve the user experience and hence, become the key feature of many social websites and e-commerce systems. Two popular recommendation techniques are applied, the CF approach (Collaborative Filtering) [20][13] and the content-based approach [3][28].

The CF approach can be further classified into heuristic-based methods and model-based methods. Heuristic-based methods compute the similarity between a user and all the other users in the database using data such as ratings, purchase history, common links and duration time [1]. Model-based methods, on the other hand, will first establish a prediction model and estimate the similarity using the model [26]. The main problem of CF-based approaches is the cold-start problem [21]. Namely, without enough samples, the CF model cannot provide a satisfied result.

Content-based approaches are a type of information filtering techniques. They are also employed to provide news or microblog recommendations. State-of-the-art IR approaches, such as TF/IDF measure, clustering and association rules, are used in the context-based recommendation.

Recently, the CF approach and content-based approach are combined together to provide a more accurate result for the social network [11][22]. Those work also consider the structure of the social network as the implicit relationship between users. It was shown that the social-based approach outperforms the content-based one, especially for the friend recommendation task.

To our knowledge, none of existing work applies the RNN for personalized recommendation in the e-commerce systems. The RNN can effectively model the interactions between users and e-commerce systems, and detect the purchase patterns. Our results verify that the RNN performs much better than the CF approach.

VII. Conclusion

In this paper, we present our real-time recommendation approach powered by the DRNN (Deep Recurrent Neural Network) model. Our approach is designed for the NetEase's e-commerce website, Kaola(http://www.kaola.com), and has been successfully deployed for beta test. In the DRNN model, a user session is represented as a sequence of web pages, denoting the path from the first page to the purchased item. DRNN model extracts the common purchase patterns of users and tries to shorten the path for future users. So users can quickly reach the pages of their desired products. A user session may consist of an arbitrary number of web pages, while DRNN model can only maintain a limited number of states. Therefore, we propose using a history state to aggregate all the information of stale states. Finally, in the DRNN, we also implement a FNN (Feedforward Neural Network) to model the purchase history of users. The two networks are integrated together to produce the final prediction. We develop an optimizer to automatically tune the parameters of our neural networks to achieve a better performance. Our results on real dataset show that the DRNN approach outperforms previous CF (Collaborative Filtering) approach significantly.

Acknowledgment

The work of Sai Wu and Gang Chen was partly supported by National Basic Research Program (973 Program, Grant No. 2015CB352400).

References

[1] C. C. Aggarwal, J. L. Wolf, K.-L. Wu, and P. S. Yu. Horting hatches an egg: A new graph-theoretic approach to collaborative filtering. In *SIGKDD*, pages 201–212. ACM, 1999.

[2] A. F. Atiya and A. G. Parlos. New results on recurrent network training: unifying the algorithms and accelerating convergence. *IEEE Transactions on Neural Networks*, 11(3):697–709, 2000.

[3] D. Billsus and M. J. Pazzani. User modeling for adaptive news access. *User modeling and user-adapted interaction*, 10(2-3):147–180, 2000.

[4] K. Doya. Bifurcations in the learning of recurrent neural networks 3. *learning (RTRL)*, 3:17, 1992.

[5] K. Funahashi and Y. Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[6] A. Graves, M. Liwicki, H. Bunke, J. Schmidhuber, and S. Fernández. Unconstrained on-line handwriting recognition with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 577–584, 2008.

[7] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Processing of IEEE International Conference on Acoustics, Speech and Signal*, pages 6645–6649. IEEE, 2013.

[8] M. Hermans and B. Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 190–198, 2013.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[10] K.-i. Kamijo and T. Tanigawa. Stock price pattern recognition-a recurrent neural network approach. In *International Joint Conference on Neural Networks*, pages 215–221. IEEE, 1990.

[11] K. Lerman. Social networks and social information filtering on digg. In *ICWSM*, 2007.

[12] F. H. Leung, H.-K. Lam, S.-H. Ling, and P. K. Tam. Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *Neural Networks, IEEE Transactions on*, 14(1):79–88, 2003.

[13] G. Linden, B. Smith, and J. York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing*, 7(1):76–80, 2003.

[14] M. LukošEvičIus and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[15] W. Maass, P. Joshi, and E. D. Sontag. Principles of real-time computing with feedback applied to cortical microcircuit models. *Advances in neural information processing systems*, 18:835, 2006.

[16] W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

[17] T. Mikolov, S. Kombrink, L. Burget, J. H. Černockỳ, and S. Khudanpur. Extensions of recurrent neural network language model. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5528–5531. IEEE, 2011.

[18] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng. On optimization methods for deep learning. In *ICML*, pages 265–272, 2011.

[19] Y. Ouyang, W. Liu, W. Rong, and Z. Xiong. Autoencoder-based collaborative filtering. In *Neural Information Processing*, pages 284–291. Springer, 2014.

[20] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, pages 285–295. ACM, 2001.

[21] A. I. Schein, A. Popescul, L. H. Ungar, and D. M. Pennock. Methods and metrics for cold-start recommendations. In *SIGIR*, pages 253–260. ACM, 2002.

[22] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. In *SIGKDD*, pages 678–684. ACM, 2005.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[24] J. J. Steil. Backpropagation-decorrelation: online recurrent learning with o (n) complexity. In *Proceedings. 2004 IEEE International Joint Conference on Neural Networks*, volume 2, pages 843–848. IEEE, 2004.

[25] J.-T. Tsai, J.-H. Chou, and T.-K. Liu. Tuning the structure and parameters of a neural network by using hybrid taguchi-genetic algorithm. *Neural Networks, IEEE Transactions on*, 17(1):69–80, 2006.

[26] S. Vucetic and Z. Obradovic. Collaborative filtering using a regression-based approach. *Knowledge and Information Systems*, 7(1):1–22, 2005.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[28] Y. Zhang, J. Callan, and T. Minka. Novelty and redundancy detection in adaptive filtering. In *SIGIR*, pages 81–88. ACM, 2002.

[29] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.