

Evaluation of Hearthstone Game States With Neural Networks and Sparse Autoencoding

Jan Jakubik

Wrocław University of Science and Technology
Department of Computational Intelligence
Wrocław, Poland
Email: jan.jakubik@pwr.edu.pl

Abstract—In this paper, an approach to evaluating game states of a collectible card game Hearthstone is described. A deep neural network is employed to predict the probability of winning associated with a given game state. Encoding the game state as an input vector is based on another neural network, an autoencoder with a sparsity-inducing loss. The autoencoder encodes minion information in a sparse-like fashion so that it can be efficiently aggregated. Additionally, the model is regularized by decorrelation of hidden layer neuron activations, a concept derived from an existing regularizing method DeCov. The approach was developed for AAIA'17 data mining competition "Helping AI to play Hearthstone" and achieved 5th place out of 188 submissions.

I. INTRODUCTION

VIDEOGAME AI is one of the most well-known applications of Artificial Intelligence methods in software development. Designing challenging, smart and believable opponents has always been an important goal for videogame developers. Implementation of intelligent actors in games requires development of efficient methods for searching large state spaces and designing game-specific heuristics for state evaluation.

Recently, a breakthrough in the domain of board games achieved by the AlphaGo [1] project has demonstrated the potential of machine learning methods, specifically deep neural networks [2], in game AI. Coupled with a Monte Carlo tree search approach [3], a combination of neural networks for move policy and game state evaluation has achieved results previously thought to be at least a decade of research away and was shown capable to defeat top-level human players. The results are promising for multiple types of games, as Monte Carlo tree search approach is general enough to cover varying types of gameplay, including card games.

This paper describes a solution to the data mining challenge competition organized within the framework of the 12th International Symposium on Advances in Artificial Intelligence and Applications. The goal of the challenge was to evaluate game states of the collectible card game Hearthstone, using machine learning algorithms, given a training sample of contextless game states with a single win/loss variable to predict. Our solution is based on a combination of autoencoder neural network for game state encoding and a deep neural network for the actual game result prediction.

II. CHALLENGE DESCRIPTION

Hearthstone is a collectible card game developed by Blizzard Entertainment in which two players, represented by heroes chosen from a pool of 9 character classes, fight each other using minion, spell and weapon cards. Additionally, each hero has access to a "hero power" which can be used once per turn. Possible plays are limited by crystals called "mana". A player starts the game with a single mana crystal, gains one mana crystal per turn and can use their mana crystals once per turn to pay the cost associated with playing a card.

Minions persist on the game board until destroyed, can attack the opposing player or other minions once per turn, and can have various special properties. Minions are positioned on the game board in a single line for the player and another for the opponent; up to 7 minions can be played on each side and adjacency can be relevant to the functioning of certain cards.

Spells are cast by the player, affect the game state in a particular way, and leave the game afterward. Most of spells do not persist on the game board after resolving their effect.

Weapons can be equipped by the player's hero, are persistent, and allow the hero to attack, similarly to the way minions do. However, weapons have a limited durability, which goes down by 1 with each attack, effectively limiting the number of times they can be used. Moreover, only one weapon can be equipped at a time.

The goal of the game is to bring down the opposing player's health points (HP) to 0 or below, with both players starting at 30 HP. Minions are particularly important for this purpose due to their persistence on the game board and the ability to attack every turn.

The challenge data consists of contextless snapshots of game states during the player turn. For training data, a single variable indicating whether the game was won or lost by the player is provided. The data was created using simulations of games between two AI, driven by a Monte Carlo tree search approach. The simulations only use cards from the original card set of Hearthstone, released in 2014. Provided datasets are divided as follows:

- initial training set: 4 data chunks of 500000 game states each
- initial test set: 1250000 game states; later became available as training set

TABLE I
PROPERTIES OF THE PLAYER AND THE OPPONENT

Property	Meaning
deck_count	Number of cards in deck
played_minions_count	Number of minions in play
fatigue_damage	Takes this much damage with next draw
hand_count	Number of cards in hand
crystals_all	Number of mana crystals
spell_dmg_bonus	Damage added to damaging spells
crystals_current	Mana crystals still available this turn
weapon_durability	Uses of equipped weapon left
armor	Additional HP which can go above 30
hp	Health points, maximum 30
attack	Deals this much damage on attack
hero_card_id	One of 9 hero classes
special_skill_used	Hero power was used this turn

- final test set: 750000 game states

The goal of the challenge was to provide real number evaluations of game states present in the final test set, quality of which would be measured by area under the ROC curve (AUC).

