

UNIVERSITY OF KENTUCKY
LEWIS HONORS COLLEGE

Deep State-Value Estimation for Long-Term Planning: A Generic Strategy Game Test Case

by

Aaron J. Moseley

AN UNDERGRADUATE THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DISTINCTION OF UNIVERSITY HONORS

Approved by the following faculty advisors:

Dr. Jerzy W. Jaromczyk

Dr. Neil F. Moore

Department of Computer Science

LEXINGTON, KENTUCKY
May 2024



ABSTRACT

Turn-based strategy games have long been used as a testing ground for artificial intelligence agents on general-purpose resource management and long-term planning tasks. To be effective in this environment, agents must be able to determine their priorities over an extended period of time while managing the minutiae of each turn. Much previous work has focused on tree-based search approaches that analyze many possible future states with predetermined utility values for each resource type. More recently, research has investigated the ability of agents to use reinforcement learning and deep models to discover strategies and improve over time. However, most of these methods involve initial training with established strategies in existing games. Our approach, called Deep-State Learning (DSL), successfully combines traditional tree-search algorithms with deep models in a generic strategy game without requiring knowledge of tactics that have been previously created. Experimental evaluations on a generic strategy game show that our approach outperforms tree-search strategies using a standard utility function by up to 15% and drastically surpasses greedy and random baselines.

Code publicly available at <https://github.com/AaronMoseley/HON491-DSL>.

1. INTRODUCTION

Strategy games have historically stood out as useful analogs for real-world resource management and long-term planning tasks. These games offer intricate environments where players must navigate limited assets and identify optimal approaches to secure victory. Past research in artificial intelligence has rightly focused on these games, from classical examples like Deep Blue¹ to modern applications such as AlphaGo.²

At the core of this research lie hard-coded tree search algorithms like Minimax.³ These algorithms enable agents to evaluate possible moves, anticipate opponents' actions, and select the most advantageous path forward. Deep Blue, IBM's chess-playing agent, showcased the potential for this kind of algorithm by defeating champion Garry Kasparov in 1997 using only preset utility values to evaluate possible future states.¹

As work in this area has progressed, reinforcement learning techniques have become favored over hard-coded utility values. Specifically, Q-Learning and Deep Q-Learning (DQN) have revolutionized the field and enabled artificial intelligence agents to learn and adapt to their environments. Q-Learning techniques use a Q matrix to associate states and actions with a Q value used to approximate the reward for that action in the given state. DQN is a more advanced version of this learning style that uses a neural network to approximate the reward rather than a Q matrix. This strategy has been shown to dramatically improve the performance of agents on simple Atari 2600 games and has become a standard approach to reinforcement learning.⁴

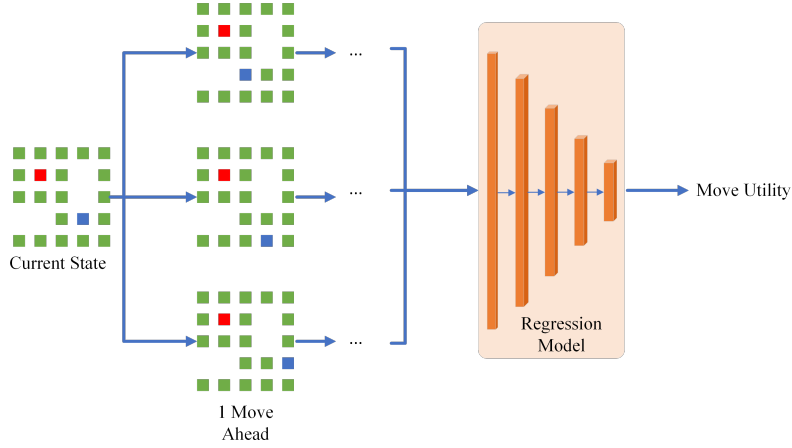


Figure 1. Training and inference method for our Deep-State Learning strategy. Our approach uses the Minimax tree search algorithm and uses an image regression model to evaluate future states’ utility.

By utilizing tree search with deep learning models, researchers have created agents to play much more complicated games such as Go. AlphaGo, using reinforcement learning, achieved a near 100% win rate against other Go programs.² Reinforcement learning has also been similarly applied in the popular strategy game Starcraft II where the agent AlphaStar achieved the highest possible rank in the game, beating over 99% of players.⁵ This was especially impressive considering StarCraft II is a continuous game with complex visuals, making tree search impossible. However, in both of these cases, training required expert knowledge of established strategies that had been discovered by humans. This drawback inspired AlphaGo Zero, a model similar to AlphaGo that does not require any previously held human knowledge.⁶ But, by relying solely on self-play throughout the learning process, AlphaGo Zero disregards the potential of using simpler algorithms as opponents in training. While these algorithms may not achieve the same level of performance as deep reinforcement learning architectures, they can act as useful teachers early in the training process as the model begins with entirely random moves.

