Personal Recommendation Using Deep Recurrent Neural Networks in NetEase

2016.08.29

서지혜

Personal Recommendation Using Deep Recurrent Neural Networks in NetEase

Sai Wu #1, Weichao Ren #2, Chengchao Yu #3, Gang Chen #4, Dongxiang Zhang ◁5, Jingbo Zhu ◇6

College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[1,2,3,4] {wusai, weichaor, 21311200, cg}@zju.edu.cn

◁ *School of Computer Science and Engineering, University of Electronic Science and Technology of China*
[5] zhangdongxiang37@gmail.com

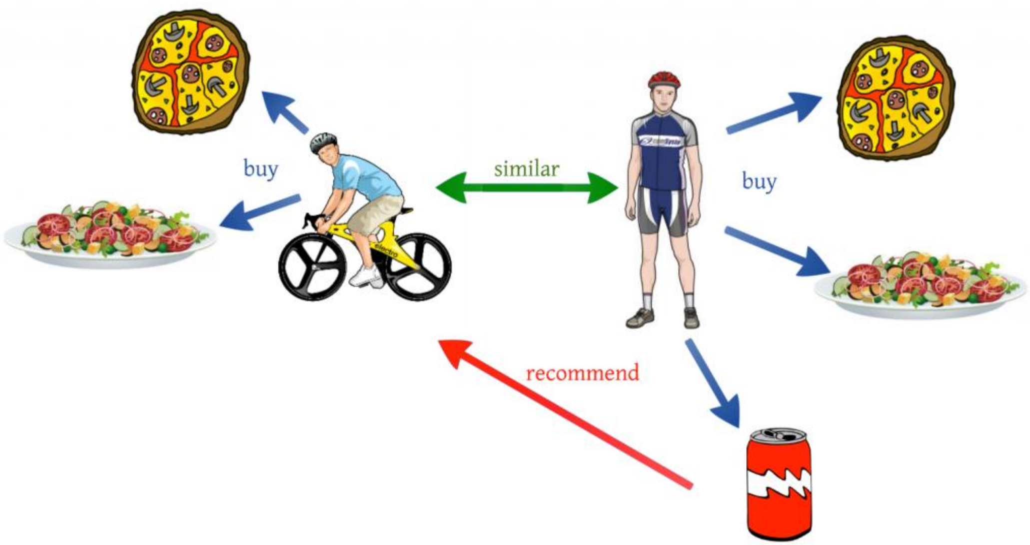
◇ *NetEase (Hangzhou) Network Co., Ltd., Hangzhou, China*
[6] zhujingbo@corp.netease.com

***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**

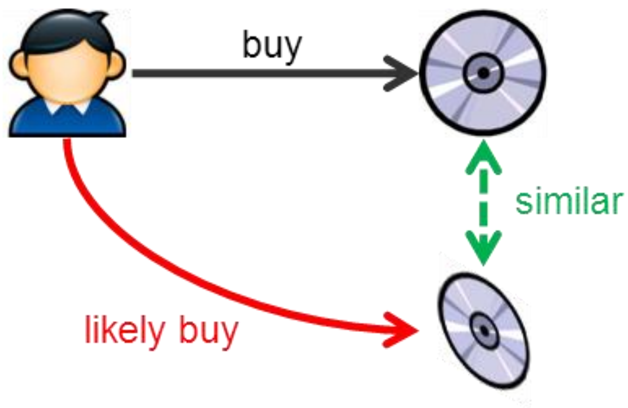
session, which update their values throughout the session. Before making his/her purchase, a user will view a long list of web pages (In Kaola, a user views 12 pages averagely before adding one item to the cart). This viewing history actually records how users interact with the system and perform their 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

