Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks

Marius Stanescu, Nicolas A. Barriga, Andy Hess and Michael Buro
Department of Computing Science
University of Alberta, Canada
{astanesc|barriga|athess|mburo}@ualberta.ca

***Abstract*—Real-time strategy (RTS) games, such as Blizzard's StarCraft, are fast paced war simulation games in which players have to manage economies, control many dozens of units, and deal with uncertainty about opposing unit locations in real-time. Even in perfect information settings, constructing strong AI systems has been difficult due to enormous state and action spaces and the lack of good state evaluation functions and high-level action abstractions. To this day, good human players are still handily defeating the best RTS game AI systems, but this may change in the near future given the recent success of deep convolutional neural networks (CNNs) in computer Go, which demonstrated how networks can be used for evaluating complex game states accurately and to focus look-ahead search.**

In this paper we present a CNN for RTS game state evaluation that goes beyond commonly used material based evaluations by also taking spatial relations between units into account. We evaluate the CNN's performance by comparing it with various other evaluation functions by means of tournaments played by several state-of-the-art search algorithms. We find that, despite its much slower evaluation speed, on average the CNN based search performs significantly better compared to simpler but faster evaluations. These promising initial results together with recent advances in hierarchical search suggest that dominating human players in RTS games may not be far off.

I. Introduction

The recent success of AlphaGo [1], culminating in the 4-1 win against one of the strongest human Go players, illustrated the effectiveness of combining Monte Carlo Tree Search (MCTS) and deep learning techniques. For AlphaGo, convolutional neural networks (CNNs) [2], [3] were trained to mitigate the prohibitively large search space of the game of Go in two ways: First, a policy network was trained, using both supervised and reinforcement learning techniques, to return a probability distribution over all possible moves, thereby focusing the search on the most promising branches. Second, MCTS state evaluation accuracy was improved by using both value network evaluations and playout results.

In two-player, zero-sum games, such as Chess and Go, optimal moves can be computed by using the minimax rule that minimizes worst case loss. In theory, these games can be solved by recursively applying this rule until reaching terminal states. However, in practice completely searching the game tree is infeasible, the procedure must be cut short, and an approximate evaluation function must be used to estimate the value of the game state. Because states closer to the end of the game are typically evaluated more accurately, deeper search produces better moves. But as game playing agents often have to make their move decision under demanding time constraints, great performance gains can be achieved by improving the evaluation function's accuracy.

The size of the state space of the game of Go, although much larger than that of Chess, is tiny in comparison to real-time strategy (RTS) games such as Blizzard's StarCraft. In the game of Go, at every turn, a single stone can be placed at any valid location on the 19×19 board and the average game length is around 150 moves. In RTS games, each player can simultaneously command many units to perform a large number of possible actions. Also, a single game can last for tens of thousands of simulation frames, with possibly multiple moves being issued in each one. Moreover, RTS game maps are generally much larger than Go boards and feature terrain that often affects movement, combat, and resource gathering. Therefore, for RTS games, good state evaluations and search control, such as using policy networks, plays an even greater role.

CNNs are adept at learning complex relationships within structured data due to their ability to learn hierarchies of abstract, localized representations in an end-to-end manner [3]. In this paper we investigate the effectiveness of training a CNN to learn the value of game states for a simple RTS game and show significant improvement in accuracy over simpler state-of-the-art evaluations. We also show that incorporating the resulting learned evaluation function into state-of-the-art RTS search algorithms increases agent playing strength considerably.

II. Related Work

Search based planning approaches have had a long tradition in the construction of strong AI agents for abstract games like Chess and Go, and in recent years they have progressively been applied to modern video games, especially the RTS game StarCraft. This is a difficult endeavor due to the enormous state and action spaces, and finding optimal moves under tight real-time constraints is infeasible for all but the smallest scenarios. Consequently, the research focus in this area has been on reducing the search space via different abstraction mechanisms and on producing good state evaluation functions to guide this search effort.

In this section we briefly discuss some of these attempts, starting with various methods used for state evaluation in

RTS games. We then present recent research on deep neural networks and their use in game playing agents.