Our novel reinforcement learning strategy combines traditional Minimax tree search, a DQN learning strategy, and simple algorithmic opponents in the training process in order to improve upon existing approaches.

2. METHODS

Our strategy, called Deep-State Learning (DSL), contains multiple innovations on standard tree-search and reinforcement learning approaches. Firstly, in both training and inference, an adapted deep image analysis model, in this case ResNet18,⁷ is used for state evaluation in place of hard-coded utility values. During training, we use a slightly modified DQN approach that utilizes Minimax tree search. The training process employs a combination of self-play and simple algorithmic opponents.

2.1 Deep Minimax Search

For selecting moves during both training and inference, we employ Minimax search³ with Alpha-Beta Pruning⁸ and Iterative Deepening.⁹ This allows us to efficiently search the state space of the game and set a state budget for the search, limiting the amount of time it takes to select each action. At each iteration of the algorithm, using iterative deepening, a maximum depth is set. When the search tree reaches that depth, our strategy uses an image analysis model like ResNet⁷ to evaluate the state rather than a standard utility calculation. We have adapted ResNet to perform regression on the game state rather than classification by removing the final Sigmoid layer. We also modified the number of input layers to correspond with the structure of the strategy game we have used for evaluation. After calculating the expected utility of each move, an incentive is applied for interesting moves that capture the opponent’s resources or develop the player’s assets. If the utility is positive, the value is multiplied by this incentive value. If the utility is negative, the value is divided by the incentive value. This move-selection process is shown in 1.

2.2 DQN Training

To train our model, we use a slightly modified form of DQN.⁴ In traditional DQN, a neural network $Q()$ is used to predict the expected reward, or Q-value of an action given the current state, S_0 . At each turn in the game, $Q()$ is used to evaluate each action $a \in A$ where A is the set of possible actions. The action with the highest Q-value, a_n , is selected. The state S_1 results from this action and the reward r is calculated using a reward function. Specifically, S_1 is the game state after a_n is executed and the opposing player responds. The array (S_0, a_n, r, S_1) is stored in a list called the experience replay. When the experience replay list is large enough, a batch can be selected and used for training. To stabilize training, a target network, $Q^*()$, is used where the weights of $Q()$ are copied to $Q^*()$ after some number of turns. As part of our strategy, we copy these weights at the beginning of each game. For each element in the batch, (S_0, a_n, r, S_1) , the Q-value for a_n is calculated as $Q(S_0) = q_n$. A target Q-value is also calculated as $Q^*(S_1) + r = q_n^*$. The loss is then determined as the average mean-squared error over the batch between each pair of q_n and q_n^* . This loss is only used to train the primary network $Q()$, not the target network $Q^*()$. By training in this manner, the network $Q()$ is trained to anticipate external changes to its environment while prioritizing the maximization of reward. We have modified this process such that when the action is being chosen, Deep Minimax search is used as described in 2.1. The Q-value, however, is calculated directly with $Q()$ without any tree search.

2.3 Opponent Selection

During the training process, we utilized a combination of self-play and simpler opponents. During the first segment of training, the model plays solely against opponents using the same Minimax search





Unit Name	Display Representation	Lifespan	Description
Town		30	Produces a worker every 6 turns
Worker		40	Creates a barracks
Barracks		8	Produces a soldier every 4 turns
Soldier		40	Attacks the opponent's units

Table 1. A list of units present in the game and their description.

algorithm with a simple utility calculation to evaluate future states. This utility calculation is the same as the reward function used in the training of the model. These simple opponents also employ the same incentive for interesting moves as the Deep Minimax Search described in 2.1. After this section of training, the opponent alternates between self-play with a slightly mutated network and the simpler opponents. During self-play, the mutated opponent network is initialized as $Q() + (\varepsilon * \mathcal{R})$. For each parameter in $Q()$, $(\varepsilon * \mathcal{R})$ is added where ε is a small value and \mathcal{R} is a random number selected from a normal distribution with mean 0 and variance 1. This number is different for each parameter. This network is not trained throughout the game and is reinitialized at the beginning of each self-play game. By playing against simple tree search opponents early on rather than relying solely on self-play, we hope to force the network to learn more quickly and move away from making entirely random moves. Including these simple opponents throughout the rest of the training process continues to encourage the model to not veer too far away from intelligent strategies.