Collaborative Filtering

User-based Recommendation



Item-based Recommendation



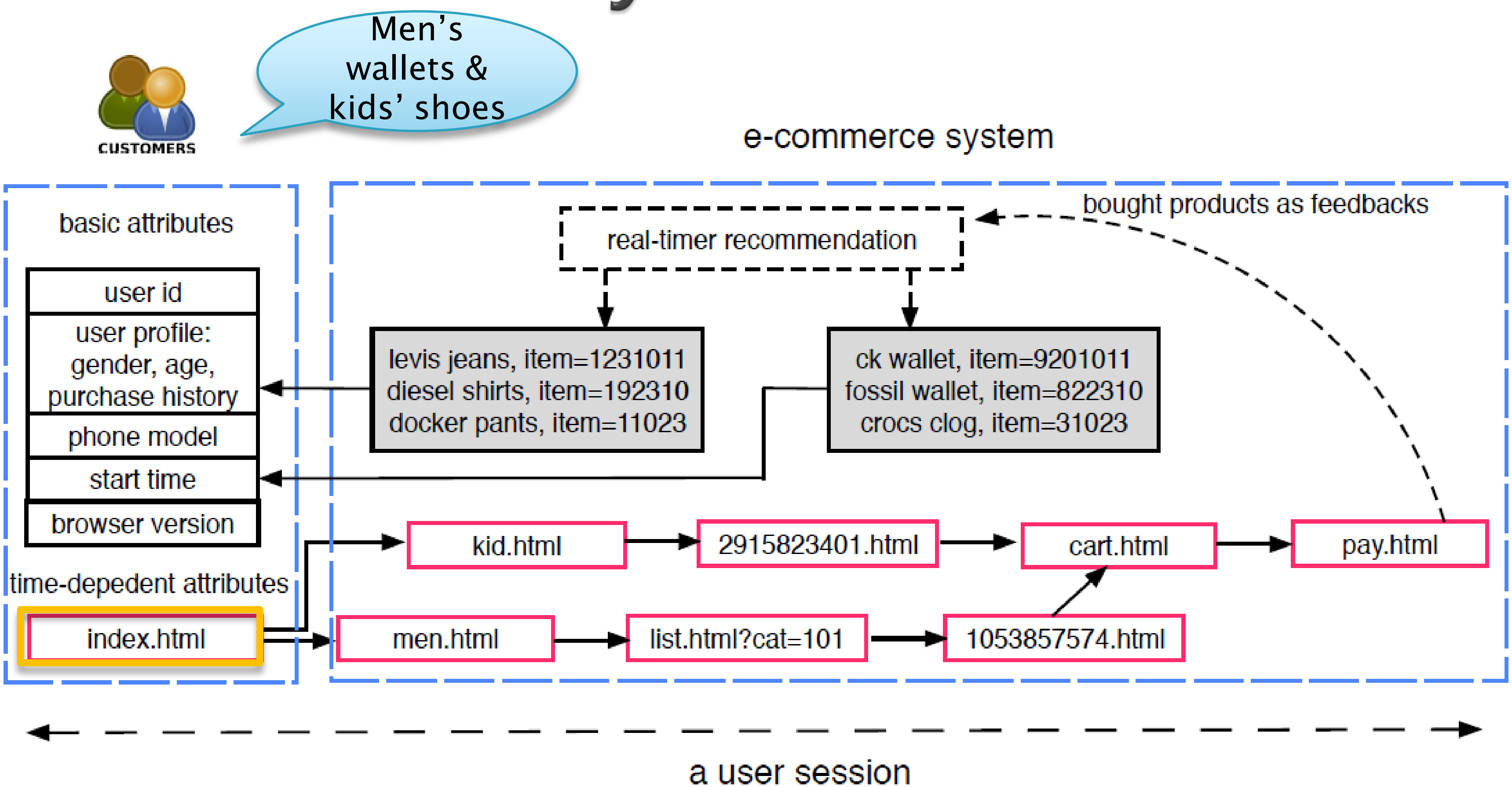
출처 : https://blog.guillaumeagis.eu/recommendation-algorithms-with-apache-mahout/

Personal Recommendation Using Deep Recurrent Neural Networks in NetEase

- Collaborative Filtering
  - Unable to provide a real-time customized recommendation service
  - Established using stale data

- In this paper
  - Using a history state
  - Build a DRNN with FNN
  - Develop optimizer to automatically tune the parameters of NN

Real-time Recommendation in E-commerce System



Overview of Recommendation Module

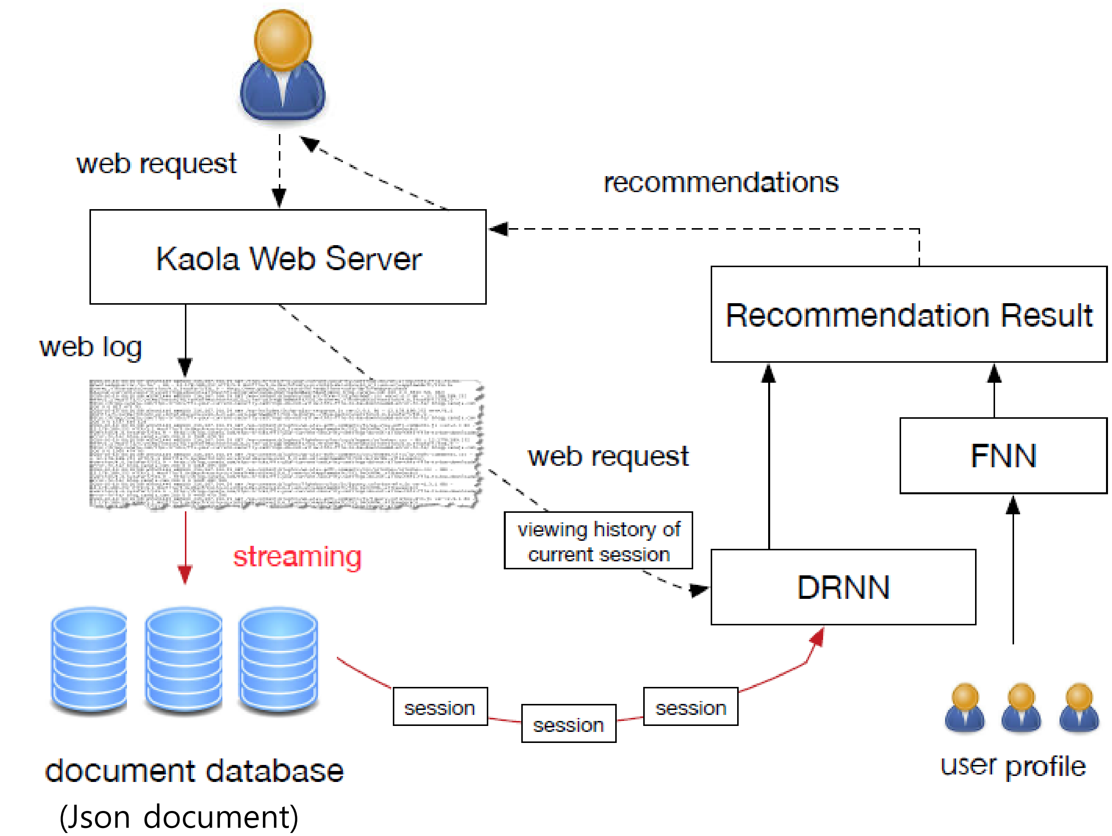


Fig. 2. Work Flow of Recommendation Module

- Consider both

Interest of current session (DRNN)

and past interest (FNN)

