AI agent for a board game Dicewars

Adam Kučera <xkucer95@stud.fit.vutbr.cz>

January 2, 2021

1 Intro

The goal of this work was to implement an artificial intelligence for a board game called Dicewars. I've designed agend which combines multiple approaches in order to yield the best playing strategy given time and hardware restrictions. Traditional techniques for a board games agents are state space search algorithms. However, there are two big problems in Dicewars with this approach. First is the stochastic nature – dice throws determine the evolution of the game. This uncertainty can be modeled with probabilistic algorithms, but it contributes a lot to the exponential "explosion" of the state space. And that is the second problem itself. The size of the state space in Dicewars is too big for being modeled as a whole. There are also Monte-Carlo approaches, which are good for such huge state spaces, but the faster, the less optimal they are. On the other hand, the most modern and successful techniques for playing games are based on machine learning, namely convolutional neural networks, e.g. AlphaGo [3]. But despite being extremely successful, they are also extremely computationally demanding and must be run on many GPUs in parallel. Because the proposed agend needs to be able to run on a single CPU core, I've implemented a much simpler machine learning models. After being trained on abstract high level features, I finally added a very pruned state space search to improve the decision making even more.

2 Baseline

Very important in order to evaluate improvements and progress of the agent development, the so called baseline agent is required. I've implemented a trivial but not stupid agent, who just performs the best attacks in terms of success probability of conquering the target area (pre-calculated in table), but only if source area power is greater or equal to target area power. All the training data for machine learning models has been collected from games of provided AIs and this baseline agent.

3 Solution

The overall architecture of the agent consist of 3 main components, as shown on figure 1.

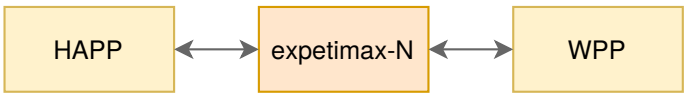

Figure 1: Proposed AI agent

3.1 Hold Area Probability Predictor (HAPP)

The first part is a machine learning model, which estimates a probability of holding or keeping an area during other players turns, to which I gave the working name HAPP. This model basically estimates, how "strong" the area is given its state, which is descibed by the following feature vector:

1. probability of survival if all players would attack it
2. ratio of dice power of player to enemy in the neighborhood
3. number of areas owned by enemies in the neighborhood
4. number of areas owned by player in the neighborhood
5. number of unique enemies in the neighborhood
6. ratio of players score to number of all areas (i.e. winning score)

It is noticeable that all features except the last one are based on the neighborhood of the area. The last one is a global feature, which is used to add "confidence" of making move when the player is strong enough and likely to have a good reserve dices.

First, I tried a simple logistic regression, but despite giving a good results on a cross-validation set, the accuracy and loss curves were very noisy with huge jumps up and down. To me, this was an indication that data might not be perfectly linearly separable in the given space. Therefore I've created simple neural network model with one hidden layer of the same size as the input, as shown in the figure 2.

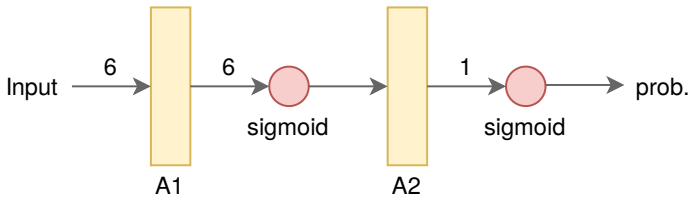

Figure 2: HAPP model

The final accuracy wasn't signifiantly highger than for the logistic model, but curves were much more monotonic this time, as can be seen in the plot 3.

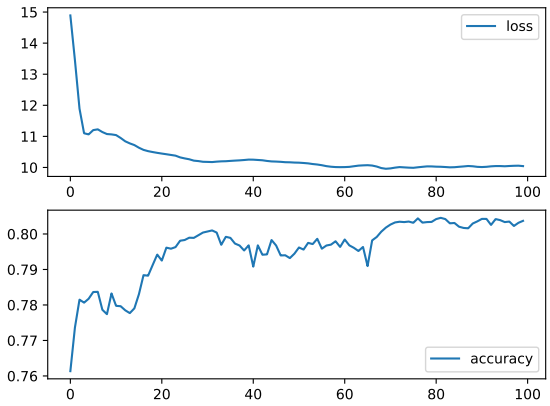

Figure 3: HAPP training

The training data has been pruned in order to get equal apriori class probability (50/50), so the training bias would be minimized. This step was very important in order to achieve reasonable accuracy. The final accuracy on the cross-validation set of size 20000 samples after 100 epochs of training on 87194 samples was 80.46%. For a general binary classification problem, this might be poor, but taking into account the highly stochastic nature of the process, such accuracy is very solid. Batch size of 64 and Adam optimizer [2] with learning rate $10e^{-4}$ have been used.

