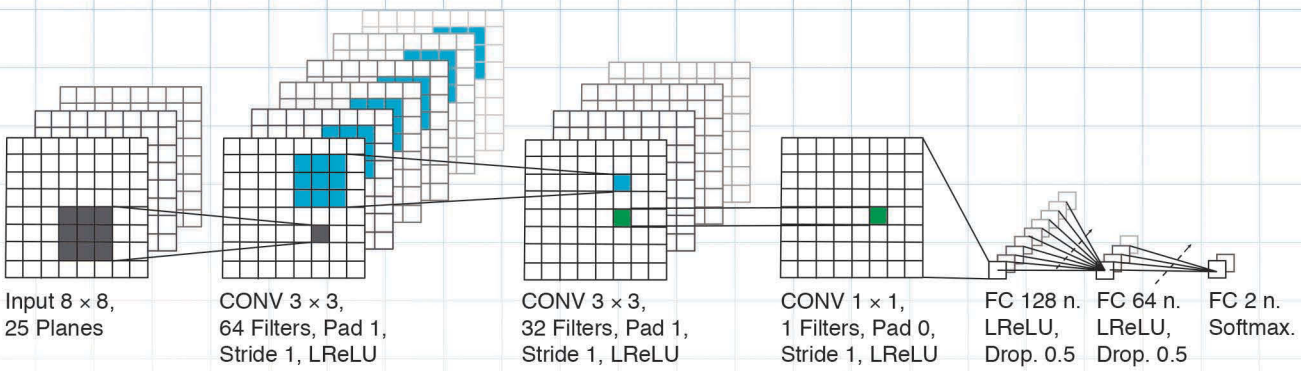
Improving RTS Game AI by Supervised Policy Learning, Tactical Search, and Deep Reinforcement Learning



Nicolas A. Barriga
School of Videogames Development and VR Engineering, Universidad de Talca, Talca, CHILE

Marius Stanescu
Department of Computing Science, University of Alberta, Edmonton, AB, CANADA

Felipe Besoain
School of Videogames Development and VR Engineering, Universidad de Talca, Talca, CHILE

Michael Buro
Department of Computing Science, University of Alberta, Edmonton, AB, CANADA

***Abstract*—**Constructing strong AI systems for video games is difficult due to enormous state and action spaces and the lack of good state evaluation functions and high-level action abstractions. In spite of recent research progress in popular video game genres such as Atari 2600 console games and multiplayer online battle arena (MOBA) games, to this day strong human players can still defeat the best AI systems in adversarial video games. In this paper, we propose to use a deep Convolutional Neural Network (CNN) to select among a limited set of abstract action choices in Real-Time Strategy (RTS) games, and to utilize the remaining computation time for game tree search to improve low-level tactics. The CNN is trained by supervised learning on game states labeled by Puppet Search, a strategic search algorithm that uses action abstractions. Replacing Puppet Search by a CNN frees up time that can be used for improving units' tactical behavior while executing the strategic plan. Experiments in the μRTS game show that the combined algorithm results in higher win-rates than either of its two independent components and other state-of-the-art μRTS

Digital Object Identifier 10.1109/MCI.2019.2919363
Date of publication: 17 July 2019

Corresponding Author: Nicolas A. Barriga (Email: nbarriga@utalca.cl)

agents. We then present a case-study that investigates how deep Reinforcement Learning (RL) can be used in modern video games, such as Total War: Warhammer, to improve tactical multi-agent AI modules. We use popular RL algorithms such as Deep-Q Networks (DQN) and Asynchronous Advantage-Actor Critic (A3C), basic network architectures and minimal hyper-parameter tuning to learn complex cooperative behaviors that defeat the highest difficulty built-in AI in medium-scale scenarios.

I. Introduction

Being able to construct high-performance video game AI systems that can act as Non-Player Characters (NPCs), adversaries, allies, or instructors would revolutionize the video game industry. For instance, similar to Chess trainers, competing against opponents who are challenging but not overwhelmingly so is enjoyable, as is being able to rely on the advice of a dependable AI adviser. Gradually mastering a game is fun [1] and learning from an opponent's increasingly complex strategies is one way of creating satisfying game experiences for players. It resembles solving more and more difficult puzzles in a portal game, but the complexity is embodied in opponent behaviors rather than in the environment. Also, automated game testing and semi-automated game balancing would become a reality as learning AI systems would be able to adjust to game changes and expose imbalances.

