EasyChair Preprint

№ 1053

GBDT, LR & Deep Learning for Turn-based Strategy Game AI

Like Zhang, Hui Pan, Qi Fan, Changqing Ai and Yanqing Jing

May 28, 2019

GBDT, LR & Deep Learning for Turn-based Strategy Game AI

Like Zhang, Hui Pan, Qi Fan, Changqing Ai, Yanqing Jing

Turing Lab, Tencent Inc.

{likezhang, qfan, changqingai, frogjing}@tencent.com, hpan2010@my.fit.edu

***Abstract*-This paper proposes an AI fighting strategies generation approach implemented in the turn-based fighting game StoneAge 2 (SA2). Our research aim is to develop such AI for choosing the logical skills and targets to the player. The approach trained the logistical regression (LR) model and deep neural networks (DNN) model, individually. And combined both output at inference process. Meanwhile, to transform the features into a higher dimension binary vector without any manual intervention or any prior knowledge, we put all category features into Gradient Boosted Decision Tree (GBDT) before LR component. The main advantage of this procedure is the approach combines the benefits of LR models (memorization of feature interactions) and DL (generation the unseen feature combination through low-dimensional dense feature) for the AI fighting system. In our experiment, we evaluated our model with some other AI strategies (Reinforcement Learning (RL), GBDT, LR, DNN) to against a robot script. The results shown that the players, participating in the experiment, are capable of using reasonable strategic skills on the different targets. As a consequence, the win rate (versus with the robot script) of our system is higher than the others. Finally, we productionized and evaluated the system on SA 2, a commercial mobile turn-based game.**

Keywords

Machine Learning, Logistical Learning, Gradient Boosted Decision Tree, Deep Neural Network, Reinforcement Learning, turn-based games;

1. Introduction

Since AlphaGo got some historic successes in the couple of last years, reinforcement learning and deep neural network has opened a new era for the game, more and more AI technology has been applied to the game field. As early as 2013, DeepMind demonstrates that a convolutional neural network can learn successful control policies from raw video data in complex RL environments [18][19][20][21][22][23]. Recently, StarCraft becomes a new challenge for deep reinforcement learning research [2]. Comparing to the aged Atari games which were built upon limited action choices[24], the action space in current commercial computer games are vase and diverse, the player selects actions among a combinatorial space of approximately hundreds of possibilities leading to a rich set of challenges. And those challenges have not been overcome yet. These previous research efforts suggested developing an approach for a specific goal in game is more feasible than trying to build a general purpose AI model.

In this paper, we design a system that when player temporary leaving or offline, the system can choose the skills and apply it on the target, reasonably. Meanwhile, we productionized and evaluated the system on a mobile turn-based game StoneAge2 [1].

StoneAge 2 is a turn-based strategy game in which players will build their own group of monsters, deciding building strategies by distributing "building points", and arrange the combination of monsters to form up the line in battles. fighting part is one of the most important part for entertaining players (Fig. 1). In our game, each combat includes twenty thirty-seconds rounds. And, in each round, the play must choose a skill from the skill list and apply this skill to a target (the target could be our side or the opponent) based on the different situations.

Figure 1, screenshot of SA combat board, right side is the skill list, left top is the opponent side, left bottom is the player side.

There are two different scenarios for the battles: PvE(Boss Battle) and PvP (Player vs Player). In the PvE scenario, usually game developers have to create different behavior scripts for the NPCs (the Boss) and make sure the script-defined behaviors will give some challenges for players. While in PvP scenario, in most cases there won't be any automated behavior for either side since player themselves are expected to make all the moves. These approaches work well for traditional computer games, but exhibit drawbacks for mobile games like StoneAge 2. The major problems include:

- Mobile games have much shorter development cycle than traditional computer games. Designers expect a simple

solution to create an AI for the NPC, not complicated action scripts.

- For mobile game players, there are many reasons they can go offline. For example, a player in a battle could be simply waiting in the line for bus. When the bus is coming, he could just turn off the game and forget about the battle. The unpredictable behavior from mobile game players bring extremely bad gameplay experience for PvP scenarios, and designers are eager to find a solution by replacing "offline" players with certain AI models.