A. State Evaluation in RTS Games

Playing RTS games well requires strategic as well as tactical skills, ranging from building effective economies, over deciding what to build next based on scouting results, to maneuvering units in combat encounters. In RTS game combat each player controls an army consisting of different types of units and tries to defeat the opponent's army while minimizing its own losses. Because battles have a big impact on the result of RTS games, predicting their outcome accurately is very important, especially for look-ahead search algorithms.

A common metric for estimating combat outcomes is LTD2 [4], which is based on the lifetime damage each unit can inflict. LTD2 was used, in conjunction with short deterministic playouts, for node evaluation in alpha-beta search to select combat orders for individual units [5]. A similar metric was later used as state evaluation, this time combined with randomized playouts [6], [7].

Likewise, Hierarchical Adversarial Search [8] requires estimates of combat outcomes for state evaluation and uses a simulator for this purpose. However, because simulations become more expensive as the number of units grows, faster prediction methods are needed. For instance, a probabilistic graphical model trained on simulated battles can accurately predict the winner [9]. This model, however, has several limitations such as not modeling damaged units and not distinguishing between melee and ranged combat. Another model, based on Lanchester's attrition laws [10], does not have such shortcomings. It takes into account the relative strength of different unit types, their health and the fact that ranged weapons enable units to engage several targets without having to move, which causes a non-linear relationship between army size differences and winning potential. After learning unit strength values offline using maximum likelihood estimation from past recorded battles, this improved model has been successfully used for state evaluation in a state-of-the-art RTS search algorithm [11].

All mentioned approaches focus on a single strategic component of RTS games, i.e. combat, and lack spatial reasoning abilities, ignoring information such as unit positions and terrain. Global state evaluation in complex RTS games such as StarCraft has been less successful [12], likely due to the limited expressiveness of the linear model used.

B. Neural Networks

In recent years deep convolutional neural networks (CNNs) have sparked a revolution in AI. The spectacular results achieved in image classification [3] have led to deep CNNs being effectively applied to a wide range of domains. For vision tasks, CNNs have been applied to object localization [13], segmentation [14], facial recognition [15], super-resolution [16] and camera-localization [17] to name just a few examples, all the while continuing to make further progress in image classification [18]. Deep CNNs have also been successfully applied to tasks as diverse as natural language categorization [19], [20], translation [21] and algorithm learning [22].

Deep CNNs owe their success to their ability to learn multiple levels of abstraction, each one building upon abstractions learned in previous layers. More specifically, deep CNNs learn a hierarchy of spatially invariant, localized representations, each layer aggregating and building upon representations in previous layers toward the combined goal of minimizing loss [13].

There is a long history of using simple linear regression and shallow neural networks to construct strong AI systems for classic board games such as Backgammon and Othello [23]. However, scaling up state evaluations to more complex games such as Go only became possible when it was discovered how to effectively train weights in deep neural networks, which can be considerably more expressive than shallow networks with the same number of weights [24].

Since then CNNs have been successfully used to play Atari video games with a policy network trained by supervised learning, using training data generated by a slow but strong UCT player [25]. Similar networks have been trained with reinforcement learning [26], [27]. Most remarkable, however, is the recent 4-1 win of AlphaGo [1], a deep CNN based Go playing program, over one of today's best Go players Lee Sedol. AlphaGo combines MCTS with deep CNNs for state evaluation and move selection that were trained by supervised and reinforcement learning.

This historic accomplishment sparks hope that CNNs can also be used for even more complex tasks, such as playing real-time games with imperfect information — a domain still dominated by human players.

III. A Neural Network for RTS Game State Evaluation

In this section we describe the dataset, the neural network structure and the procedure used for training a state evaluation network for μRTS[1], a simple RTS game designed for testing AI techniques. μRTS provides the essential features of an RTS game: it supports four unit and two building types, all of them occupying one tile, and there is only one resource type. The game state is fully observable. μRTS supports configurable map sizes, commonly ranging from $8{\times}8$ to $16{\times}16$ in published papers. The game user interface and details about the unit types are shown in Figure 1. μRTS comes with a few basic scripted players, as well as search based players implementing several state-of-the-art RTS search techniques [6], [28], [7], making it an useful tool for benchmarking new AI algorithms.