Real-Time Strategy (RTS) games, such as Blizzard's StarCraft and The Creative Assembly's Total War game franchises, are war simulations in which players have to manage economies, create units and structures, and control them, taking into consideration the uncertainty about the opposing unit locations. These game mechanics are constrained by the creation or acquisition of limited resources that are part of the main economy of the game. Despite the research progress made, good human players can still defeat the best RTS game AI systems. Due to the recent success of AlphaGo [2] which defeated one of the strongest human Go players 4-1 in 2016, there is widespread confidence in the research community that the effectiveness of combining Monte Carlo Tree Search (MCTS) and deep learning techniques will, in the near future, produce an AI system able to defeat professional human RTS game players.

For the original AlphaGo system, Convolutional Neural Networks (CNNs) were trained to sample 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 (RL) techniques, to compute a probability distribution over all possible moves. These results were used to focus the search on the most promising branches. Second, MCTS state evaluation accuracy was improved by combining playout results with evaluations produced by a value network. Considering how successful combining CNN training with search is in Go, its application to RTS games seems promising, too.

In this paper, we apply CNNs to a simplified research RTS game (μRTS [3]) and Creative Assembly's popular commercial RTS game Total War: Warhammer (TW). In particular, we address three questions: 1) how to apply CNNs to RTS game state evaluation comparing it with various other evaluation functions by means of tournaments played by several state-of-the-art search algorithms; 2) how to train CNNs to predict the output of costly high-level search and freeing up most of the available time to run tactical search, and 3) how to apply CNNs to battles in the commercial TW game which poses more difficult challenges because of the lack of a forward model and by complex movement and targeting, which are crucial to games such as μRTS and StarCraft, being affected by additional game mechanics.

In what follows we first discuss related work, then describe how CNNs can be used to evaluate RTS game states and how to adapt them to predict strategic search algorithm choices, then provide preliminary results of using deep RL for combat scenarios in the TW game, and finish the paper with conclusions and proposing future work.

II. Background

Adversarial tree search algorithms are the standard approach for turn-based, 2-player, zero-sum, perfect information games. Even though RTS games are real-time, multiplayer, imperfect information games, until recently, tree search has been the method of choice for constructing RTS game agents. Methods were devised for serializing moves in real-time games [4], multiplayer matches of more than 2 players have been largely ignored by the research community, and while some research has been done with regards to imperfect information (e.g., [5]), the broad trend has been to either focus on the perfect information parts of the game [4], [6], [7], or to simply ignore the issue for the time being [8–10].

A. Adversarial Search Algorithms

After the initial wave of papers describing solutions on adapting tree search algorithms to this new setup (e.g., [11]), it became clear that the problem was too large for exhaustive search. Either smart sampling of nodes was needed [7], [12], [13], or some form of abstraction [8–10], or possibly both.

Portfolio greedy search [7] uses scripts that suggest unit actions to a search algorithm which then greedily chooses actions for each unit based on simulation results. Stratified Strategy Selection [13] applies the same idea of reducing the possible number of actions by only considering actions suggested by scripts and reducing the number of units by grouping them by type.

NaïveMCTS [3] treats the problem of assigning actions to multiple units as a Combinatorial Multi-Armed Bandit (CMAB) problem. Previously, as in UCTCD [7], RTS game CMABs were transformed into regular Multi-Armed Bandit (MAB) problems by treating each possible legal value combination as a different arm, losing the internal structure in the process. NaïveMCTS takes advantage of that internal structure by assuming that the overall reward distribution is just the sum of the reward distributions for each variable. Their proposed

Real-Time Strategy (RTS) games have attracted the interest of the AI research community due to their large state and action spaces, interesting sub-problems, and the availability of professional human players.

sampling scheme outperformed other MCTS variants in μRTS at the time.

A3N [14] combines NaïveMCTS with asymmetric action abstractions [15]. The action abstractions are defined for two sets of units, a restricted set and an unrestricted one. The unrestricted set encompasses units engaged in combat, and all legal moves are allowed for those. The remaining units, forming the restricted set, are only permitted to issue actions suggested by a set of scripts.

A different approach to action reduction by means of scripts is Puppet Search (PS) [8], which uses the scripts themselves as abstract actions. This allows for deeper look-ahead, because the scripts can be run for an arbitrary duration before a new node is inserted into the search tree. A tradeoff exists: a longer look-ahead time allows for deeper search horizon, while a shorter time explores a wider variety of scenarios.