To solve the above problems, we explored different machine learning based algorithms and successfully designed a mixed approach into StoneAge 2.

At the beginning of our experiments, we also applied RL environment on our game, but its performance is lower than our expectation. The reason is: we defined the percentage of the sum(HPs) of the opponents as the reward function. the RL AI tendency use attack kind skill rather than others. The shortage is that the assistant role or the healer tried to use the normal attack skill instead of some other assistant skills, such as speed up skill, control skill, etc. Therefore, we proposed a mix-model which combines decision tree, logistic regression plus deep learning and get the better result. The comparison results have been displayed in Section 5.

Our system can be viewed as a recommendation system, where the input query is the current battle status, such as each player's current HP, MP, buff, role speed, etc. And the output is a ranked list of skill-target pair by the scores. The scores are the probabilities of the win-rate of the player apply this skill on the target. Given a status, the system is to find the all legal pairs of skills and targets, and then rank these pairs. Therefore, the traditional challenge in recommender systems is also present in ours, which is to achieve both memorization and generalization. In our case, generalization can be defined as explores and transitivity the skills which have never or rarely used in the past. On the other side, memorization is based on learning the frequent co-occurrence of skills and targets in the historical data. In previous work by H. Cheng et al [3], a combination of wide linear model and DNN was proposed that focuses on achieving this goal. The model's prediction is:

$$P(Y = 1|x) = \sigma(W_{wide}^{T}[x, \emptyset(x)] + W_{deep}^{T} + b) \quad (1)$$

where Y is the binary class label, $\sigma(\cdot)$ is the sigmoid function, W_{wide} and W_{deep} are the vector of all wide and deep model weights, respectively. $\emptyset(x)$ are the cross-product transformations of the original feature **x**. And, b is the bias term. This model requires more feature engineering effort or the prior knowledge to define the cross-product feature.

In this paper, we present the Tree, LR & DNN framework to achieve both memorization and generalization in one model. In the tree component, we used GBDT to transform the categorical features into a higher dimensional, sparse space. Meanwhile, inspired by continuous bag of words language models [4], we learn high dimensional embeddings for each player ID and skill ID in a fixed vocabulary and feed these embeddings into a forward neural network. The model architecture is illustrated in Figure 2.

The paper is organized as follows: a brief system overview is presented in Section 2. The GBDT, LR and DNN component will be explained in Section 3. In Section 4, we describe the three stages of this skill-target recommendation pipeline: data generation, model training, and model serving. In Section 5, we show the experiment results between different configurations. We also evaluated different turn-based combat strategies. Finally, Section 5 presents our conclusions and lessons learned.

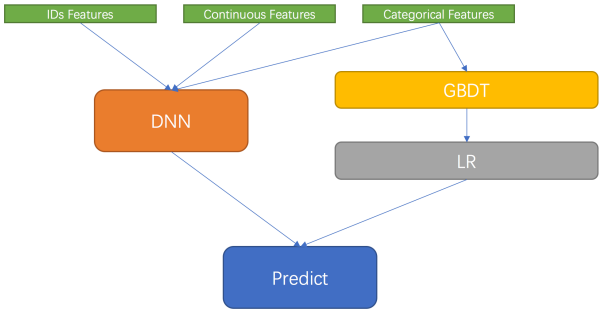


Figure 2, Tree, LR & DNN model architecture

2. System Overview

The overall structure of our skill-target recommendation system is illustrated in Figure 3. A query, which include each player's information and contextual features, is generated at the beginning of each round. There are total 20 players in the battle, and each player has 6 - 8 active skills, 2 passive skills. In our experiment, we only considered the active skills.

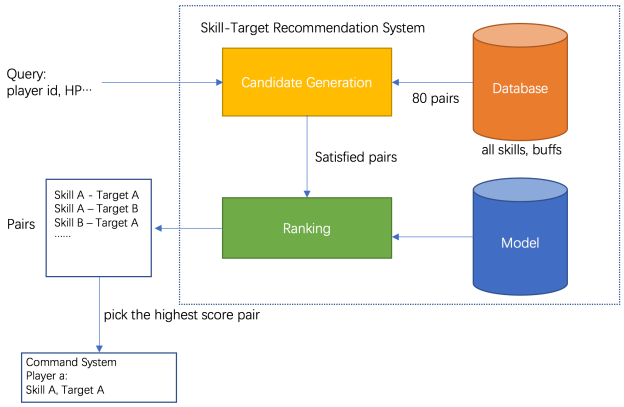


Figure 3, skill-target recommendation system architecture demonstrating the "funnel" where candidate skills are retrieved and ranked before presenting only a few to the user.