III. GAME STATE ENCODING

In the following section, the term "player" will be used in reference to the player from whose perspective the games are observed, and the term "opponent" will refer to the second player.

The key properties of a single game state consists of turn number, player stats, opponent stats, up to 7 player minions, up to 7 opponent minions and up to 10 cards in player's hand.

Representing these variables so that they can serve as an input to a neural network is non-trivial due to the varying number of minions. Basic information about a single minion can be expressed as a numerical vector. However, a concatenation of seven vectors into a single "board vector" representing one side poses multiple problems. Firstly, this representation does not guarantee equivalent or even similar results for equivalent board states (i.e., shuffling minion positions). Secondly, samples with minions present on all positions are rare in the training set. Usually, only the first few positions are occupied.

Proposed solution is based on using a sparse autoencoder to encode minion data. While autoencoders are typically a dimensionality reduction method, sparse coding aims to detect patterns and improve aggregation of data. E.g., given a set of objects which form clusters in the data space, the simplest form of sparse coding is a dictionary approach in which each object is encoded as a one-hot vector that contains the object's cluster assignment. A sum of such sparse vectors contains information of how much objects of each type there are in the dataset. More complex dictionary encoding methods exist [4], and neural network encoding with sparsity constraints can be viewed as a non-linear extension of them [5].

In our approach, we encode information about each player's minion sparsely and then sum them into a single vector

TABLE II
PROPERTIES OF A MINION

Property	Meaning
hp_max	Initial HP, cannot be healed above this value
charge	Can attack on the turn it is played
frozen	Cannot attack until next turn
taunt	Allies (without taunt) cannot be attacked
poisonous	Kills any minion it damages
freezing	Attacked enemies become frozen
forgetful	50% chance to attack wrong enemy
crystals_cost	Cost to play in mana
shield	Negates first instance of damage dealt to it
attack	Deals this much damage on attack
hp_current	Current HP
windfury	Can attack twice per turn
stealth	Cannot be targeted by spells and attacks
id	Unique ID number of a card
can_attack	Can still attack this turn

representing player minions. The same is done to opponent minions. The input vector is a concatenation of these minion vectors and the remaining information about the game state.

The training vector takes a form shown below (1):

$$turn|player|opponent|\sum_{i=1}^7 p_i|\sum_{i=1}^7 o_i|hand \quad (1)$$

where $x|y$ denotes concatenation of vectors. $turn$ is the turn number, $player$ is all of the available player information and $opponent$ is all of the available opponent information. p_i is a vector of information about i -th player minion encoded by the autoencoder network described in Section IV, and o_i is a vector of information about i -th enemy minion encoded using the same network.

In order to build $player$ and $opponent$ vectors, hero class information is encoded in a 9 element one-hot vector. All remaining binary and numerical properties (Table I) are treated as real numbers. Numerical properties are normalized to have values in $[0,1]$ range.

The input to the autoencoder network is a 17-element vector, where first 14 elements are all numerical and binary properties of a minion (Table II), with the exception of the "id" property (unique identification number). The remaining elements of the vector are used for information about certain unique abilities. 15-th is a unique board-buffing ability (set to 1 for Stormwind Champion, 0.5 for Raid Leader, 0 for other minions), 16-th is a unique adjacent minion buffing ability (set to 1 for Flametongue Totem, 0.33 for Dire Wolf Alpha, 0 for other minions) and 17th element is set to 1 only for Healing Totem to represent its unique healing ability.

Vector $hand$ contains information about player's hand and uses dictionary encoding. Its dimensionality is equivalent to the number of unique cards in the training set, and i -th element indicates the number of times i -th card occurs in player's hand.

Cards which appear in the player's hand in the test set, but not the training set are ignored.

IV. SPARSE AUTOENCODER

Autoencoder [7] is a network that attempts to recover input data from a hidden layer representation, as seen in equations (2-4):

$$h_i = \sigma_e(W_e x_i + b_e) \quad (2)$$

$$x'_i = W_d h_i + b_d \quad (3)$$

$$RE(X) = \sum_{i=1}^n \|x_i - x'_i\|_2^2 \quad (4)$$

where x_i is the i th input vector, h_i is its encoded hidden layer representation, and x' is the input reconstruction. $\|\dots\|_2$ denotes L2 norm. σ_e is a log-sigmoid activation function. Weight matrices W_e , W_d and bias vectors b_e , b_d of the model are trained to minimize reconstruction error $RE(X)$ using stochastic gradient descent.