The purpose of the neural network we describe here is to approximate the value function $v^*(s)$, which represents the win-draw-loss outcome of the game starting in state s assuming perfect play on both sides.

[1] https://github.com/santiontanon/microrts

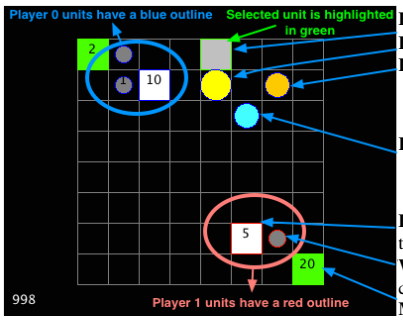

Fig. 1. Screenshot of μRTS, with explanations of the different in-game symbols.

In practice we have to approximate this value function with v_θ, for instance by using a neural network with weights θ. These weights are trained by regression on state-outcome pairs (s, w), using stochastic gradient descent to minimize the mean squared error between the predicted value $v_\theta(s)$ and the corresponding outcome w. The output of our network will be the players' probabilities of winning the game when starting from the input position.

A. Data

The dataset used for training the neural network was created by playing round-robin tournaments between 15 different μRTS bots, 11 of which are included in the default μRTS implementation. The other 4 are versions of the Puppet Search algorithm [11]. Each tournament consists of $(15\times14)/2 = 105$ matches. One 8×8 map was used, with 24 different initial starting conditions. All scenarios start with one base and one worker for each player, but with different, symmetric, initial positions. These tournaments were played under four different time limits: maximums of $100ms$, $200ms$, 100 playouts and 200 playouts per search episode. In total $105\times24\times4 = 10\,080$ different games were played from which draws were discarded ($\approx 8\%$).

Predicting game outcomes from data consisting of complete games leads to overfitting because while successive states are strongly correlated, the regression target is shared for the entire game. To mitigate the problem, the authors of AlphaGo [1] add only a single training example (s, w) to the dataset from each game. Because we have significantly less data (10 thousand vs. 30 million episodes), we chose to sample 3 random positions from each game. As a result, for game i we add $\{(s_{i1}, w_i), (s_{i2}, w_i), (s_{i3}, w_i)\}$ to the dataset, and slightly over 25 000 positions are generated.

The dataset was split into a test set (5 000 positions) and a training set (the remaining 20 000 positions). Finally, the training set was augmented by including all reflections and rotations of each position for a total of 160 000 positions.

B. Features

Each position s is preprocessed into a set of 8×8 feature planes. These features correspond to the raw board representation and contain information about each tile of the μRTS map: unit ownership and type, current health points, game frames until actions are completed and resources.

TABLE I
INPUT FEATURE PLANES FOR THE NEURAL NETWORK.

Feature	# of planes	Description
Unit type	6	Base, Barracks, worker, light, ranged, heavy
Unit health	5	$1, 2, 3, 4$, or ≥ 5
Unit owner	2	Masks to indicate all units belonging to one player
Frames to completion	5	$0-25$, $26-50$, $51-80$, $81-120$, or ≥ 121
Resources	7	1, 2, 3, 4, 5, $6-9$, or ≥ 10

All integers, such as unit health points, are split into K different 8×8 planes of binary values using the one-hot encoding. For example, five separate binary feature planes are used to represent whether an unit has $1, 2, 3, 4$ or ≥ 5 health points. The full set of feature planes is listed in Table I.

C. Network Architecture & Training Details

The input to the neural network is an 8×8×25 image stack consisting of 25 feature planes. There are two convolutional layers that pad the input with zeros to obtain a 10×10 image. Each then is convolved with 64 and respectively 32 filters of size 3×3 with stride 1. Both are followed by leaky rectified linear units (LReLUs) [29], [30]. A third hidden layer convolves 1 filter of size 1×1 with stride 1, again followed by an LReLU. Then follow two fully connected (dense) linear layers, with 128 and 64 LReLU units, respectively. A dropout ratio of 0.5 is applied to both fully connected layers. The output layer is a fully connected layer with two units, and a softmax function is applied to obtain the winning probabilities for player 0 and player 1 ($P(p0)$ and $P(p1)$). All LReLUs have negative slope of $\alpha = -1/5.5$. The resulting architecture is shown in Figure 2.