All of the mentioned search algorithms have the need for an evaluation function in common. Even MCTS variants, when used in an RTS game context, limit their rollouts and apply an evaluation function. Previous attempts to evaluate RTS game states have focused mainly on combat. LTD2 [16], a handcrafted evaluation function, tries to estimate the life-time damage that units can inflict on the opponent. Probabilistic graphical models have been used [17] to take into account other features, such as range and armor, which results in a more accurate assessment of a combat's result. A model based on Lanchester's attrition laws [18] further improves combat prediction by taking into account units that start combat with partial hit-points and predicting the remaining army size for the winner. Global state evaluation with handcrafted features and weights optimized by logistic regression has been less successful [19]. All of these methods share a big flaw: a lack of positional understanding.

B. Deep Convolutional Neural Networks

CNNs have an outstanding record of extracting features and classifying 2D data, from images [20] to games [2], [21], [22]. The usually rich stream of observational data from video games is an ideal input for deep neural networks and their powerful function approximation and representation learning abilities. These networks can automatically find compact low-dimensional representations of high-dimensional data—such as text, images, and even raw pixels in 3D shooter games—which can then be used for classification, state evaluation, or action selection.

Two recent success stories show how powerful these methods are. The first is the development of an algorithm that learned to play 49 Atari 2600 video games at super-human level directly from screen pixels [22]. The second is the AlphaGo system that defeated a human world champion in Go [21], a landmark result comparable to Deep Blue winning against Chess world champion Kasparov in 1997 [23]. While still using a standard heuristic search algorithm, the traditional hand-crafted state evaluations that were used in Chess agents have been replaced by neural networks trained with both supervised and reinforcement learning. Later, the supervised learning component—which was based on human expert games—was replaced by self-play learning from scratch. The resulting AlphaGo Zero program is so strong that even human experts have a hard time analyzing its strategy [2].

C. Convolutional Neural Network Layers

When using CNNs, it is helpful to think about them in terms of neurons that can be arranged in 3 dimensions: width, height, and depth ($W \times H \times D$). Sometimes we might want to process multiple inputs simultaneously to fully take advantage of a powerful graphics card, and group many game states in a single batch to speed up computations. This would add an extra dimension, the batch size, which we will disregard here for simplicity.

A straightforward CNN is simply a sequence of layers, each transforming a 3D input volume of activations to a 3D output volume through a differentiable function. Some of these layers contain parameters, but others do not. Convolutional (CONV)

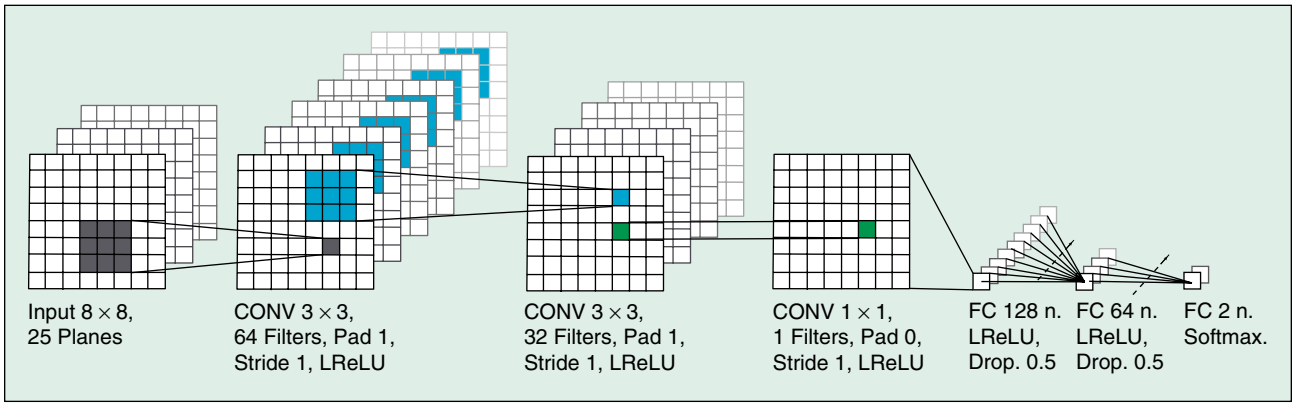


FIGURE 1 Basic Convolutional Neural Network Architecture. Some layers aggregated (e.g.: CONV+LReLU).

and fully connected (FC) layers perform computations that depend on the activations in the input volume as well as the parameters (neurons weights and biases). These parameters are trained with gradient descent such that the class scores that are the output of the CNN are consistent with the labels in the training data. Other layers, such as (Leaky) Rectified Linear Unit ((L)ReLU) layers and pooling (POOL) layers, implement fixed functions and do not have trainable parameters. Figure 1 shows a simple CNN architecture for analyzing small RTS game maps, annotated with short descriptions for each layer. The layers we use in this paper are:

Input: the game state for 8×8 to 128×128 maps in several channels—or planes—similar to the 3 RGB channels of an image. Each of these planes will correspond to different game features, such as unit type, health, or terrain.