Once the candidate generation system received the query, it retrieves a small subset of skill-target pairs from the total 120 - 160 candidate pairs based on some rules, such as: the MP of the candidate skill should less or equal than the predict player's current MP. Some skills cannot be used for opponents or vice versa.

After reducing the candidate pairs, the ranking system split the features to the ranks all skill-target pairs by their scores. The scores are usually P(y|x), the probability of a player action label y given the features x, including player features (e.g., attack, defense, agility), contextual features (e.g., last attack target, last used skill). We implemented the ranking system by using GBDT, LR & DL framework.

3. GBDT, LR & Deep Learning

In this section, we split our GBDT, LR & Deep Learning (GLD) model to three components, and illustrated each component in three sub sections.

3.1 The GBDT Component

In the Tree Component, we used gradient boosted decision trees, as illustrated in Figure 4 to transform features into a higher dimensional, sparse space. Then trained a linear model (LR component) on these features.

Given GBDT training data $X = \{x_i\}_{i=1}^{N}$ and their labels $Y = \{y_i\}_{i=1}^{N}$ with $y_i \in \{0, 1\}$, the goal of this step is to choose a classification function F(x) to minimize the aggregation of some specified loss function $L(y_i, F(x_i))$:

$$F' = argmin_F \sum_{i=1}^{n} L(y_i, F(x_i))$$

Gradient boosting considers the function estimation F in an additive form:

$$F(x) = \sum_{m=1}^{T} f_m(x),$$

where T is the number of iteration. At each iteration, GBDT builds a regression tree that fits the residuals from the previous trees. The $\{f_m(x)\}$ are designed in an incremental fashion; at the m-th stage, the newly added function, f_m is chosen to optimize the aggregated loss while keeping $\{f_j\}_{j=1}^{m-1}$ fixed[6].

Once the GBDT training process finished, we fit an ensemble of these gradient boost trees on the training set. Then each leaf of each tree in the ensemble is assigned a fixed arbitrary feature index in a new feature space. These leaf indices are then encoded in a one-hot fashion. Each sample goes through the decisions of each tree of the ensemble and ends up in one leaf per tree. The sample is encoded by setting feature values for these leaves to 1 and the other feature values to 0. The resulting transformer has then learned a supervised, sparse, high-dimensional categorical embedding of the data. For instance, there are three subtrees in the Figure 3, where the first subtree has 4 leaves, the second 3 leaves and the third 2 leaves. If an example ends up in leaf 3 in the first subtree, leaf 2 in the second subtree and leaf 3 in the third subtree. The output will be the binary vector [0, 0, 1, 0, 0, 1, 0, 1, 0], which will be the input to LR component.

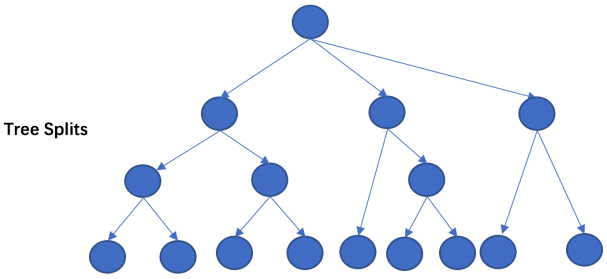


Figure 4, gradient boosted decision tree for transform the categorical features into a higher dimensional, sparse space.

3.2 The Logistic Regression (LR) Component

We expect the LR component (Figure 5) can learning the frequent co-occurrence of skills and targets in the historical data. The LR component learned a supervised, sparse, high-dimensional categorical embedding of the features generated from the tree component. And, the logistic regression model of the form:

$$y = \frac{1}{1 + e^{-W}}$$

where, $W = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$

y is the prediction, x = [x_1, x_2, ..., x_d] is a vector of features from the tree model, w = [w_1, w_2, ..., w_d] are the model parameters and b is the bias.