3. EXPERIMENTS/RESULTS

To experimentally evaluate our approach, we have created a simple strategy game meant to be representative of other games in the genre and serve as an equivalent for other long-term planning tasks. The models were trained and compared against baselines on multiple randomly generated levels, including ones not included in the training set.

3.1 Generic Strategy Game

To effectively test our DSL training method, we have developed a new generic strategy game where the players cannot rely on established knowledge. This game is modeled after popular civilization management video games where each player must manage military and support assets with the goal of defeating the units of other players. Our game is played on an 8×8 grid with 2 players, similar to chess, and uses 4 separate types of units, described in 3.1. The purposes of each unit, their lifespan, and their display character are shown in this table. The units are all represented as chess pieces for ease of display. The lifespan of a unit is the number of turns it is allowed to exist on the board. After this period, the unit is automatically removed from the game. This was implemented to ensure games do not end in stalemates.

Each player begins the game with a single town placed on the upper or lower half of the board. In general, Player 1 starts on the lower half of the board while Player 2 starts on the upper half. The

town is able to create 5 workers over 30 turns. These workers can create barracks that produce 2 soldiers over 8 turns. The players can use these soldiers to capture any of their opponent’s pieces. The town and barracks are unable to move, as they represent immovable buildings. The worker and soldier, however, can move one square at a time in any direction including diagonally. Each player can make one of these moves per turn. When the player chooses to create a barracks with a worker, this action also counts as a turn and consumes the worker. The game ends when only one player has pieces on the board.

3.2 Levels and Randomization

We have created 7 total levels for evaluation and testing. Models were trained on 5 of these levels and evaluated on the entire set of 7. This was done to demonstrate the generalizability of our DSL strategy on unseen environments.

3.2.1 Level Generation

Each level was randomly generated using Perlin Noise.¹⁰ We combined 4 layers of this noise to smoothly create interesting features in every level. After the raw noise values were generated, all values above -0.2 were set as land where units are able to move. All other tiles in the grid are set to impassable and displayed as empty.

3.2.2 Training Randomization

During the training process, we wanted to expose each model to as many possible positions as we could over the 5 levels in the training set. To accomplish this, at the beginning of each game, we set a 50% chance that the level was vertically reversed to simulate playing from the other player’s position. We also set an 80% chance that the game state would be randomized. This randomization process involved consisted of placing workers, barracks, and soldiers around the board. For each tile on the board, a random one of these units was added with a probability of $15.625\% = \frac{10}{8*8}$. This would add around 10 units to the entire board. Their allegiance to Player 1 or Player 2 was determined by their vertical position on the board, with pieces on the upper half more likely to belong to Player 2 and vice versa.

3.3 Implementation Details

3.3.1 Model Architecture and Input

As previously mentioned, we use ResNet18 as a regression model to evaluate the utility of each possible future state when selecting actions. To enable this, we were forced to make small changes to the model’s architecture. We removed the Sigmoid layer at the end of the model to convert from classification to regression. We also modified the number of channels in the intermediate layers because

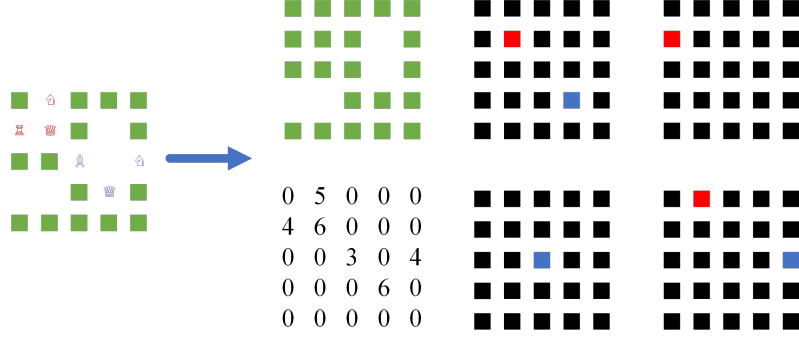


Figure 2. The input parsing process for our DSL models. The level state is broken down into 6 channels to be passed to our modified ResNet model.

of the small size of our game’s board. Instead of expanding up to 512 channels, our modified ResNet only reached 64. By doing this, we were able to keep the same number of residual blocks. The number of input channels was also modified from 3 to 6 to correspond with our game’s structure. The first of these channels is a binary matrix representing the entire level map, with 1 being land that units can move to and 0 being impassable obstacles. The next 4 channels correspond to each unit type. For each of these channels, -1 signifies a unit belonging to Player 2, 1 signifies a unit belonging to Player 1, and 0 signifies that no unit is present. The final channel contains the current age of each unit in turns where 0 indicates that no unit is present. This input parsing process is shown in 2.