- Session continues

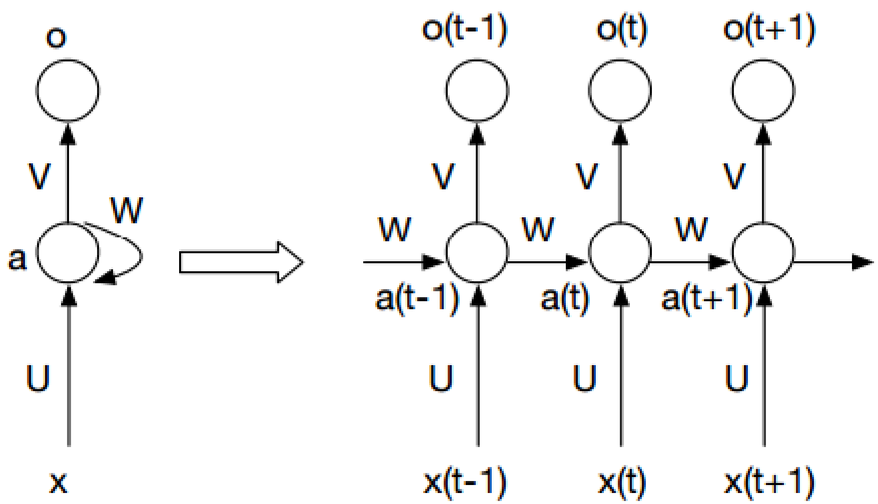
Gradually refine

- Session completes

Adjust our model using new training sample

Deep Recurrent Neural Network

- RNN



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

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

Deep Recurrent Neural Network

- Deep RNN Model with Infinite State

  Input → $\{p_0, p_1, \ldots, p_{n-1}\}$ (a sequence of web pages)

  Output → $\{v_0, v_1, \ldots, v_{N-1}\}$

  - Update function

    $a_i(t)$ denote the state of the ith layer at state t.

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

W_i : weights of connections from state t-1

Z_i : weights of connections from the lower hidden layer at the same state

$b_i(t)$: bias

$\theta(p_t)$: bias for input layer

Deep Recurrent Neural Network

- Deep RNN Model with Infinite State

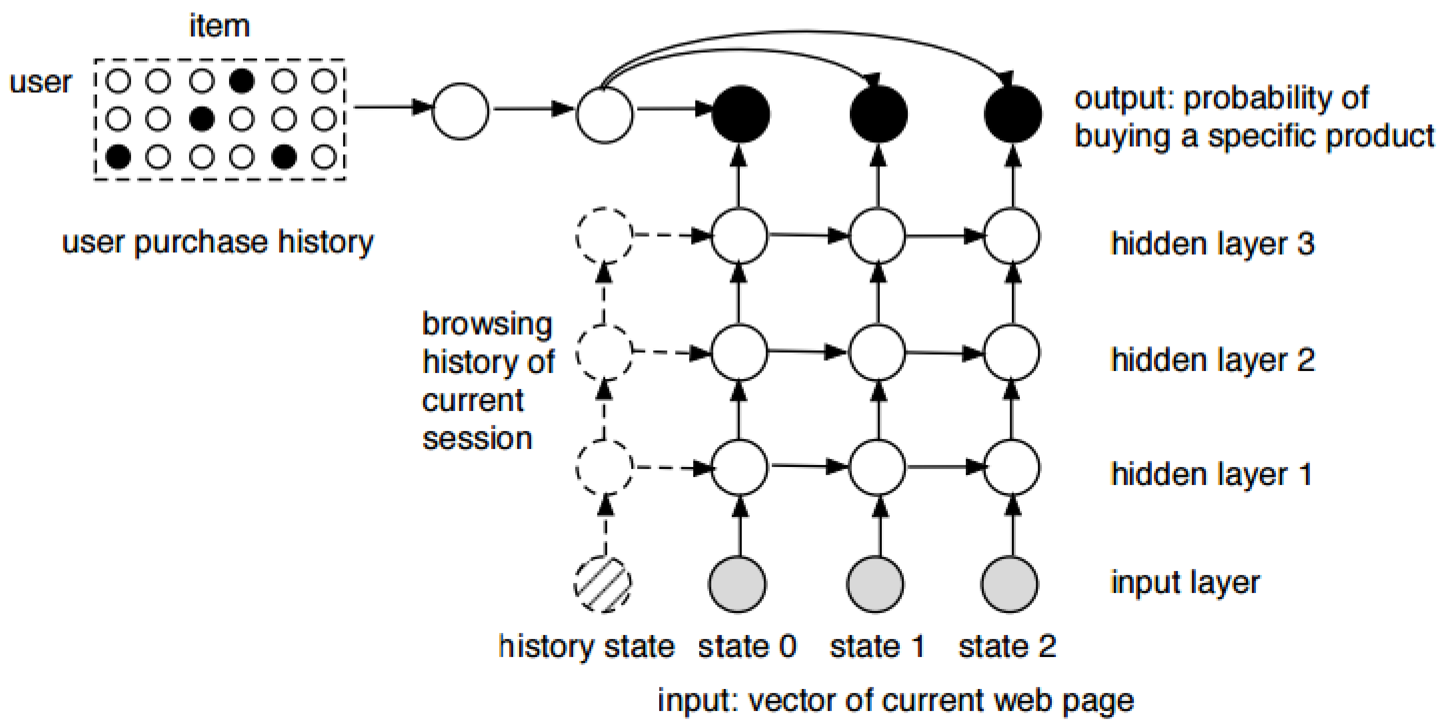


Fig. 4. Illustration of Deep RNN model

Deep Recurrent Neural Network

Deep RNN Model with History State

- All pages : $\{p_0, p_1, \ldots, p_x\}$ If $x \leq n$ → history state is not used.