Convolution: composed of a set of learnable filters, each connecting to only a local (across width and height) neighborhood, that extends through the full depth dimension.

LReLU: leaky rectified linear units compute the following piece-wise linear function: $f(x) = \alpha x$ if $x < 0$ and x otherwise.

Fully Connected Layer: every input is connected to every output node.

Global Average: value average across a plane. No learned weights.

Softmax: non-linearly normalizes the output vector so that each value is between 0 and 1, and all values add up to 1

D. Neural Network Training

In the original 2D image recognition tasks to which CNNs were first applied, supervised learning was used on labeled images. Training was accomplished by stochastic gradient descent to minimize a loss function. In multi-step tasks, in which labeling is harder, RL [24] can be used. In RL, an agent iteratively observes the environment, takes an action, and observes a reward for that action. RTS game domains are hard RL problems because the environment is partially observable, action results can be stochastic, and rewards are delayed.

To train our networks we use supervised learning and standard RL algorithms such as Deep Q-Network (DQN) learning [22] and the Asynchronous Advantage-Actor Critic (A3C) algorithm [25]. DQN is a sample efficient algorithm that uses a replay buffer to store past experiences (tuples containing a state, action chosen, reward received, and next state). Batches of random—and hopefully uncorrelated—experiences are drawn from the buffer and used for updates, forcing the network to generalize beyond what it is currently doing in the environment. In A3C multiple agents interact with the environment independently and simultaneously, and the diverse experiences are used to update a global, shared neural network. The main advantage of A3C over DQN is its capacity for mass parallelization, which reduces training time in a nearly linear fashion w.r.t. the number of agents working in parallel [25]. The agents' uncorrelated experiences serve a similar function as the DQN experience replay and improve learning stability, although at the cost of being less sample efficient. For more details we refer interested readers to surveys on deep reinforcement learning such as [26].

μRTS contains most features of a standard RTS game, while at the same time keeping things simple. The few unit and building types available have the same size, and there is only one resource type.

III. Learning State Evaluation and Strategy Selection in μRTS

In this section, we demonstrate how supervised machine learning techniques can be used to estimate state values and learning whole-game playing strategies. We also show how strategy learning and tactical search can be combined to achieve greater playing performance. Finally, we describe an attempt to lift the dependency on labelled data by means of RL, which—albeit eventually failing—inspired us to investigate learning in RTS games without forward models and at a smaller problem scale (combat), which will be reported in Section IV.

A. μRTS

In the experiments reported later, we use μRTS [3], 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 each, and there is only one resource type. μRTS supports configurable map sizes, commonly ranging from 8×8 to 128×128 tiles, and full observability is an option we have chosen for our experiments. The μRTS software repository features a few basic scripted players, as well as search-based players implementing several state-of-the-art RTS search techniques [8], [12], [27]—making it a useful tool for benchmarking new AI algorithms.

B. Supervised State Value Learning

Evaluation (or value) functions are commonly used in game-playing programs to estimate the value of a position for a given player. They map a state to a single number, usually representing either the probability of winning or the expected payoff difference between the players. 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.

1) Architecture

A common network input format inspired by AlphaGo's design [21] is a set of 2-dimensional binary planes that represent the state. Table I shows a μRTS example. The first six planes register the positions of all units. Each plane contains a 1 where there is a unit of a particular type, and 0 s everywhere else. The next five layers contain a 1-hot encoding of the unit's

health. The following two planes are masks for the owner of the units. Planes 14 to 18 contain 1-hot encodings that record how many frames are left for building or training units. These are all set to 0 if the unit in question is completely built or trained. The final seven planes indicate the amount of resources present on a tile (either by a mine holding resources, a worker carrying some after mining, or a base storing unused resources). These 25 planes are the input channels, while the input width W and height H will be determined by the specific map size.

The network architecture we chose (see Figure 2) uses a combination of small filter sizes with strides larger than 1 to limit the total number of weights. We also favor global averaging layers that have no tunable weights over fully connected layers. This results in a network with a constant number of weights, regardless of the input size, at the cost of the loss of spatial fidelity over the combined receptive size of the convolutional layers (22 in our case). The constant parameter size of the network helps to transfer learning results, for instance, by pretraining the network on a smaller map size, and later fine-tuning it on the full-size map, which can result in significant time savings. Figure 1 shows a more traditional architecture we have used for smaller ($\leq 16 \times 16$) map sizes. The output is a 2-way softmax, representing a probability distribution over two possible outcomes: first player win or loss.

TABLE I 25 Input feature planes for the neural network.