Our architecture was motivated by current trends toward the use of small filter sizes ($\leq$ 3×3), few (or no) pooling layers, and same-padded convolution (multiple layers of the same width and height, each layer padded with zeros following convolution) [31], [1]. We were also guided by the principle of gradually decreasing the dimension of internal representations as one moves from input toward task; one example being the reduction from 64 to 32 filters, another being the use of 1×1 convolutions for dimensionality reduction [18]. This principle can also be seen in the fully connected layers. LReLUs were used following suggestions from [29] and [30].

Before training, we used Xavier random weight initialization [32] which equalizes signal variance. During training, the stepsize alpha was initialized to 0.00001 and was multiplied by 0.2 every $100K$ training steps. We used adaptive moment estimation (ADAM) with default values of $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ as suggested in [33]. The network was trained for $400K$ mini-batches of 64 positions, a process which took approximately 20 minutes on a single GPU to converge.

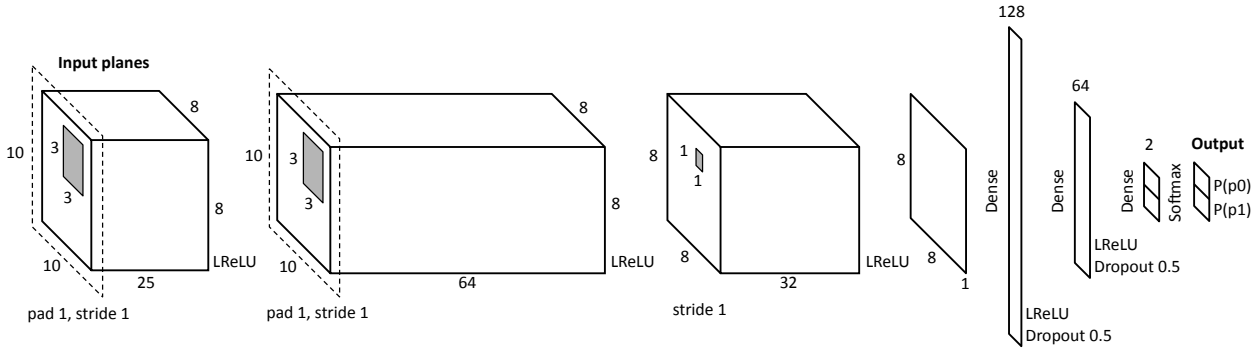

Fig. 2. Neural network architecture.

For training, we used the Python (2.7.6) interface to Caffe [34], utilizing CUDA[2] version 7.5 and cuDNN[3] version 4. The machine used for training the neural network had an Intel(R) Pentium(R) CPU G2120 3.10GHz processor, 8 GB RAM and one GeForce GTX 760 GPU (1152 cores and 4 GB memory) running Linux Mint 17.3.

IV. Experiments and Results

All experiments that are reported below were performed on Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz with 8 GB RAM machines running Ubuntu 14.04. The test machines do not have CUDA capability 3 and the neural network computations were run solely on the CPU. μRTS software is implemented in Java and compiled and run with JDK 8u74.

A. Winner Prediction Accuracy

In the first set of experiments we compare the speed and accuracy of our neural network for evaluating game states with a Lanchester model [10] and a simple evaluation function that takes into account the cost and health points of units and the resources each player has. This is the default evaluation function that the μRTS search algorithms use. In equation 1 player indices are either 0 or 1: $player \in \{0, 1\}$.

$$eval(player) = E_{player} - E_{1-player} \quad (1)$$

In equation 2, R_p is the amount of resources a player currently has, W_p is a player's set of workers, R_u is the amount of resources each worker unit is carrying, C_u is the cost of unit u, HP_u the current health points of unit u, and MaxHP_u its maximum health points. $W_{res}, W_{work}, W_{unit}$ are constant weights.

$$E_p = W_{res}R_p + W_{work}\sum_{u \in W_p} R_u + W_{unit}\sum_{u \in p}\frac{C_u \text{HP}_u}{\text{MaxHP}_u} \quad (2)$$