3.3.2 Hyperparameters and Training

We trained each model over 100 games with a learn rate of 0.001 and batch size of 15. As mentioned in 2.3, alternating between simple opponents and self-play began after the 25th game. When using self-play, the mutation power for the opposing model was set at $\varepsilon = 0.01$. These values were selected through preliminary testing. We use a simple utility function to generate the reward for the model where towns are worth 9, barracks are worth 6, workers are worth 4, and soldiers are worth 3. Units held by the opposing player count for the same amount with their values subtracted from the overall utility. The final value is also divided by the number of units held by the opposing player to further encourage the agent to play aggressively. The incentive value as described in 2.1 was set to 1.2. This is the same utility function used by the simple opponent during training. The simple opponent is given the ability to look at a maximum of 10,000 potential future states during this search.

3.4 Baselines and Comparisons

When evaluating our model against baselines, we used a consistent simple opponent employing the same Minimax search as our architecture, just with a utility function instead of a deep model to assess future states. In this utility function, this constant opponent used the values shown in 3.4. Similar to the deep Minimax search used by our DSL strategy, the final utility value is divided by the number of opposing units, and an incentive value of 1.2 is applied to interesting moves. By analyzing the

Unit Name	Friendly Value	Enemy Value
Town	3	-2.5
Worker	0.5	-1
Barracks	1	-2
Soldier	0.8	-1.5

Table 2. The utility values for each unit used in our constant simple opponent.

performance of each of our models and each of the baselines against this constant opponent, we can easily determine the level of difference between them. Four different versions of this constant opponent were used, each given a different state budget for their tree search. These values were 1,000, 2,500, 5,000, and 10,000.

We trained three different versions of our DSL model with different state budgets to determine its effectiveness when given access to different levels of information. These models were trained with budgets of 1,000, 2,500, and 5,000.

We compared the performance of our model against multiple baselines as follows.

- **Random Bot:** Makes any valid random move during each turn.
- **Greedy Bot:** Always captures an opposing piece or creates a barracks with a worker whenever possible. When this is not possible, will select a random valid move.
- **Simple Opponent:** Utilizes Minimax tree search using the model’s reward function to evaluate future states. Analyzing our model’s performance against this baseline will indicate whether the modifications we made to the original search method resulted in any improvements when everything else is kept the same. This agent was given a state budget of 5,000.
- **Self-Play:** A model trained in the same way as our DSL model solely with self-play. This allows us to judge the effectiveness of our opponent selection strategy. The model was trained and evaluated using a state budget of 5,000.

3.5 Results in Seen Environments

Our results comparing DSL to multiple baseline agents are shown in 3. These results were gathered by playing 10 games against each version of the constant opponent on the 5 levels present in the training set. Each agent played as Player 1 and Player 2 in every environment. As shown, our DSL model with a state budget of 5,000 clearly outperforms every other agent by at least 15% in total wins. Even without our model-selection strategy, demonstrated with the DSL Self-Play agent, the performance reaches the same level as standard Minimax search with a simple utility function. This proves that our training method can serve as a sufficient alternative to standard tree-search methods without deep models. Interestingly, when our DSL model uses a state budget of 5,000 and plays against an

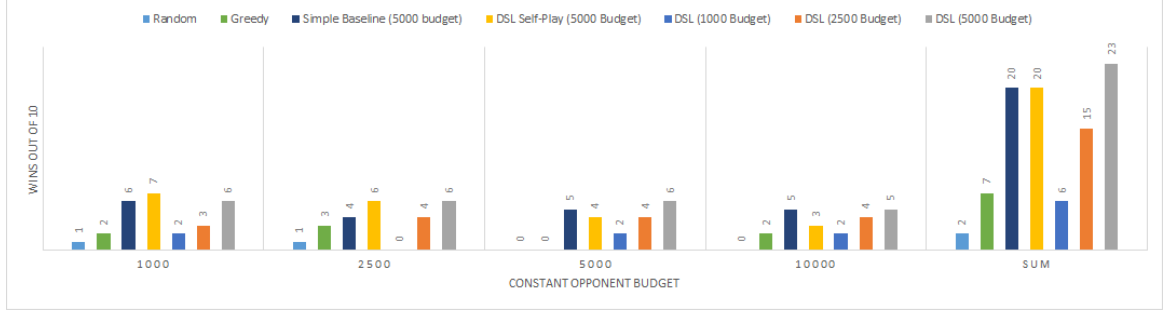


Figure 3. Number of wins out of 10 games on the 5 levels in the DSL training set. The sum total of each agent’s wins is displayed on the right.