- Input for the history state

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

V_i : vector for page p_i

ϵ_i : the aging factor for old state

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

Deep Recurrent Neural Network

- Collaboration with Collaborative Filtering
  - CF (Feedforward Neural Network)
    → Generates a good recommendation if users follow their old purchase patterns
  - RNN
    → Effectively predict the unexpected purchase of a specific user

Deep Recurrent Neural Network

- Collaboration with Collaborative Filtering

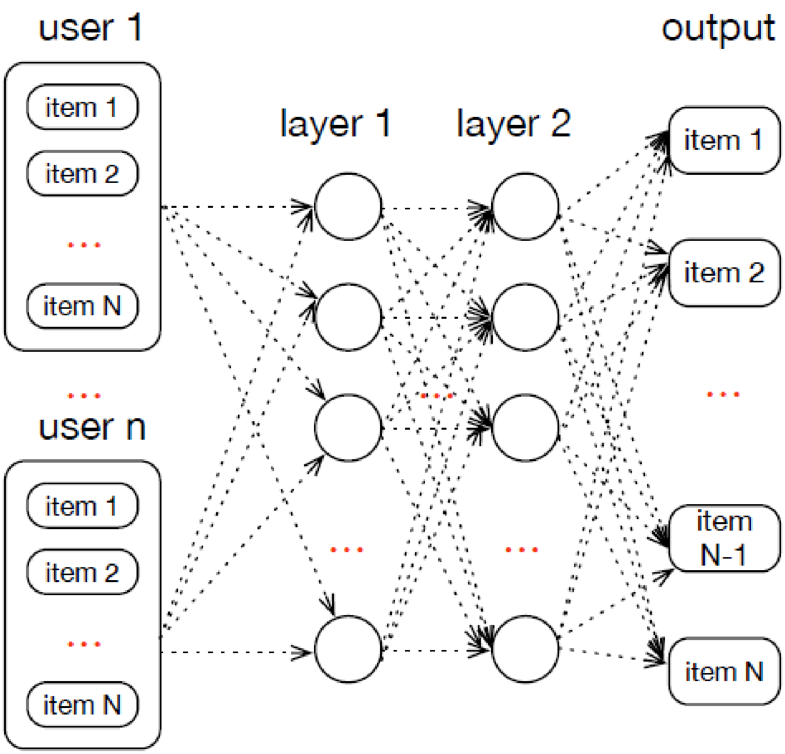


Fig. 5. FNN for CF

- State update

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

$\bar{a}_j^{(i)}$: jth neurons at the ith hidden layer

$\bar{E}$: number of each hidden layer neurons

$w_j^{(i-1)}$: weight from the last layer

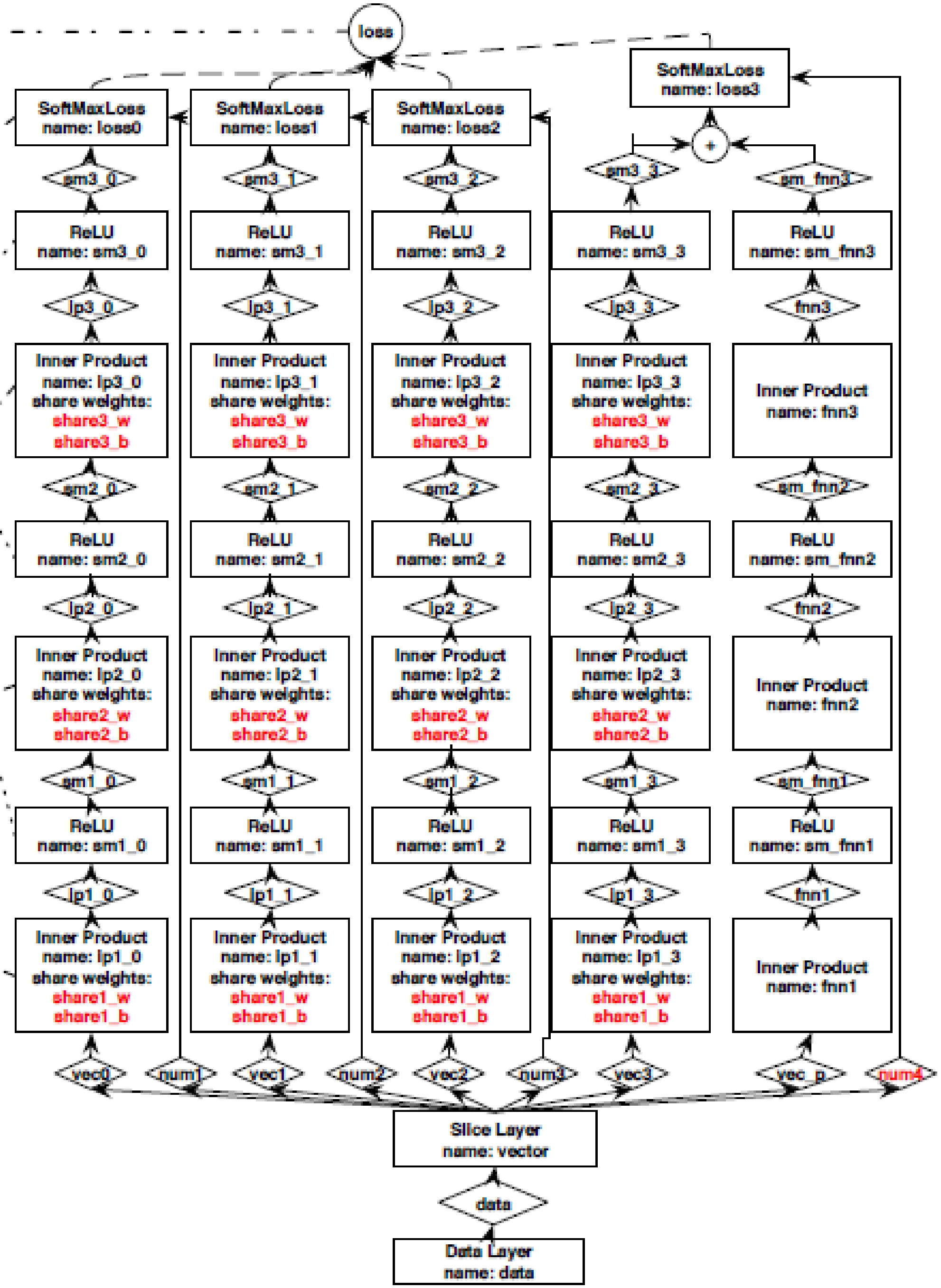
$b_x^{(i-1)}$: corresponding bias

DRNN Implementations

```
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
   }
}
```