FEATURE	PLANES	DESCRIPTION
UNIT TYPE	1–6	BASE, BARRACKS, WORKER, LIGHT, RANGED, HEAVY
UNIT HEALTH POINTS	7–11	1, 2, 3, 4, OR ≥ 5
UNIT OWNER	12–13	MASKS TO INDICATE ALL UNITS BELONGING TO ONE PLAYER
FRAMES TO COMPLETION	14–18	0–25, 26–50, 51–80, 81–120, OR ≥ 121
RESOURCES	19–25	1, 2, 3, 4, 5, 6–9, OR ≥ 10

2) Training

The training data is a set of game states labeled with a Boolean value indicating whether the first player won. These can originate from human games if they are available, or as we have done, be generated by running a tournament between different AI players.

We want to cover as much of the input space as possible, while at the same time keeping in mind that both generating the data and training the network will get slower with every extra data point. The three ways we try to ensure coverage are: using players with very different strategies, using a diverse map set, and sampling states at different phases of the games. We take several random samples from each game, and while the best number of samples is hard to determine, we found the sweet spot to be between 6 and 12. More than that, and the samples are too correlated, which adds training time but doesn't increase the network's accuracy. Choosing less than that we need to spend more time running extra games or we

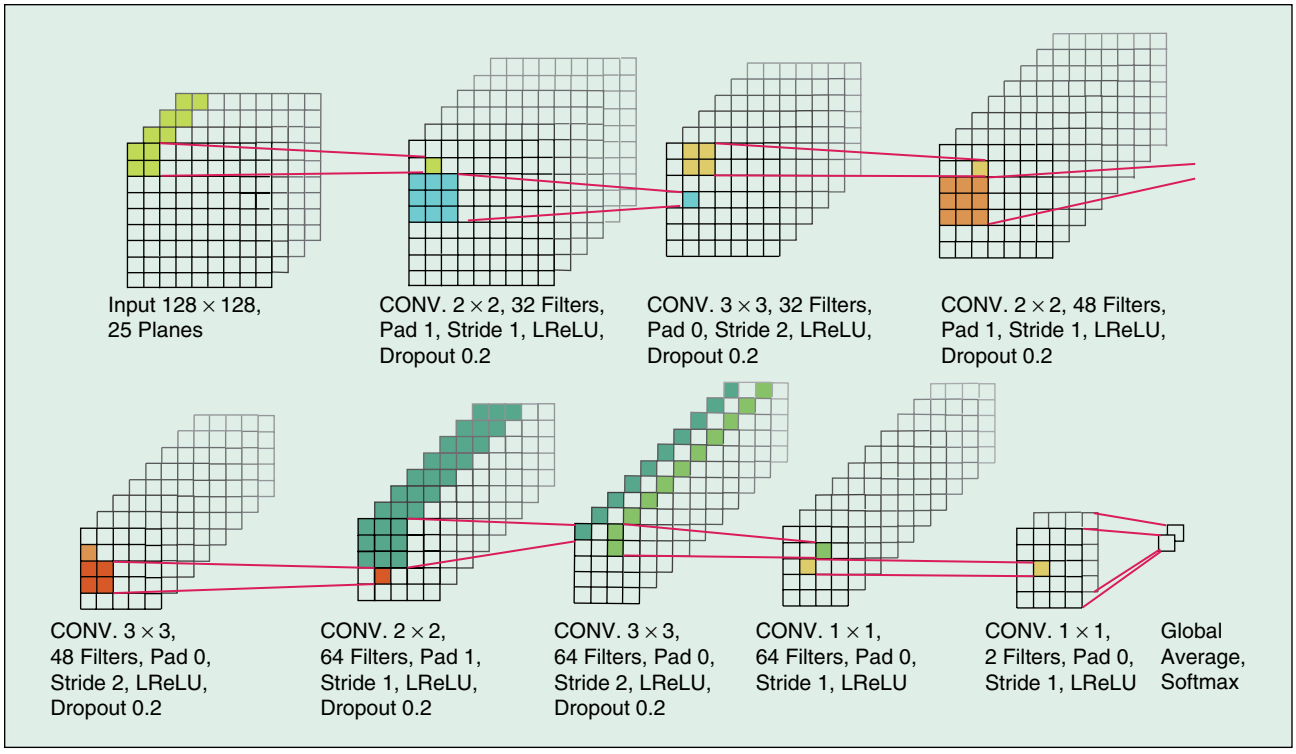


FIGURE 2 State evaluation network architecture.

risk not having enough data for your network to generalize properly.

