Solution Report

Tomasz Garbus & Łukasz Raszkiewicz

May 2018

1 Team

Members:

- Tomasz Garbus (tgarbus)
- Łukasz Raszkiewicz (lraszkiewicz)

2 Cross-validation

We tested our solutions with 4-fold cross-validation. Instead of splitting the set of training games into 4 parts, we split the set of 400 training decks. As the set of game results passed to the model in each fold of cross-validation, we had selected those games in which decks used by both players were in the set of the 300 decks.
And we had a really pretty progress bar.

3 Solutions

3.1 K Nearest Neighbors

Our initial solution was to use a kNN classifier. We defined a function of distance between two decks:
$D(d_1, d_2) = 30 - |d_1 \cap d_2| +$ `(if d1.hero != d2.hero then 10 else 0)`
For a given deck d, using this function, we found the closest decks in the set of training decks, and used their results to try to predict d's winrate. We did not use the results of decks for which the distance was larger or equal to 30, even if that meant using less than k decks. We tried different variations of how to approach that, including:

- different values of k
- predicting a deck's results against each of the decks in the training set, and then getting the winrate
  - predicting game results as 0/1 or as a probability to win
  - weighting the different predicted results by reverse distance to d
  - using the final HP of the winning hero from training games to determine if a game was one-sided or not, increasing the winning probability
- predicting a deck's winrate as an average of its closest decks (weighted or not weighted by reverse distance to d)
- using results only of the (bot, deck) pair that we are currently trying to predict, or also using results of different bots with the same deck

Details on some of the points above can be found later in this report.

Despite trying various combinations of the parameters above, we had little success with kNN. The scores on the non-final test set on the submission system varied from 9.77 to about 11.

3.2 Predicting winrate vs individual games

In both approaches – neural nets and kNN – we tried predicting overall winrate at once as well as result of each individual game. Predicting the winrate was significantly faster and performed just as well (if not better) on our local validation. However, on submission system, none of the submissions generated by predicting the winrate was able to reach RMSE < 9.0.

After concluding that we want to predict results of individual games, we have tried two ways of calculating the winrate:

- $WR(d) = \frac{\sum_{g \in G_d} round(p(g))}{|G_d|}$
- $WR(d) = \frac{\sum_{g \in G_d} p(g)}{|G_d|}$

where G_d is the set of games played by player $d = (bot, deck)$ and $p(g)$ is predicted probability that player d wins game g. Not surprisingly, the second equation gave better results.

3.3 Neural Net

We have implemented a neural net in TensorFlow. It is configurable with regard to:

- neurons count in each layer
- learning rate
- batch-size
- number of epochs
- dropout rate (applied after each hidden layer)
- learning rate decay period (`‘lr_decay_time = N‘` indicates that learning rate is divided by 2 after N epochs)

3.3.1 Input format

A game is presented to the model as a concatenation of inputs for both players. Input for a single player consists of:

- bot, encoded as one-hot
- hero, encoded as one-hot
- $\sum_{i=1}^{30} c_i$, where c_i is i-th card in the player's deck encoded as one-hot

3.3.2 Output format

Output from the network is a single float in range [0..1] – probability that player 0 wins. Obviously, for training games, probability is equal to either 0 or 1 (with one exception – see section *input enhancement*).

3.3.3 Input enhancement

We have tried two ways of enhancing the input data:

- Instead of supplying probability for training games as 0 or 1, we defined it as:

  $$p(\text{game}) = \begin{cases} 0.5 + \frac{\text{game['winner\_hp']}}{60} & \text{game['winner'] == 0} \\ 0.5 - \frac{\text{game['winner\_hp']}}{60} & \text{game['winner'] == 1} \end{cases}$$

  where `game['winner_hp']` is the winner's health before last turn.

  Unfortunately, this approach did not improve our results. As labels were closer to 0.5, the network learned to be "less sure" about its predictions and the variance of predicted winrates was very low – almost all predicted winrates were in range [0.4..0.6].

- We have downloaded stats of each card from `http://hearthstoneapi.com/` (or set them to 0 if not applicable) – cost, health and attack. Then we have sorted the values of each type, normalized them to range [0..1] and appended them to input vector.

  The effect on results was rather negative. The network was more sure about its predictions (variance was higher), but also RMSE was higher. This may be because stats health and attack only make sense for minions.

3.3.4 Special case: linear regression (final submission)

Our final submission is generated with **linear regression**. To predict the winrate of a given pair (bot, deck), we predict the score of each game separately and compute the mean of results.
Parameters:

- loss function: log loss
- activation: sigmoid
- learning rate: 0.2
- lr decay: $lr := \frac{lr}{2}$ after $5 \cdot 10^3$ epochs
- batch size: 256
- number of epochs: 10^4

With these parameters, we obtained score 7.61 on the final leaderboard.

4 Implementation

Neural net and linear regression were implemented using TensorFlow. In our efforts to enhance input data we also used `http://hearthstoneapi.com/`.
All source code is available at `https://gitlab.com/tomasz.garbus1/sus_hearthstone`.

5 Work division

Tomasz Garbus:

- neural net and linear regression
- downloading cards stats from API

Łukasz Raszkiewicz:

- kNN
- testing and cross-validation
- parsing jsons and csvs (in particular extracting winners HP)
- maintaining code quality