It is possible to encourage a sparse representation within an autoencoding network adjusting the loss function, as seen in equation (5):

$$SparsePenalty(X) = \sum_{i=1}^m \left(\rho \log \frac{\rho}{\hat{\rho}_i} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_i} \right) \quad (5)$$

where $\hat{\rho}_i$ is the average activation of i -th output in the encoding layer (m is the number of neurons in the layer). Parameter ρ is a low positive value, which encourages low average activations. In practice, this causes the network to encode information in a sparse-like way, i.e., some "active" neurons have significantly higher activations than others. The "inactive" neurons do not have zero activations if rectified linear units are not used, so the representation is not sparse in the strict sense.

Overall loss function of a sparse autoencoder can be defined as (6):

$$L(X) = RE(X) + \lambda_0 SparsePenalty(X) \quad (6)$$

where λ_0 is a hyperparameter.

V. REGULARIZATION OF THE PREDICTION MODEL

The prediction model is a deep neural network with four hidden layers. The detailed network parameters and the learning process are described in Section VI. In this section, regularization of the model is described in detail. As training set contains data from a set of simulations disjoint from the set of simulations used to create the test data, it is easy to overfit any model by learning to identify specific games, so regularization becomes a key issue.

For regularization, standard L2 norm penalty is applied to weight matrices. In addition, the correlation between outputs in hidden layers is explicitly punished. This is inspired by DeCov

[6], a recently proposed regularization method that adds a loss based on the covariance between outputs in a hidden layer (7):

$$DeCov(H_i) = \|Cov(H_i) - \text{diag}(Cov(H_i))\|_F^2 \quad (7)$$

where $\|\dots\|_F$ is the Frobenius norm and $Cov(H_i)$ denotes the covariance matrix of outputs in i -th layer, i.e., element in j -th row, k -th column is the covariance (7) between activations of j -th and k -th hidden unit in that layer. Diagonal of the covariance matrix is subtracted from it since the diagonal elements correspond to standard deviations of particular units.

Covariance between outputs h_i , h_j with means μ_i , μ_j and standard deviations σ_i , σ_j is given by equation (8):

$$\text{cov}(h_j, h_k) = (\sigma_j - \mu_j)(\sigma_k - \mu_k) \quad (8)$$

And the relation between correlation and covariance is (9):

$$\text{cov}(h_j, h_k) = \sigma_j \sigma_k \text{corr}(h_j, h_k) \quad (9)$$

This means DeCov punishes both correlation and high standard deviation in activations for any neuron that has a non-zero correlation with another neuron. The authors of DeCov mention this issue and remark the effects of a loss dependent on standard deviations are similar to L2 regularization. However, a similar regularization penalty term which does not punish standard deviations can be used (10):

$$DeCorr(H_i) = \|Corr(H_i)\|_F^2 \quad (10)$$

where $Corr(H_i)$ denotes a correlation matrix of H_i , analogous to $Cov(H_i)$. Full loss function (11) is then formulated as:

$$L(X, Y) = MSE(X, Y) + \lambda_1 \sum_i DeCorr(H_i) + \lambda_2 \sum_i \|W_i\|_F^2 \quad (11)$$

where $MSE(X, Y)$ denotes mean square error given inputs X and target outputs Y , H_i denotes outputs of the i -th hidden layer, W_i is the weight matrix of the i -th hidden layer. Unlike the original formulation of DeCov, this loss function allows us to control respective regularizing effects of the L2 weight penalty and decorrelation through the hyperparameters λ_1 and λ_2 .

VI. EXPERIMENTAL SETUP AND RESULTS

The neural network was implemented using Theano [8] python library which handles both gradient calculation and GPU computation.

The tuning process which led to choosing parameters reported below was based on four original chunks of training data. We trained on three chunks and evaluated on the fourth, repeating the process four times with a different chunk for evaluation. In preliminary tests, we noticed this approach achieved worse performance (measured by AUC) compared to a setup in which all chunks are mixed together and then 75% of data is selected for training. This led us to hypothesize that contents of a single chunk may be sharing similarities which

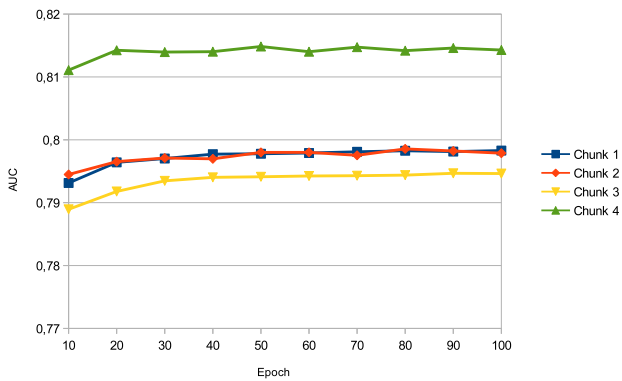


Fig. 1. Observed performance during training on the initial training set (one chunk used for verification, three remaining for training).