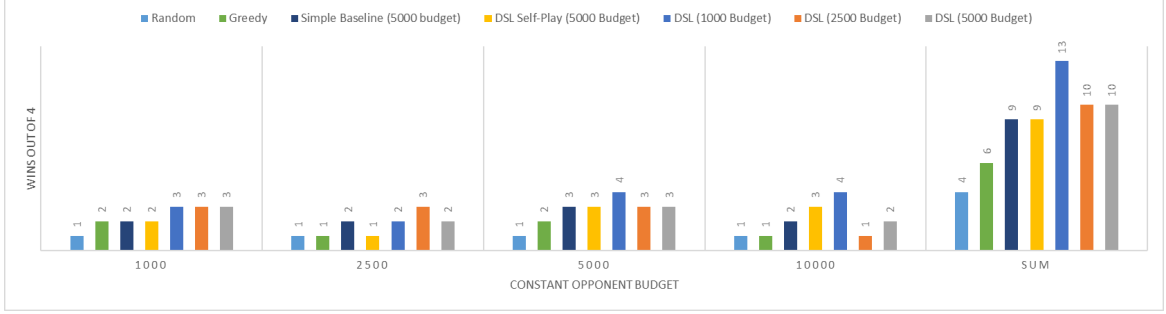


Figure 4. Number of wins out of 4 on the 2 levels outside the DSL training set. The sum total of each agent’s wins is displayed on the right.

opponent with a budget of 10,000, it wins in half of the games. This shows that our approach can use the available information more efficiently, matching agents given a higher budget.

3.6 Results in Unseen Environments

We also compared our approach with baselines on 2 levels outside the training set to test its generalizability. These results are shown in 4. As with the results on seen environments, we find that our DSL approach results in consistent improvements in total wins over all baselines. Surprisingly, we found that the DSL model with a budget of only 1,000 outperformed all other agents by a minimum of 3 total wins. This could be attributed to the relative unpredictability of this agent given that it has access to a lower amount of information when compared to the others. These results also confirm the idea that the model using solely self-play performs similarly to an agent using standard tree search.

4. CONCLUSIONS AND DISCUSSION

In this work, we present a new deep-state learning approach for strategy games and general-purpose long-term planning tasks. Our strategy combines traditional tree search algorithms that have long been established with deep image analysis models and a novel opponent selection approach. We have evaluated our method against multiple baselines on a generic strategy game meant to correlate to other applications. Through these evaluations, we have found that our work matches or exceeds all relevant baselines and have verified that our opponent selection strategy constitutes a vital improvement. There exist multiple avenues for future work on this project, including using different imaging model architectures, employing a more advanced reward function, and exploring other tree-search algorithms.

REFERENCES

- [1] Campbell, M., Hoane, A., and hsiung Hsu, F., “Deep blue,” *Artificial Intelligence* **134**(1), 57–83 (2002).
- [2] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D., “Mastering the game of go with deep neural networks and tree search,” *Nature* **529**, 484–489 (Jan 2016).
- [3] Rivest, R. L., “Game tree searching by min/max approximation,” *Artif. Intell.* **34**, 77–96 (1987).
- [4] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D., “Human-level control through deep reinforcement learning,” *Nature* **518**, 529–533 (Feb 2015).
- [5] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D., “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” *Nature* **575**, 350–354 (Nov 2019).
- [6] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D., “Mastering the game of go without human knowledge,” *Nature* **550**, 354–359 (Oct 2017).
- [7] He, K., Zhang, X., Ren, S., and Sun, J., “Deep residual learning for image recognition,” (2015).
- [8] Knuth, D. E. and Moore, R. W., “An analysis of alpha-beta pruning,” *Artificial Intelligence* **6**(4), 293–326 (1975).
- [9] Korf, R. E., “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial Intelligence* **27**(1), 97–109 (1985).
- [10] Perlin, K. and Hoffert, E. M., “Hypertexture,” in [*Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*], *SIGGRAPH ’89*, 253–262, Association for Computing Machinery, New York, NY, USA (1989).