3.2 Win Probability Predictor (WPP)

The second machine learning model the agent uses is predictor of victory, to which I gave the working name WPP. This model estimates probability of winning the game by a player given the game state. I've gathered game states 5 turns before game end as a training data for this model. I've engineered the game state descriptor as the following feature vector:

1. ratio of players score to number of all areas (i.e. winning score)
2. ratio of length of player border to borders of other players
3. ratio of players dice power on borders to his total dice power
4. ratio of number of areas owned by player to number of all areas
5. mean of attribute 1. in area descriptor for players areas
6. mean of attribute 2. in area descriptor for players areas

It might be good idea to create a multiclass neural network with sofmax at it terminal but I've found out that for the sake of perfomance (the model is used in the state space search), the simple logistic regression is good enought. The resulting accuracy was 99.25%, which is satisfying. Training data was harder to obtain, so the training set was only 7418 and cross-validation set only 2000 samples. It has been collected from games of 4 players, so apriori probability of victory is 25%. Training process took 200 epochs, batch size of 32 and Adam optimizer with learning rate $10e^{-4}$ have been used. The loss and accuracy curves show almost monotonic improvements (plot 4), as the probabilistic hyperplane was enough to separate the data.

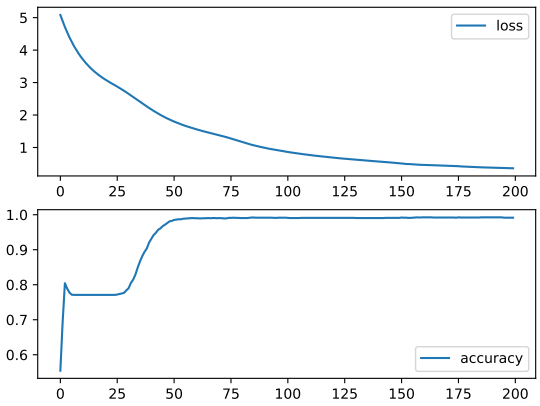

Figure 4: WPP training

3.3 Expectimax-N

Finally, I've implemented the state space search algorithm. The chosen one was a multiplayer extension of probabilistic version of minimax algorithm [1]. This extension is called Expectimax-N [1]. The algorithm finds optimal solution to the given depth, so called equlibrium state, when all players play in best way possible (for them). However, they might change their strategy to handicap certain player without changing their scores. As Dicewars include probability, this algorithm is appropriate decision, but the state space must be significantly reduced in order to find solution in some feasible amount of time. The reduction is done by HAPP. All possible attacks where source area power is greater or equal to target area power, are simulated as successful and both source and target areas are than fed into HAPP. The final set of best attacks is constructed as shown in equation 3.3, where p_s, p_t and p_a are probabilities of holding source, holding target and winning the battle.

$$A = \{attack \mid p_s \cdot p_t \cdot p_a > 0.1\} \tag{1}$$

The threshold of 0.1 is not empirical, but derived from joint probability – the minimum confidence for an attack to be considered possibly acceptable

[1] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction

is when HAPP is >50% sure both source and target areas will survive other players turns and at the same time, attack success probability is >50% too. Finally, the confidence of HAPP is roughly 80%, so this is incorporated into the final formula: $0.5 \cdot 0.5 \cdot 0.5 \cdot 0.8 = 0.1$.

After state space reduction, leaves of expecitmax-N graph have to be evaluated. This can make big difference in results as the rest of algorithm decides according to this evaluation. As the heuristics for this evaluation I've used the mentioned WPP model. Because the number of attacks in a single turn of one player is variable and makes big difference in the quality of the turn in overall, I found it very good to simulate as long attack sequences as possible. I've created an adaptive mechanism, which when enought time, simulates up to 4 moves of the player and then 2 moves of each enemy player up to depth of 10 game tree levels in total. If time left is less than 6 seconds, it simulates only 2 moves of the player, totaling with depth of 8. If time left falls bellow 2 seconds, no state space search is perfomed and the best attack from HAPP is used instead (if any).

4 Final implementation

As the game has been implemented in Python, I've implemented most of the functionality, including feature vectors generation and Expectimax-N in pure Python. I've used numpy where it was reasonable and for both machine learning models, I've used the PyTorch library. I've implemented 3 version of the method `ai_turn()`, which is called from the client. The names methods are the following:

- `ai_turn_impl_1()` – the baseline
- `ai_turn_impl_2()` – performs best attacks from HAPP (if any)
- `ai_turn_impl_3()` – the final state space search from 3.3

5 Evaluation

I've performed all tests on Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz. All 3 implementations were tested in games of 4 players with provided AIs, namely the `dt.ste`, `dt.sdc`, `dt.wpm_c`, `dt.rand` and `xlogin00`. Each test

consisted of 200 boards played by my agent `xkucer95`, each for 4 games, so 800 matches in total. Tests were performed with the command:
`python3 scripts/dicewars-tournament.py -r -g 4 -n 200 -ai-under-test xkucer95`

5.1 Results

Final evaluation results, i.e. win-ratios of implemented agents are being shown in the table 1. Both HAPP-only and the final agent outperformed all provided AIs by a significant margin.

AI agent	**win-ratio**
Baseline	28.5 %
HAPP-only	43.6 %
Final AI	47.7 %

Table 1: Evaluation of agents

HAPP model has turned out to be a very strong heuristic on its own, so the state space search din't improve the agents performance as much as one might expect, although it obviously did. The expectimax-N impact on the speed is notable, however the adaptive approach prevents the final agent from vasting time by "thinking" while he still can act with relatively high confidence.

References

[1] Sten Andler. Expectimax-n: A modification of expectimax algorithm to solve multiplayer stochastic game. 2009.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, jan 2016.