Automatic Optimization Framework

- Automatic Code Generation
  - Parameters : number of states/neurons, type of activation functions and the loss function
  - Code generator
    - Accepts a configuration file defining the values of parameters
    - Outputs the Caffe script for the corresponding DRNN model
  - Basic parameters / network structure parameters

Automatic Optimization Framework

- Automatic Code Generation

Creates all neuron layers for current states

Sets up the history state without the loss layer

Connect neuron layers to their loss layers to create the shortcuts

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

```
Layer[] input = genInputLayer()
for i = 0 to w − 1 do
    Layer top = null, Layer btm =null
    for j = 0 to l − 1 do
        if btm ≠ null then
            btm = genNeuronLayer()
        top = genNeuronLayer()
        connectLayer(btm, top)
        btm = top
    Layer loss = genLossLayer()
    connectLayer(btm, loss)
Layer history_top = null, Layer history_btm =null
for j = 0 to l − 1 do
    if history_btm ≠ null then
        btm = genNeuronLayer()
    history_top = genNeuronLayer()
    connectLayer(history_btm, history_top)
    history_btm = history_top
for i = 0 to w − 1 do
    Layer loss = getLossLayer(i)
    Layer next = input[j].getTopLayer()
    for j = 0 to h − 1 do
        connectLayer(next, loss)
        next = next.getTopLayer()
```

Automatic Optimization Framework

- Model Tuning

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

$\tau = 0$
$P(\tau)$ = GeneratePopulation()
maxFit = maximum fitness of chromosomes in $P(\tau)$
best = chromosome with maximum fitness in $P(\tau)$
while $\tau < N$ and user does not interrupt **do**
  randomShuffle($P(\tau)$)
  for i = 1 to $\frac{P(\tau).size}{2}$ **do**
    c_1= $P(\tau)[i * 2 - 1]$
    c_2 = $P(\tau)[i * 2]$
    r_1, r_2 and r_3 are random values in [0, 1]
    if $r_1 < p_c$ **then**
      crossover(c_1, c_2)
    if $r_2 < p_m$ **then**
      mutate(c_1)
    if $r_3 < p_m$ **then**
      mutate(c_2)
    if $r_1 < p_c$ or $r_2 < p_m$ **then**
      maxFit = Max(getFitness(c_1, maxFit)
      best = chromosome with maximum fitness in new $P(\tau)$
    if $r_1 < p_c$ or $r_3 < p_m$ **then**
      maxFit = Max(getFitness(c_2, maxFit)
      best = chromosome with maximum fitness in new $P(\tau)$
  τ++

$P(\tau)$: chromosome set in the τth iteration

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

Experiments

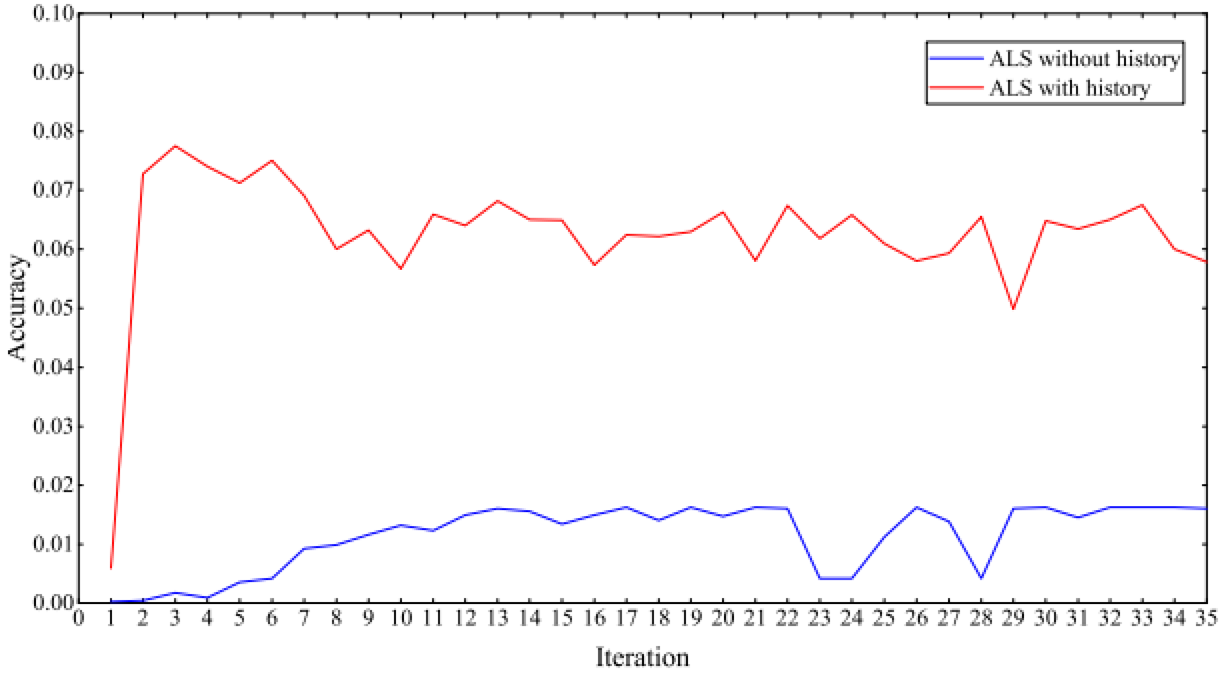
- Data
  - Web log : 232,326 records (June 1st , 2015)
  - 27,985 session 37,667 unique users
  - 60% training , 20% validation, 20% testing
- Equipment
  - Xeon 2.6G 8-core CPUs / 64GB memory
  - One GeForce GTX Titian Z GPU (12GB DDR5)
  - Caffe (GPU mode)
- DRNN model structure
  - 4 state
    - One history state
  - 3 hidden layers