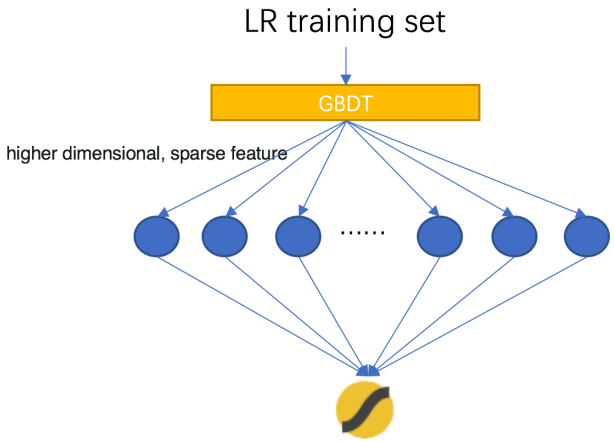


Figure 5, The structure of logistic regression model

3.3 The Deep Learning (DL) Component

We utilized DL component in our system to explores and transitivity the skills which have never or rarely used in the past. We fed hundreds of features in it with some preprocessing feature engineering. The feature roughly split evenly between categorical, continuous and IDs. Despite deep component alleviate the burden of engineering features by hand, the nature of our raw data does not easily lend itself to be input directly

into neural networks. We still expend considerable engineering resources transforming player and skills into useful features. Therefore, we build a vector embedding to every category type and normalized each continuous feature.

Embedding Categorical Features

In principle, a neural network can approximate any continuous function [14, 15] and piece wise continuous function [16]. However, it is not suitable to approximate arbitrary non-continuous functions as it assumes certain level of continuity in its general form [17]. The continuous nature of neural networks limits their applicability to categorical variables. Therefore, naively applying neural networks on structured data with integer representation for category variables does not work well. A common way to circumvent this problem is to use one-hot encoding, but it has two shortcomings: First when we have many high cardinality features one-hot encoding often results in an unrealistic amount of computational resource requirement. Second, it treats different values of categorical variables completely independent of each other and often ignores the informative relations between them. Therefore, we learn high dimensional embeddings for each skill in a fixed vocabulary and feed these embeddings into a feedforward neural network. A player's used skills history is represented by a variable-length sequence of sparse skill IDs which is mapped to a dense vector representation via the embeddings. The network requires fixed-sized dense inputs and simply averaging the embeddings performed best among several strategies (sum, component-wise max, etc.). Features are concatenated into a first layer, followed by several layers of fully connected Rectified Linear Units (ReLU) [6].

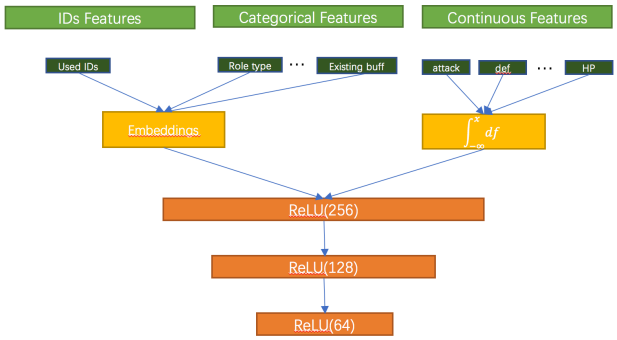


Figure 6, deep neural network helps to explores and transitivity the skills which have never or rarely used in the past.

Normalizing Continuous Feature

Since the range of values of raw data varies widely, in some machine learning algorithms, objective functions will not work properly without normalization. For example, the majority of classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance. Meanwhile, gradient descent converges much faster with feature scaling than without it. Moreover, the neural networks are notoriously sensitive to the scaling and distribution of their inputs [9]. We used cumulative distribution to transform a continuous feature x with distribution f to $\tilde{x}$ by scaling the values such that the feature is equally distributed in [0, 1). The formula is:

$$\tilde{x} = \int_{-\infty}^{x} df$$

These low-dimensional dense embedding vectors are then fed into the three hidden layers of a neural network in the forward pass. Specifically, each hidden layer performs the following computation:

$$a^{(l+1)} = f(W^{(l)}a^{(l)} + b^{(l)})$$

where l is the layer number and f is the activation function, often rectified linear units (ReLUs). a(l), b(l), and W (l) are the activations, bias, and model weights at l-th layer. We used the sigmoid function as the final activation function.