Two versions of this simple evaluation functions were used: one with μRTS's default weights, and one optimized via logistic regression on the same training set used for the neural network. The Lanchester model keeps the two resource terms of the simple evaluation function but revises the army's impact. While the contribution of the buildings is similar to equation 2, a new term is added for combat units:

$$E'_p = W_{res}R_p + W_{work}\sum_{u \in W_p} R_u + W_{base}\frac{\text{HP}_{base}}{\text{MaxHP}_{base}} + W_{barracks}\frac{\text{HP}_{barracks}}{\text{MaxHP}_{barracks}} + N_p^{(o-1)}\sum_{u \in p}\alpha_u \frac{\text{HP}_u}{\text{MaxHP}_u} \quad (3)$$

In equation 3, α_u is a strength value unique to each unit type, N_p is the total number of units of player p and o is the Lanchester attrition order. For μRTS, our experiments suggest that an attrition order of $o = 1.7$ works best on average if we had to choose a fixed order for all possible encounters. The four W and four α constants (one for each unit type) are optimized with logistic regression.

Figure 3 shows the position evaluation accuracy of the neural network, compared to the default μRTS evaluation function, the optimized version and the Lanchester model, on the previously described test set of 5 000 positions sampled from bot games. A scripted playout evaluation was also tested, in which the position is played until the end using the WorkerRush script, described in section IV-B, to generate moves for both players. Values of 1, 0 or -1 are returned, corresponding to a player 0 win, draw or loss, respectively. The WorkerRush script is the strongest of the four scripts described in the next section, and produces the most accurate winner prediction function, though slightly worse than the simple evaluation function. A random playout was also tried, but it performed even worse.

The neural network is consistently more accurate during the first half of the game. At the beginning of the game before any unit has been built, the simple evaluation function and the Lanchester model mostly predict draws, because they do not take positional information into account. During the second half of the game army balance is more relevant, and both the neural network and the Lanchester model perform better than the simple evaluation functions.

[2]https://developer.nvidia.com/cuda-toolkit
[3]https://developer.nvidia.com/cudnn

The average time needed for a single simple evaluation is $0.012\mu s$, the Lanchester model takes $0.087\mu s$, while a full network evaluation on the CPU takes $147\mu s$. This time includes processing the games state into feature planes, sending the data to a Python thread (on the same CPU core as the search algorithm), running a forward pass on the network and returning the outcome. The network evaluation takes close to two thirds of the time, around $102\mu s$. We tested the speed of the network evaluation on a GPU as well. On a mid-range NVIDIA GTX 760, the time is slightly shorter than the CPU-only version ($118\mu s$).

However, processing only one position at a time does not take advantage of the pipelined GPU architecture. To measure potential gains of evaluating positions in parallel, we ran batches of 256 positions whose evaluation took $10\,707\mu s$, of which $9\,985\mu s$ was spent on the CPU (feature planes) and $722\mu s$ on the GPU, for an average of $2.8\mu s$ of GPU time per evaluation. A search algorithm — like AlphaGo's — that can perform leaf evaluations asynchronously would benefit greatly from doing state evaluations on the GPU.

B. State Evaluation in Search Algorithms

A second set of experiments compares the performance of four game tree search algorithms — ϵ-Greedy MCTS, Naïve MCTS, AHTN-F and AHTN-P, described below — when using the simple evaluation function, the optimized evaluation function, the Lanchester model or the neural network for state evaluation.

The sixteen resulting algorithms played against the following eleven opponents provided by the μRTS implementation, all using default parameters and the simple μRTS evaluation function:

WorkerRush: a hardcoded rush strategy that constantly produces workers and sends them to attack.