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

Experiments

- Performance of ALS



Experiments : Effect of Batching

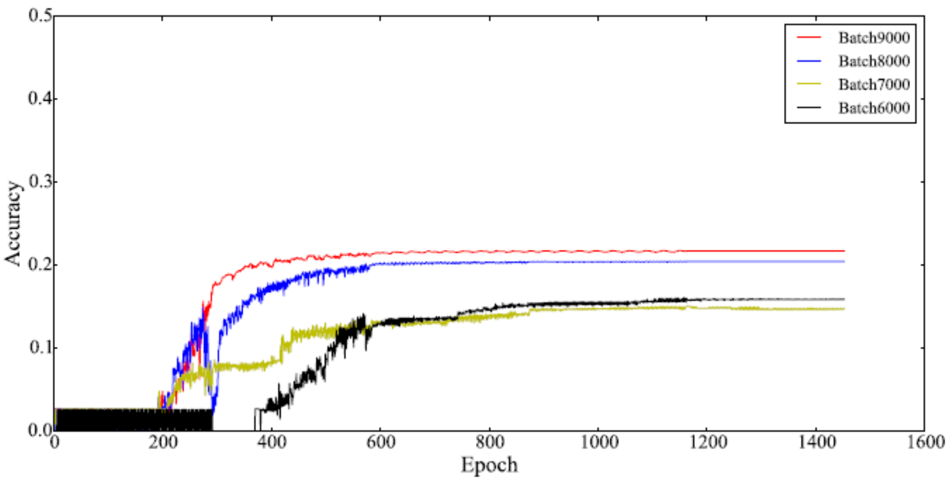


Fig. 9. Effect of Batching without Tuning

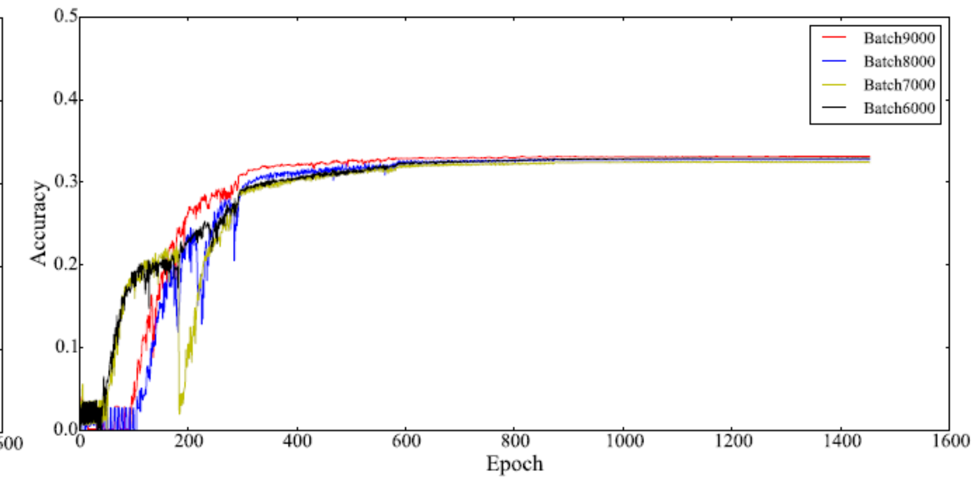


Fig. 10. Effect of Batching with Tuning

Experiments : Effect of FNN

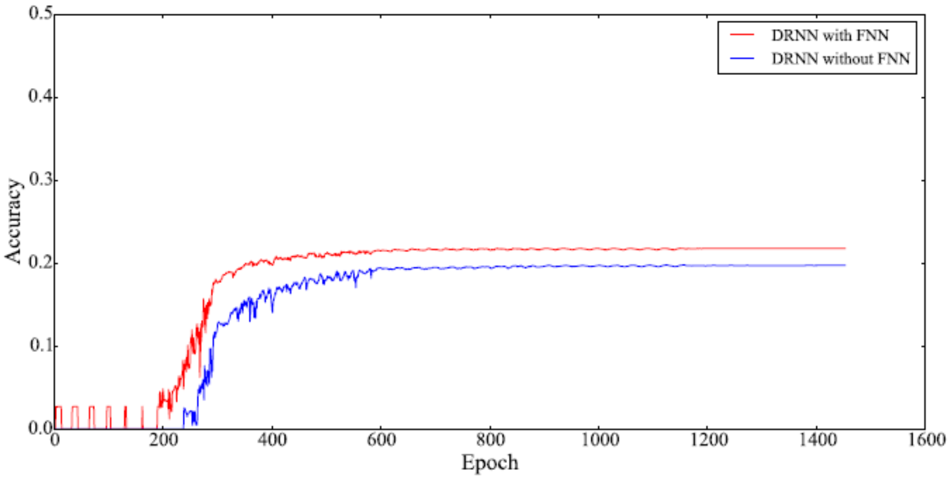


Fig. 11. Effect of FNN without Tuning