3.4 Training of Tree & Deep Model

We trained the LR and DNN model with two different ways: ensemble training and joint training. Joint training optimizes all parameters simultaneously by taking both the LR and DNN part as well as the weights of their sum into account at training time. The LR component and DNN component are combined using a weighted sum of their output log odds as the prediction, which is then fed to one common logistic loss function. Joint training is done by back-propagating the gradients from the output to both the wide and deep part of the model simultaneously using mini-batch stochastic optimization [3].

Ensemble training is: individual models are trained separately without knowing each other. The prediction score from both LR component and DL component are combined at inference time not at training time. The structures of both joint training and ensemble training are shown in Fig. 7.

In Fig. 7, The model's prediction for both ensemble training and joint training is:

Ensemble training:

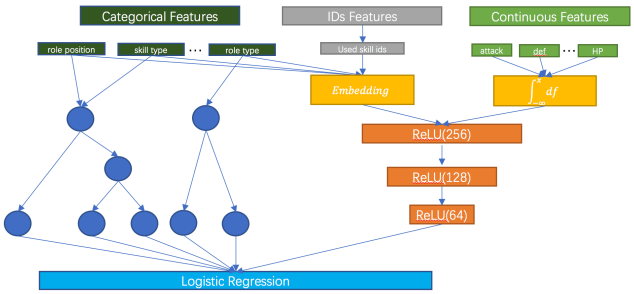
$$P(Y = 1 \mid x) = \sigma(W_{LR}^{T}GBDT(x) + b_0) + (1 - \alpha)(W_{DNN}^{T}a^{(lf)} + b_1)$$

Joint training:

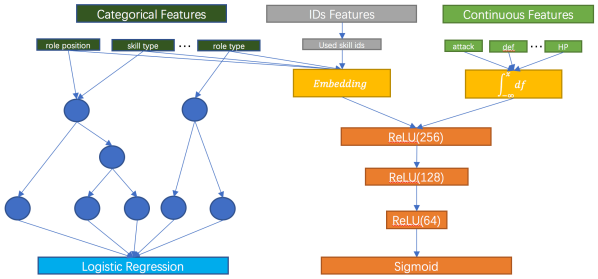
$$P(Y = 1 \mid x) = \sigma\left(W_{LR}^{T}GBDT(x) + W_{DNN}^{T}a^{(lf)} + b_0\right)$$

where Y is the binary class label, $\sigma(\cdot)$ is the sigmoid function, GBDT(x) is the tuple features generated from GBDT component. W_{LR} is the vector of all LR model weights, and W_{DNN} are the weights applied on the DNN's final activations $a^{(lf)}$. α is set to 0.5.

In the experiments, we implemented both ensemble and joint training, and the results were shown in Section 5.



(a)



(b)

Figure 7, (a) LR and DNN model's weights are training simultaneously for joint training. (b) LR and DNN model's weights are trained separately.

4. System Implementation

The turn-based battle recommendation pipeline consists of four stages: data generation, model training, validation module, and model serving as show in Figure 8.

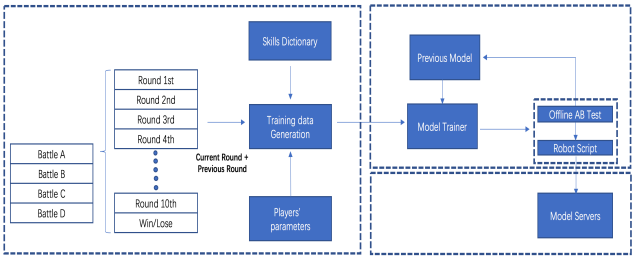


Figure 8: turn-based battle recommendation pipeline overview

4.1 Data and Label Generation

In the training data generation module, each player's parameter, current and previous round information, and the predict player's skill list were used to generate the training examples. The label is the battle's final result: 1 if the player win, and 0 otherwise. In order to reduce the noises in the positive examples, we only take the battles which the battle finished with the player win in 10 rounds as the positive example.