We used the set of bots shipped with the μRTS system and 5 different maps, each with 12 different starting positions. Ties were discarded, and the remaining games were split into 2,190 training games, and 262 test games. These test games were used to validate the network's ability to generalize to previously unseen data. Twelve game states were randomly sampled from each game, for a total of 26,280 training samples and 3,144 test samples. Several evaluation functions were trained, for comparison, on the same dataset: the evaluation network, a Lanchester attrition law based function [18], a simple linear evaluation with hard-coded weights that comes with μRTS, and a version of the simple evaluation with weights optimized using logistic regression.

3) Results

The evaluation network reached 95% accuracy in predicting the winner of game states from the test samples. Figure 3 shows the accuracy of different evaluation functions as game time progresses. Table II shows the performance of the Puppet Search adversarial search algorithm when using the Lanchester evaluation function and the neural network. The performance of the network is significantly better (p-value = 0.0011) than Lanchester's, even though it is three orders of magnitude slower. Evaluating a game state using Lanchester takes an average of 2.7μs, while the evaluation network uses 2,574μs. LightRush and HeavyRush are used as baseline scripts and their descriptions can be found in [28].

C. Supervised Strategy Selection Learning

Most adversarial AI systems in commercial video games are scripted—or hand-authored—by designers. Finite-state machines, behavior trees and decision trees are the norm. In some game genres, such as first-person shooter games, these scripts control a single unit, while in others, like RTS games, they control all player's units. In both cases, several scripts can be used to implement different strategies, and realism and difficulty can be raised by smartly choosing when to switch between them [8]. Based on this work, here we will train a neural network to recognize which script, among four basic ones available in μRTS, is the best choice for a given game state.

1) Architecture Changes

Our evaluation network was a classifier charged with choosing between two categories: win or loss. The first change we need in this context is to have one output per possible scripted strategy. The second change is to add an extra input plane to indicate for which player should the network compute a strategy.

2) Training

Training is analogous to the previous network, but labels are now a choice of strategy rather than win/loss. To generate the labels we executed a 10 second Puppet Search on the same positions we used to train the evaluation network.

3) Results

The resulting policy network has an accuracy for predicting the correct puppet move of 73%, and a 95% accuracy for predicting any of the top 2 moves. Table III shows experimental results

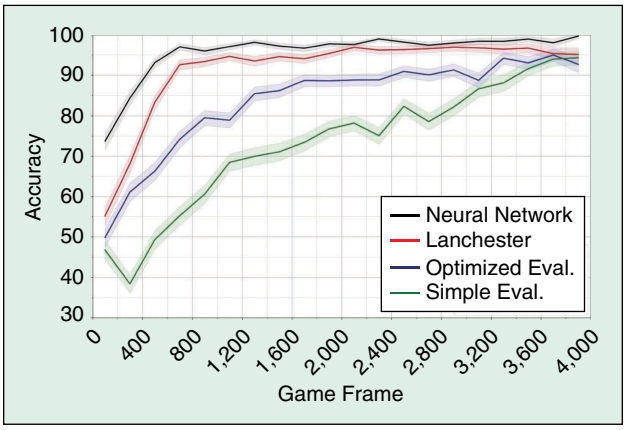


FIGURE 3 Evaluation accuracy of different value functions in µRTS. The accuracy of predicting the game winner is plotted against game time. Results are aggregated in 200 simulation frame buckets. Shaded areas represent one standard error.

TABLE II Evaluation network versus Lanchester: round-robin tournament using 60 different starting positions per match-up and 100 ms of computation time. Values indicate win percentage of row vs column agent.

	PS CNN	PS LANC.	LIGHT RUSH	HEAVY RUSH	AVG.
PS CNN	–	59.2	89.2	72.5	73.6
PS LANC.	40.8	–	64.2	67.5	57.5
LIGHTRUSH	10.8	35.8	–	71.7	39.4
HEAVYRUSH	27.5	32.5	28.3	–	29.4

TABLE III Tournament results of the policy network versus Puppet Search and fixed strategy scripts that produce certain unit types and send them to attack. Reported are round-robin results using 60 different starting positions per match-up. Values indicate win percentage of row vs column agent.

	POLICY NET.	PS	LIGHT RUSH	HEAVY RUSH	RANGED RUSH	WORKER RUSH	AVG.
POLICY NET.	–	44.2	94.2	71.7	100	61.7	61.9
PS	55.8	–	87.5	66.67	91.7	93.3	65.8
LIGHTRUSH	5.8	12.5	–	71.7	100	100	48.3
HEAVYRUSH	28.3	33.3	28.3	–	100	100	48.3
RANGEDRUSH	0	8.3	0	0	–	100	18.1
WORKERRUSH	38.3	6.7	0	0	0	–	7.5

We will define a script as a hand-coded function that takes a game state, and returns a set of actions to be performed. This script could be built with any of the standard commercial game AI techniques, such as Finite-State Machines (FSM), Behavior Trees (BT) or Utility Systems.

with Puppet Search applied to μRTS. The strength of the AI player using the network is close to one using search, while using only a fraction of the time and not requiring a forward model that allows us to accurately simulate action effects.