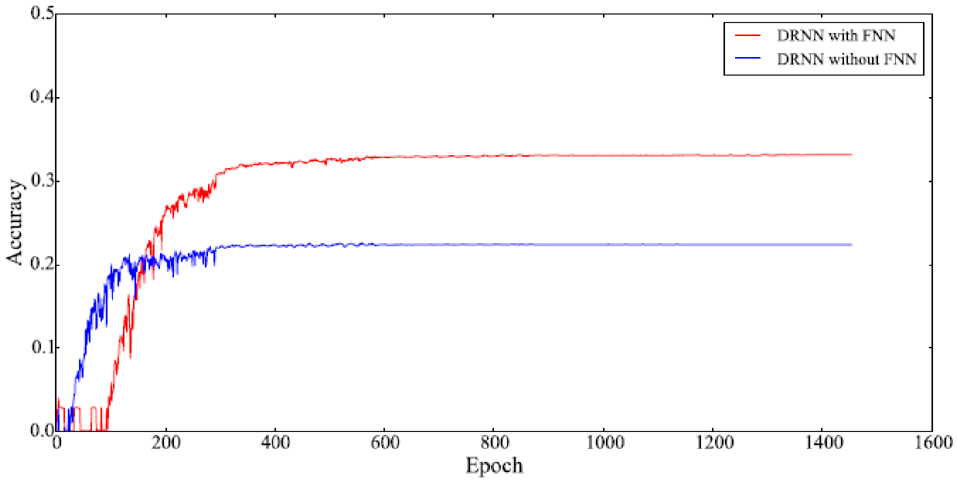


Fig. 12. Effect of FNN with Tuning

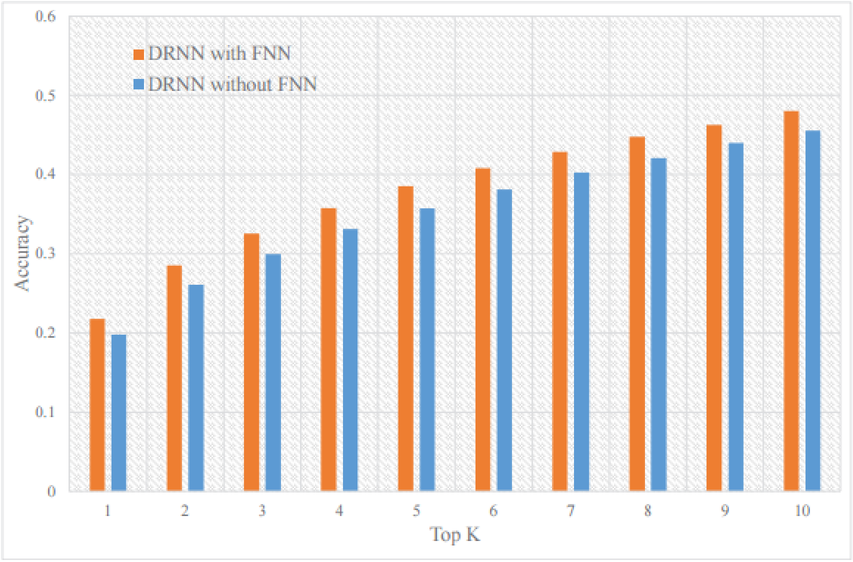


Fig. 13. FNN Top-K Result without Tuning

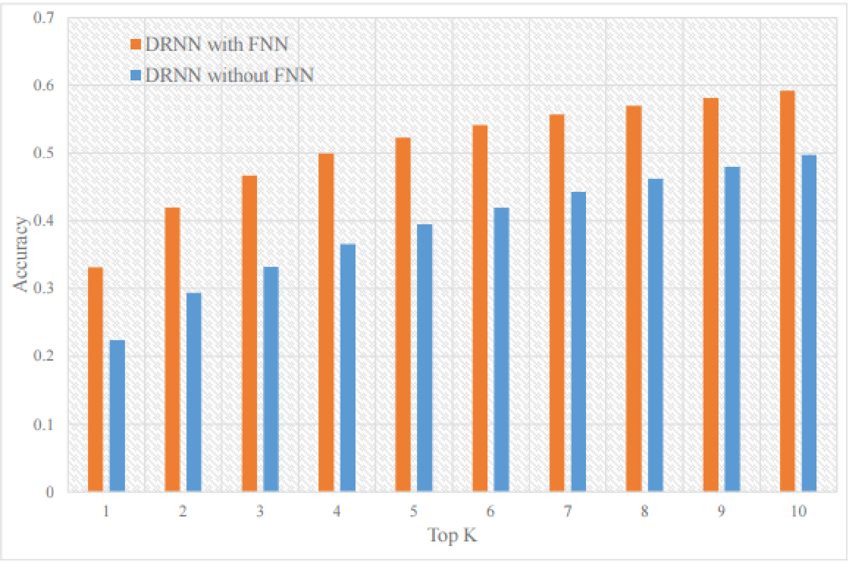


Fig. 14. FNN Top-K Result with Tuning

Experiments : Effect of History state

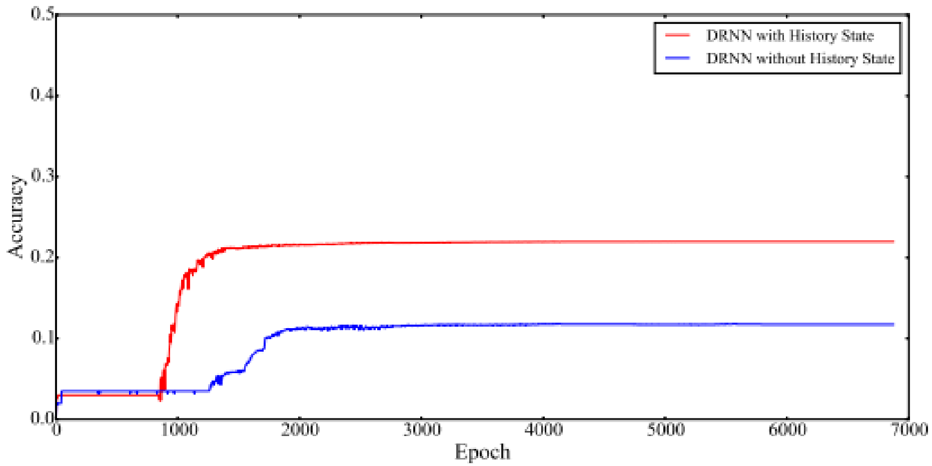


Fig. 15. Effect of History State without Tuning