4.2 Model Training

The model structure we used in the experiment is shown in Figure 7. We put all categorical features as the input to the tree models. The tree component implements non-linear and tuple feature transformations and can be understood as a supervised feature encoding that converts a real-valued vector into a compact binary-valued vector. The LR component consists of the features which generated from the tree component. All the continuous, IDs, and categorical feature were normalized and embedding, then put into DL part of the model with 3 ReLU layers: 256->128->64. Since we applied both ensemble and joint training, the output of LR and DL component are combined at inference time (ensemble training) or at training time (joint training)

The GLD models are trained on over 200,000 battles. The batch size and epoch was set to 5000 and 20, respectively.

4.3 Robot Script Validation

Validation module includes the offline AB test system and robot validation system. To effectively validate the performances of different models, run a preliminary offline evaluation on historical data to iterate faster on new ideas is well-known practice [10]. The second module is robot validation system, which generates the random robot roles with the high parameters than the AI's. The strategy of robot validation system applies the random skill on the random target. We use the win-rate between AI strategy and robot script to validate the performance of different models.

4.4 Model Serving

Once the model is trained and tested, we load it into the online model servers. For each battle round, the servers receive a list of skill-target candidate pairs from the candidate generation system and player features to score each pair. After reducing the candidate pairs, the ranking system ranks all skill-target pairs by their scores. Then, the skill and target with the highest score was set to the command system.

5. Experiment Results

To evaluate the effectiveness of Tree & Deep learning, we compared our system with some other AI strategies to against a robot script. The robot script picks the random skill with the random target, but the robot roles have the higher parameters (more HP, more attack, etc.). Meanwhile, we also experiment our system with different configurations. Finally, our battle action predict system was productionized on SA 2. The serving performance has been given in sub section 3.

5.1 Positive up sampling

Class imbalance has been studied by many researchers and has been shown to have significant impact on the performance of the learned model [6]. Because most classification models in fact don't yield a binary decision, but rather a continuous decision value. Using the decision values, we can rank test samples, from 'almost certainly positive' to 'almost certainly negative'. Based on the decision value, we can always assign some cutoff that configures the classifier in such a way that a certain fraction of data is labeled as positive. Determining an appropriate threshold can be done via the model's ROC or PR curves. We can apply the decision threshold regardless of the balance used in the training set. Assuming the model is better than random, we can intuitively see that increasing the threshold for positive classification (which leads to less positive predictions) increases the model's precision and vice versa. Therefore, in this part, we investigate the use of positive up

sampling to test the influence of the class imbalance. We empirically experiment with different positive up sampling rate to test the prediction accuracy of the learned model. We vary the rate in {0.1, 0.2, 0.3, …, 0.8, 0.9}. The experiment result is shown in Figure 9.

From the result, we can see that the positive up sampling rate has significant effect on the performance of the trained model. The best performance is achieved with positive up sampling rate set to 0.6.

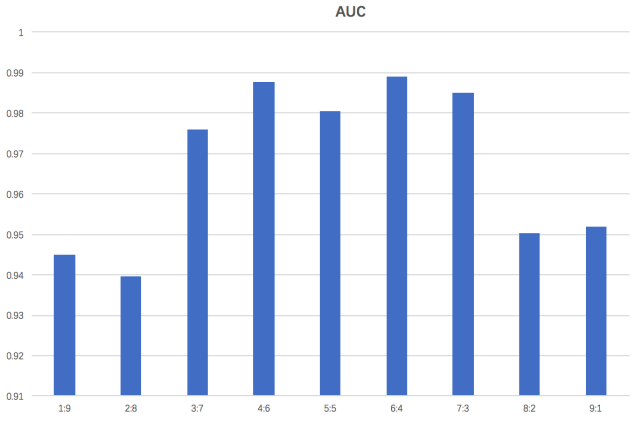


Figure 9, Fraction of the positive samples employed for learning

5.2 Experiments with GBDT Component

In the GBDT component, we fixed the number of estimators to 100, subsample to 1, learning rate to 0.1 and changed the max depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. The result shown in Table 1 that increasing the max depths, the model has the better performance on the training examples, but lower at 6, 7 max depths on the test examples. The reason for this phenomenon is overfitting: the higher depth will allow model to learn relations very specific to a particular sample.