D. Combining Learned Strategy Selection with Tactical Search

A proposed technique for reducing search complexity in RTS games is to use action and/or state abstractions [8] which capture essential game properties. High-level abstractions, like build orders, can often lead to good strategic decision making, but tactical decision quality (i.e. which concrete action to take at a specific point in time) may suffer due to lost details. A competing method is to sample the search space [12] which often leads to good tactical performance in simple scenarios, but poor strategic planning. So, why not combine both ideas to generate tactically sound strategic actions?

For this purpose, we conducted experiments in μRTS (which allows look-ahead search) comparing the playing strength of a fully search-based agent with a hybrid player based on a policy network and tactical search [27]. The search-based agent chooses among high-level scripts and evaluates script choice sequences tactically using an MCTS variant. The hybrid player uses a policy network to select a script to execute next and utilizes tactical MCTS to execute the selected script. The policy network was trained using the search-based agent as described above. It performs slightly worse than the search algorithm but selects scripts much faster. The gained time can then be used to refine script actions tactically. In a round-robin tournament between these two algorithms, with 60 different starting positions per match-up, the hybrid player won 56.7% of the games against the search-based one. The overall winning rate of the hybrid and search-based players was 88.3% and 84.0% respectively. The full experimental details and results are reported in [27].

E. Reinforcement Learning Attempt

In an effort to eliminate our need for a forward model to label data based on search we implemented double DQN [29] with experience replay [22] and used our value network to provide a virtual reward in addition to the final game result. However, in all of our experiments the network converged to always selecting the script with the best average performance, regardless of the game state. Apart from possible implementation issues and parameter updates getting stuck in local extrema, we believe there are two issues that may complicate scripted strategy learning in full RTS games. First of all, the rewards are very sparse as they are only generated when a match ends—often after thousands of simulation frames. Secondly, the choice of actions near the end of the game is mostly inconsequential, because any action (script choice) will be able to successfully finish a game that is very nearly won, and conversely, if the position is very disadvantageous, no script choice will be able to reverse it. Moreover, when action choices matter most, at the beginning of the game, the virtual rewards generated by the value network are very small and noisy. We attempted to overcome this complication by starting the learning process on using endgame scenarios and slowly adding earlier states until full games were played, but without success. We intend to revisit this anomaly in future work.

IV. Case Study: Using Deep RL in Total War: Warhammer

The methods presented in the previous section are based on search algorithms and thus have the drawback of requiring forward models, which rarely are readily available, or easy to design and adjust. In this section, we present a case-study that uses RL as an end-to-end approach that does not rely on a forward model and apply it to a modern video game—Total War: Warhammer (TW). Compared to μRTS and even StarCraft, battles in TW pose more difficult challenges because complex movement and targeting is affected by additional game mechanics: units can get tired, a defender's orientation with respect to its attacker is crucial, there are effects that reward charging an enemy but discourage prolonged melee (cavalry) or the other way around (pike infantry), and morale is often more important for achieving victory than troop health. As a consequence, strategies such as pinning, refusing a flank while trying to overpower the other flank, and rear attacks on the enemy are required. Cooperation between units is much more important than in games such as StarCraft, and more complex behaviors have to be learned to defeat even the weak built-in game AI.

A screenshot of a typical TW battle can be seen in Figure 4. The goal of this internship project at Creative Assembly was to learn control policies for agents in cooperative/competitive environments such as TW battles. A diverse set of behaviors is required, such as: deciding when to pull out from a melee fight and when to switch targets, positioning of the ranged units, avoid crossing paths of allied units, maintaining a coordinated unit front when approaching the enemy, or pinning enemies while creating superiority elsewhere. RL, in particular, represents a natural fit to learn adaptive, autonomous and self-improving behavior in a multi-agent setting. CNNs have made it possible to extract high-level features from raw data, which enabled RL algorithms such as Q-learning to master difficult control policies without the help of hand-crafted features and with no tuning of the architecture or hyper-parameters for specific games [30].