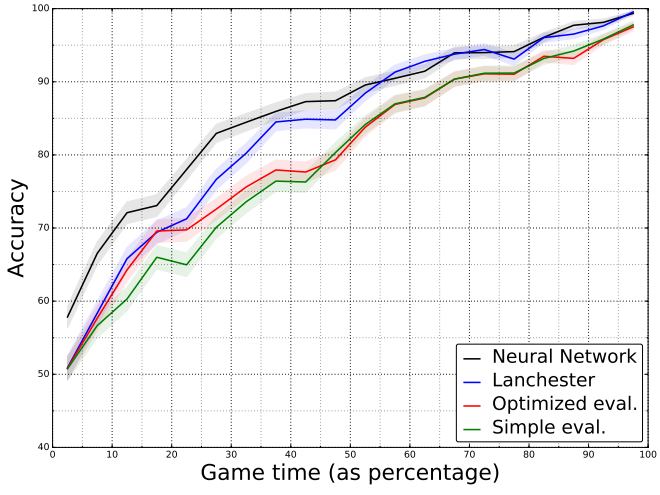

Fig. 3. Comparison of evaluation accuracy between the neural network, μRTS's built-in evaluation function, its optimized version and the Lanchester model. The accuracy at predicting the winner of the game is plotted against the stage of the game, expressed as a percentage of the game length. Results are aggregated in 5% buckets. Shaded area represents one standard error.

LightRush: builds a barracks, and then constantly produces *light* military units to attack the nearest target (it uses one worker to mine resources).

RangedRush: is identical to **LightRush**, except for producing *ranged* units.

HeavyRush: is identical to **LightRush**, except for producing slower but stronger *heavy* units.

MonteCarlo(MC): a standard Monte Carlo search algorithm: for each legal player action, it runs as many simulations as possible to estimate their expected reward.

ϵ-Greedy MC: Monte Carlo search, but using an ϵ-greedy sampling strategy.

Naïve MCTS: Monte Carlo Tree Search algorithm with a sampling strategy specifically designed for games with combinatorial branching factors, such as RTS games. This strategy, called *Naïve Sampling*, exploits the particular tree structure of games that can be modeled as a Combinatorial Multi-Armed Bandit [6].

ϵ-Greedy MCTS: like NaïveMCTS, but using an ϵ-greedy sampling strategy.

MinMax Strategy: for a set of strategies (WorkerRush, LightRush, RangedRush and Random), playouts are run for all possible pairings. It approximates the Nash equilibrium strategy using the minimax rule, whereby one player (Max) maximizes its payoff value while the other player tries to minimize Max's payoff [35].

AHTN-P: an Adversarial Hierarchical Task Network, that combines minimax game tree search with HTN planning [7]. In this AHTN definition the main task of the game can be achieved only by three non-primitive tasks (abstract actions that decompose into actions that agents can directly execute in the game). The tasks are three rushes with three different unit types.

AHTN-F: a more elaborate AHTN with a larger number of non-primitive tasks for harvesting resources, training units of different types, or attacking the enemy.

All search based algorithms (bottom seven in the list above) evaluate states by running a short playout of 100 frames. The playouts are performed using a random policy in which non-move actions (harvest, attack, build) have a higher probability than moves. The only exception is MinMax, whose playouts are 400 frames long, because it only does 16 playouts — one for each pair of strategies — and uses its fixed set of strategies instead of the random policy. The resulting states are evaluated with the simple evaluation function in equation 1, the optimized function, the Lanchester model or the neural network.

Every player has a computational budget of either a given time duration or a maximum number of state evaluations per game frame. Moreover, players can split the search process over multiple frames; for example, if the game state does not change during 10 game frames before a player needs to issue an action, then players have ten times the budget to issue actions. We call this consolidated budget a *search episode*.

In the tournament each of the 176 matchups consists of 24 games played on an 8×8 map, with different but symmetric

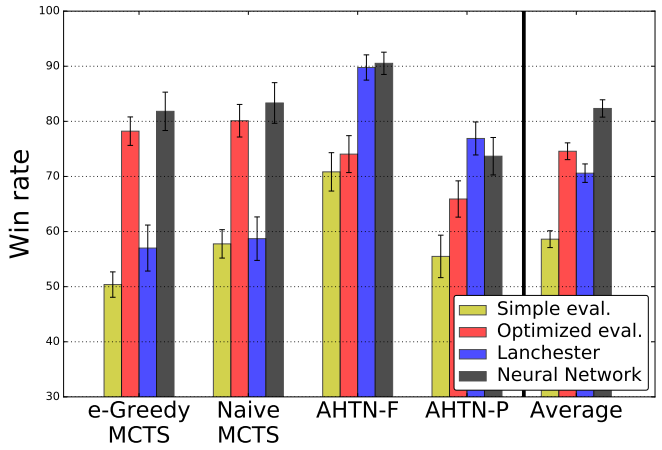

Fig. 4. Average win rate against all opponents when using the simple evaluation function described in equations 1 and 2, the same function with optimized weights, the Lanchester model or the neural network described in section III. Each algorithm has **200 milliseconds** of search time per frame. Error bars show one standard error.