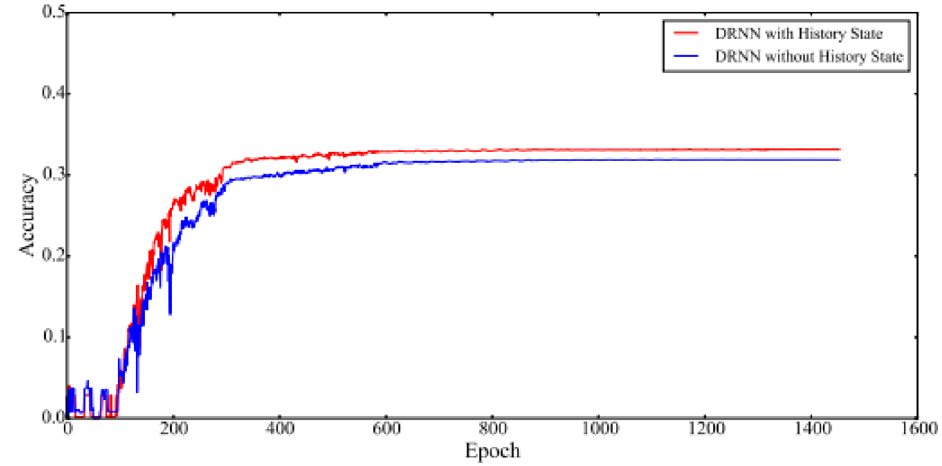


Fig. 16. Effect of History State with Tuning

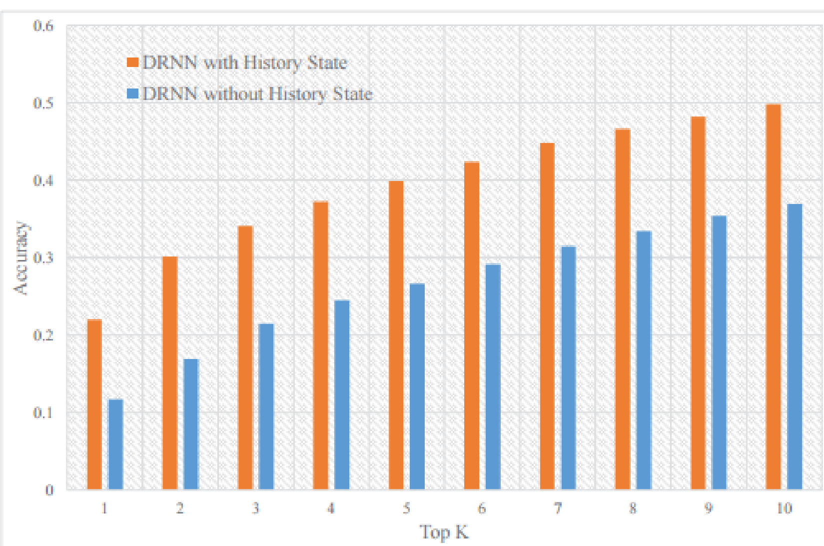


Fig. 17. History State Top-K Result without Tuning

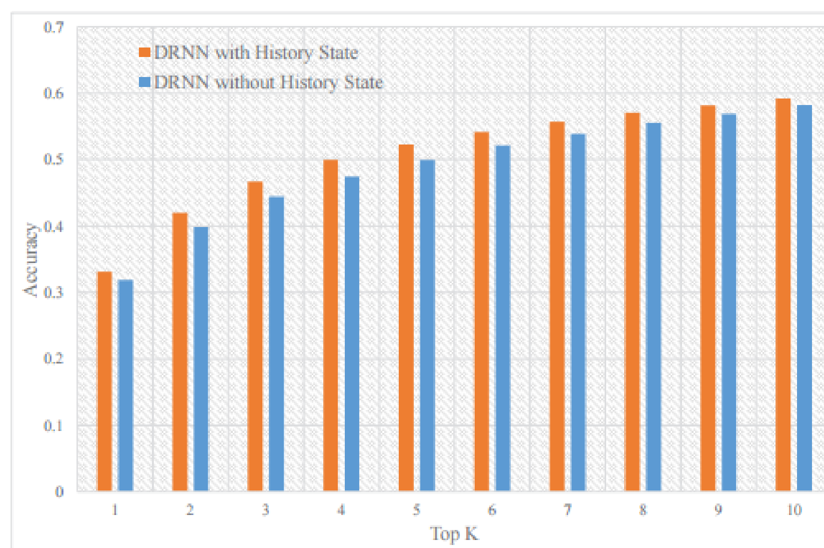


Fig. 18. History State Top-K Result with Tuning

Experiments

Convergence Rate

- Initial learning rate

$$\phi = 0.1$$

- Every 100 epochs

$$\phi = \frac{\phi}{5}$$

Fig. 19. Convergence Rate