max-depth	Train MAE	Test MAE
3 depths	0.1425	0.1562
4 depths	0.1379	0.1417
5 depths	0.1120	0.1205
6 depths	0.1088	0.1321
7 depths	0.0983	0.1379

Table 1: Experiments of GBDT component on max depth

5.2 Experiments with DNN Component Hidden Layers

Table 2 shows the results we used the same structure with different hidden layer configurations in DL module. These results show that with the increasing of both width and depth of hidden layers improves results. The trade-off, however, is server CPU time needed for inference. The configuration of a 512 hidden ReLU units followed by a 256 hidden ReLU units followed by a 128 hidden ReLU units gave us the best results while enabling us to stay within our serving CPU budget.

For the 512->256->128, we also tried the different normalization methods: min-max, standard, cumulative distribution, and none normalization. The cumulative distribution increased the win-rate by 4%, with min-max increased 3.2% and standard increased 2.7%.

Hidden layers	Train MAE	Test MAE
64 ReLU	0.0641	0.673
128 ReLU	0.0617	0.614
256 ReLU	0.0592	0.0622
512 ReLU	0.0562	0.0578
256ReLU->128ReLU->64ReLU	0.0508	0.0512
512ReLU->256ReLU->128ReLU	0.0477	0.0496

Table 2: Experiments of DNN component on different hidden layers

5.3 Experiments with different battle strategies

We used a robot validation system as the opponent to evaluate our GBDT, LR & Deep Learning model. The robot validation system generates 10 random robot roles with the higher parameters than the AI player roles, but applies the random skill with the random target. The value shown for each configuration ("win-rate") was obtained by considering both positive (win by AI) and negative (win by robots). As shown in Table 1, GBDT, LR & Deep Learning model had best win-rate compared with the random skill & random target strategy, LR-only model, GBDT-only model, GBDT+LR model, Deep Learning-only model and Reinforcement Learning (RL) model. For RL model, we defined the percentage of the sum(HPs) of the opponents as the reward function.

Model	Win rate (vs. Robot Script)
Random Skill & Random Target	7.21%
LR	16.24%
GBDT	20.12%
GBDT + LR	25.57%
DNN	28.51%
Reinforcement Learning	33.69%
GBDT + LR + DL (Joint Training)	46.73%
GBDT + LR + DL (Ensemble Training)	57.61%

Table 3, different models against to the robot script

Each experiment has been run 100,000 times. The main reason is that some AIs such as GBDT, LR can only learning the frequent co-occurrence of skills and targets in the historical data. On the other side, the RL AI tendency use attack kind skill rather than others. The shortage is that the assistant role or the healer tried to use normal attack skill instead of some assistant skills, such as speed up skill, control skill, etc. The result also shown that the ensemble training has the better performance than the joint training.

5.4 Serving Performance

Serving with high throughput and low latency is challenging with the high level of traffic faced by our commercial turn-based mobile game. At peak traffic, our recommender servers score over 100,000 skill-pair per second. With single threading, scoring all candidates in a single batch takes 3 ms on a CPU-only server.

6. Conclusion

We have described our battle action predict system on a turn-based mobile game SA 2. We treat our system as a recommendation ranking system in particular benefit from specialized features describing past player behavior with used skills.

We coalesced gradient boosted decision tree model, logistic regression and deep neural network-combine the benefits of generalization (explores the skills which have never or rarely used in the past) and memorization (based on learning the frequent co-occurrence of skills and targets in the historical data) to score the list of possible skill-action pairs. Moreover, the tree model can effectively transform the input features to the tuple features. Unlike some other AI strategies, our system does not need any prior knowledge. We presented the Tree & Wide & Deep learning framework to combine the strengths of three types of model.

In the experiments part, we test the resulting in terms of the performances against the robot script showed that the GLD model was significantly better than others.

7. Acknowledgements

We would like to thank many at Shanghai Luyou Network Technology, especially Dian Wu, Jun Qi for the robot script, and early feedback on the GLD model. We would also like to thank the Tencent K5 Cooperation Department, especially Qi Li, Zuoqiu Shen, Qi wang for comments on the manuscript.