TABLE III
RESULTS OF THE COMPETITION - TOP 10 SUBMISSIONS

Team	AUC
iwannabetheverybest	0.80185414
hieuvq	0.799223
johnpateha	0.79895085
vz	0.79733467
jj	0.79706854
karek	0.79684963
podludek	0.79657371
akumpan	0.79653594
iran-amin	0.79636944
basakesin	0.79617152

would not be present between training and test data and the better result achieved with mixed chunks may be artificially inflated. Therefore, we decided to avoid mixing chunks during the parameter tuning. The results of training the tuned network on the initial training set can be seen in Fig. 1.

The size of the minion representation returned by the autoencoder network was set to 20, which resulted in overall size of the vector representing a game state equal to 182. Autoencoder was trained with hyperparameters $\lambda_0 = 10$, $\rho = 0.01$, for 100 epochs on all minion data from both training and test sets.

The neural network used for result prediction was 5 layers deep, with hidden layers of size 128, 64, 32, 16 and a single output neuron. Hyperbolic tangent activation was used in hidden layers and logistic sigmoid in the output layer. The weight of L2 penalty term λ_2 was set to 0.5, while the weight of DeCorr penalty term λ_1 was 0.1.

The network was trained using adaptive gradient [9] (initial learning rate 0.05) for 100 epochs, although the results were saved for 20-th, 40-th, 60-th and 80-th epoch. All results were

uploaded and the one with the best performance on the test set (60 epochs) was chosen as the final submission. The final results of the competition are shown in Table III.

VII. CONCLUSIONS

As a submission to AAIA '17 data mining competition, we proposed a neural network approach to evaluation of game states for the collectible card game Hearthstone. Sparse autoencoder was used to encode minion data, and a deep neural network was employed to obtain the evaluation of a game state. Additionally, we regularized the network with a novel approach, based on adjusting an existing regularization method DeCov to allow more control over the training process through parametrization. The solution placed 5th on the final leaderboard of the challenge.

The main weakness of our method was not accounting for unique effects which are not expressed through numerical properties and appear in the test, but not training data. An example of such effect is the Northshire Cleric card, a minion which allows its owner to draw a card whenever a minion is healed. It is a complex interaction which is not expressed in any way in a contextless game state description. Evaluation of such special abilities and their effect on gameplay cannot be easily achieved through a machine learning model alone. It would require either an extended set of training data or employing additional Monte Carlo simulations in the process of training and evaluation of the neural network.

ACKNOWLEDGEMENTS

We would like to thank Silver Bullet Solutions and Knowledge Pit for providing the simulation data and a platform for the competition.

REFERENCES

- [1] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489.
- [2] Deng, Li. "A tutorial survey of architectures, algorithms, and applications for deep learning." *APSIPA Transactions on Signal and Information Processing* 3 (2014): e2.
- [3] Brugmann, Bernd. "Monte carlo go." Syracuse, NY: Technical report, Physics Department, Syracuse University, (1993).
- [4] Y. Vaizman, B. McFee, and G. Lanckriet, "Codebook based audio feature representation for music information retrieval," *IEEE/ACM Transactions on Acoustics, Speech and Signal Processing*, vol. 22, no. 10, pp. 1483-1493, 2014.
- [5] J. Nam, J. Herrera, M. Slaney, and J. Smith, "Learning sparse feature representations for music annotation and retrieval," in *Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR)*, pp. 565-570, 2012.
- [6] Cogswell, Michael, et al. "Reducing overfitting in deep networks by decorrelating representations." arXiv:1511.06068 (2015).
- [7] Ng, Andrew. "Sparse autoencoder." CS294A Lecture notes 72.2011 (2011): 1-19.
- [8] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions", arXiv:1605.02688 (2016).
- [9] Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of Machine Learning Research* 12 (2011): 2121-2159.