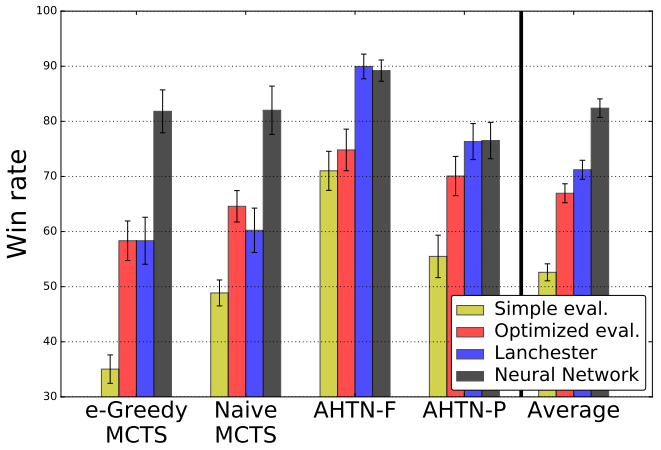

Fig. 5. Average win rate against all opponents when using the simple evaluation function described in equations 1 and 2, the same function with optimized weights, the Lanchester model or the neural network described in section III. Each algorithm is allowed to expand **200 nodes** per frame. Error bars show one standard error.

starting positions. To compute the score, every win is worth 1 point, and if the game reaches 3 000 frames, it is considered a draw, and awarded 0.5 points.

Figure 4 summarizes the average win rate against all opponents when using the different evaluation methods. On average, the neural network shows over 10% higher win rates than the other methods. Moreover, the performance of the neural network is consistent across all four algorithms, while the results of the optimized evaluation and the Lanchester model fluctuate depending on the underlying search algorithm type.

Table II shows the average number of nodes expanded per search episode. The average length of a search episode in the tournament games was around seven frames. Slow search algorithms such as AHTNs are less affected by a slow state evaluation, as most of their computational effort is expended in the tree phase. As a result, the AHTNs perform better when using the most accurate functions, regardless of their speed. The balance on the faster MCTS algorithms is more delicate, with both the fast optimized evaluation function and the neural network outperforming the relatively accurate and fast Lanchester model. The $\sim$1% accuracy increase in the first quarter of the game between the optimized simple evaluation and Lanchester is not enough to offset the $\sim$15% less nodes per second. However, the $\sim$7% accuracy gain of the neural network more than makes up for its $\sim$93% speed loss. Improving the accuracy of the evaluation function at the beginning of the game is important, as early game decisions likely have a large impact on the game outcome.

TABLE II
NODES EXPANDED PER SEARCH EPISODE, WHEN RUNNING WITH A MAXIMUM TIME LIMIT OF 200MS PER FRAME.

	Average # nodes expanded per search episode		
AI Algorithm	Simple Evaluation	Lanchester	Neural Network
ϵ-Greedy MCTS	16834	14682	1069
Naïve MCTS	16654	13876	1122
AHTN-F	937	969	507
AHTN-P	134	125	123

Figure 5 shows a summary of a similar tournament using a limit of 200 state evaluations per frame, rather than 200 milliseconds. The fastest simple evaluation function shows significantly worse performance on the MCTS algorithms, because in this experiment only the evaluation accuracy is relevant, not the speed.

To scale the neural network to larger map sizes and more complex games, the size of the network will likely have to increase, both in the size of each layer and in the number of layers. This expansion will lead to slower evaluation times. However, we have shown that a small increase in evaluation accuracy is able to compensate for several orders of magnitude in speed reduction. Furthermore, running the network in batches on a GPU rather than the CPU should counteract most of the lost speed. MCTS algorithms can readily be modified to perform state evaluations in batches, as done for AlphaGo [1], which would result in several orders of magnitude speed improvements.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have used deep CNNs to evaluate RTS game states. We have shown that the evaluation accuracy is higher than current alternatives in μRTS. This new method performed better evaluating early game positions which led to stronger gameplay when used within state-of-the-art RTS search algorithms.