8. References

[1] The StoneAge2. http://sq.qq.com/

[2] Orivol Vinyals, Timo Ewalds, Sergey barunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, et al. StarCraft II: A New Challenge for Reinforcement Learning, *DeepMind Lab. arXiv:1708.04782, 2017.*

[3] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, et al. Wide & Deep Learning for Recommender Systems, *DLRS 2016 Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.*

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean, Distributed Representations of Words and Phrases and their Compositionality, *arXiv:1310.4546, 2013*.

[5] Paul Covington, Jay Adams, Emre Sargin, Deep Neural Networks for YouTube Recommendations, *Proceedings of the 10th ACM Conference on Recommender Systems, ACM, New York, NY, USA, 2016.*

[6] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu∗, Tao Xu∗, Yanxin Shi∗, Antoine Atallah∗, Ralf Herbrich∗, et al, *Practical Lessons from Predicting Clicks on Ads at Facebook.*

[7] Makoto Ishihara, Taichi Miyazaki, Pujana Paliyawan, Chun Yin Chu, Tomohiro Harada, Ruck Thawonmas, Investigating Kinect-based Fighting Game AIs That Encourage Their players to Use Various Skills, *Consumer Electronics (GCCE), 2015 IEEE 4th Global Conference on.*

[8] Si Si, Huan Zhang, S. Sathiya Keerthi, Dhruv Mahajan, Inderjit S. Dhillon, Cho-Jui Hsieh, Gradient Boosted Decision Trees for High Dimensional Sparse Output, *Proceedings of the 34th International Conference on Machine Learning, PMLR 70:3182-3190, 2017.*

[9] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR, abs/1502.03167, 2015.*

[10] Alexandre Gilotte, Clément Calauzènes, Thomas Nedelec, Alexandre Abraham, Simon Dollé Criteo Research, Offline A/B testing for Recommender Systems, *arXiv:1801.07030v1, 2018.*

[11] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12:2121–2159, July 2011.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man´e, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vi´egas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[13] X. Amatriain. Building industrial-scale real-world recommender systems. In Proceedings of the Sixth ACM Conference on Recommender Systems, RecSys.

[14] George Cybenko, "Approximation by superpositions of a

sigmoidal function," 2, 303–314.

[15] Michael Nielsen, "Neural networks and deep learning,"

(Determination Press, 2015) Chap. 4.

[16] Bernardo Llanas, Sagrario Lantar´on, and Francisco J S´ainz, "Constructive approximation of discontinuous functions by neural networks," Neural Processing Letters 27, 209–226, 2008.

[17] Cheng Guo, Felix Berkhahn, Entity Embeddings of Categorical Variables, arXiv:1604.06737, 2016

[18] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In International Conference on Machine Learn- ing, pages 1329–1338, 2016.

[19] Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller,Playing Atari with Deep Reinforcement Learning, *DeepMind Lab. arXiv:1312.5602, 2013*

[20] M. Ishihara, T. Miyazaki, C.Y. Chu, T. Harada and R. Thawonmas, "Applying and improving Monte-Carlo Tree Search in a fighting game AI," ACM Int. Conf. Advances in Computer Entertainment Technology, 2016.

[21] S. Yoshida, M. Ishihara, T. Miyazaki, Y. Nakagawa, T. Harada and R. Thawonmas, "Application of Monte-Carlo Tree Search in a fighting game AI," IEEE Global Conf. Consumer Electronics, 2016.

[22] J. E. Laird, "Using a computer game to develop advanced ai," Com- puter, vol. 34, no. 7, pp. 70–75, Jul. 2001. [Online]. Available: http://dx. doi.org/10.1109/2.933506

[23] M. Buro and T. M. Furtak, “Rts games and real-time ai research,” in Proc. Behavior Representation Model. Simul. Conf., 2004, pp. 51–58.

[24] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In Proceedings of the 12th International Conference on Machine Learning (ICML 1995), pages 30–37. Morgan Kaufmann, 1995.

[25] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An evaluation platform for general agents. J. Artif. Intell. Res.(JAIR), 47:253–279, 2013.