While CNNs might not perform significantly better in all cases (for instance compared to Lanchester when used in ATHN-P and AHTN-F, see Figures 4 and 5), the game playing

agents based on them were stronger on average. Evaluating our CNN is several orders of magnitude slower than the other evaluation functions, but the accuracy gain far outweighs the speed disadvantage.

With these promising results, coupled with the fact that modern CNNs have shown excellent results on large problem sets [3], we are confident that the presented methods will scale up to more complex RTS games. StarCraft maps are similar in size to the images these networks are usually applied to. Using an MCTS implementation based on game abstractions similar to μRTS, that allows for asynchronous state evaluations on multiple GPUs can aid in tackling these larger problems while meeting real time constraints. Moreover, policy networks may also be trained to return probability distributions over the possible moves which can be used as prior probabilities to focus MCTS on the most promising branches.

Unlike Go, however, even RTS games with professional leagues such as StarCraft do not make replays of competition games publicly available. Without a large number of high quality records, reinforcement learning techniques will likely need to be considered in future work.

REFERENCES

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[4] A. Kovarsky and M. Buro, "Heuristic search applied to abstract combat games," *Advances in Artificial Intelligence*, pp. 66–78, 2005.

[5] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2012.

[6] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *AIIDE*, 2013.

[7] S. Ontañón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *IJCAI*, 2015, in press.

[8] M. Stanescu, N. A. Barriga, and M. Buro, "Hierarchical adversarial search applied to real-time strategy games," in *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2014.

[9] M. Stanescu, S. P. Hernandez, G. Erickson, R. Greiner, and M. Buro, "Predicting army combat outcomes in StarCraft," in *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[10] M. Stanescu, N. A. Barriga, and M. Buro, "Using Lanchester attrition laws for combat prediction in StarCraft," in *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2015.

[11] N. A. Barriga, M. Stanescu, and M. Buro, "Puppet Search: Enhancing scripted behaviour by look-ahead search with applications to Real-Time Strategy games," in *Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2015.

[12] G. Erickson and M. Buro, "Global state evaluation in StarCraft," in *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[13] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[14] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440.

[15] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 815–823.

[16] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *Computer Vision–ECCV 2014*. Springer, 2014, pp. 184–199.

[17] A. Kendall, M. Grimes, and R. Cipolla, "Posenet: A convolutional network for real-time 6-dof camera relocalization," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2938–2946.

[18] C. Szegedy, S. Ioffe, and V. Vanhoucke, "Inception-v4, inception-resnet and the impact of residual connections on learning," *arXiv preprint arXiv:1602.07261*, 2016.

[19] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," *arXiv preprint arXiv:1412.1058*, 2014.

[20] X. Zhang and Y. LeCun, "Text understanding from scratch," *arXiv preprint arXiv:1502.01710*, 2015.

[21] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[22] Ł. Kaiser and I. Sutskever, "Neural gpus learn algorithms," *arXiv preprint arXiv:1511.08228*, 2015.

[23] J. Frnkranz and M. Kubat, *Machines that learn to play games*. Nova Publishers, 2001.

[24] Y. LeCun and M. A. Ranzato, "Deep learning," Tutorial at ICML, 2013. [Online]. Available: www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf

[25] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in Neural Information Processing Systems*, 2014, pp. 3338–3346.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] A. Shleyfman, A. Komenda, and C. Domshlak, "On combinatorial actions and cmabs with linear side information," in *ECAI*, 2014, pp. 825–830.

[29] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1026–1034. [Online]. Available: http://www.cv-foundation.org/openaccess/content_iccv_2015/html/He_Delving_Deep_into_ICCV_2015_paper.html

[31] ——, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[32] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[34] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[35] F. Sailer, M. Buro, and M. Lanctot, "Adversarial planning through strategy simulation," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*. IEEE, 2007, pp. 80–87.