A. Architecture

The network architecture chosen is similar to the ones used in Atari 2600 game AI research and early experiments in the StarCraft II Learning Environment (SC2LE) [31]. Lower dimensional input was used instead of raw pixels, as coarser grid representations worked well in previous experiments on μRTS and StarCraft combat scenarios. The 64 × 64 input feature planes (described at the end of the current subsection) were first processed by three convolutional layers. The extracted map representation was then passed through a fully connected layer then through a recurrent (LSTM) layer with 128 units both. The recurrent layer provides 'memory' functionality and helps in partial observability settings and with concepts such as unit velocity which cannot be inferred from a single observation/frame. The Q-values were represented using value and advantage functions as recommended by the dueling architecture [32]. This architecture is shown in Figure 5.

Due to the limited project time and resources available, the Independent Q-Learning (IQL) [33] paradigm was chosen. In IQL, the current agent is considered to be learning while all other agents are treated as part of the environment. Thus, the multi-agent learning task is decomposed into simultaneous single-agent problems. An alternative to IQL is centralized learning, where the joint Q-function is learned directly. This can lead to better agent coordination and results, but at the cost of increased

FIGURE 4 Example battle scenario in TotalWar: Warhammer.

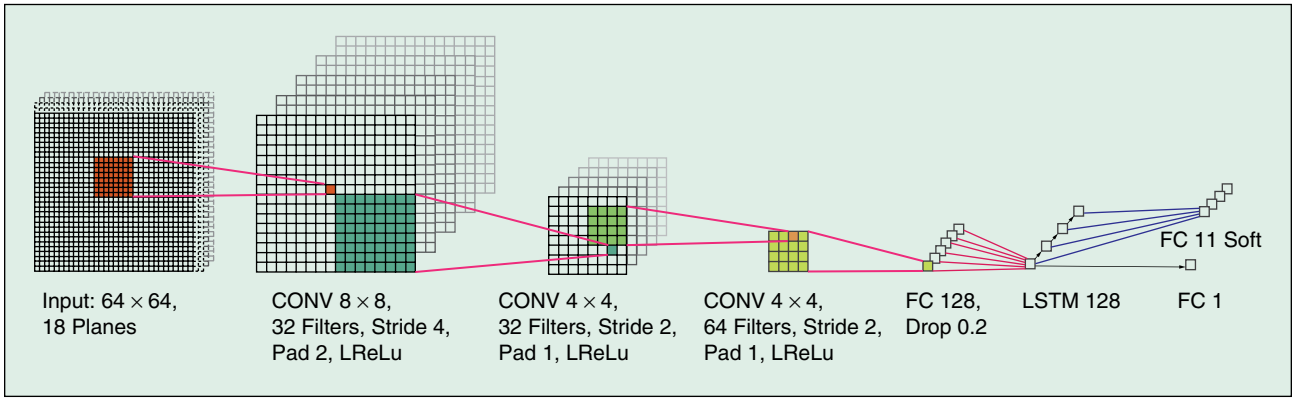


FIGURE 5 Actor-Critic Network Architecture for TW battle scenarios, assuming maximum of E = 6 enemies (number of input planes = 12 + E, number of possible actions = 5 + E).

learning time and poor scaling performance. Some methods are partially centralized, using one or more centralized critics to guide the optimization of decentralized policies in an actor-critic paradigm [34], [35]. To work well, these methods require additional information to be exchanged between agents (e.g., BicNet [36]). Furthermore, such on-policy methods are usually sample inefficient. In practice, the easy to implement IQL is a surprisingly strong benchmark method, even in mixed cooperative-competitive MARL problems [35].

To reduce the number of learnable parameters network weights were shared among agents. Using a set of network weights for each unit type (spear, melee, cavalry, and ranged) worked better than sharing the same set of parameters for all units. Training battles were fought against the built-in AI on the highest difficulty setting. The learning algorithm was based on DQN [22] with the dueling architecture update and multi-step returns. The replay buffer was chosen to contain only the most recent 1,000 games worth of experiences, to help with stability issues. The possible actions for each unit were chosen as: no-op, move in one of the 4 cardinal directions, and an attack action for each enemy unit in the scenario. The attack actions corresponding to units that are not shown in the local feature planes were masked out.

For each unit, the input state consisted of a number of 64 × 64 grid-like, local feature planes centered around the unit:

- 2 planes with Boolean values–own and enemy presence on each grid square;
- 4 Boolean planes–unit type by category: cavalry, archers, spear-men, other melee infantry;
- As many Boolean layers as enemy units used for masking and correlation with the attack enemy order;
- 4 planes–hit points, morale, orientation and stamina. Features are normalized by their maximum possible values;
- 2 planes–minimap data for both armies, providing information about the global state. The full map unit hit points data was scaled down to a 64 × 64 grid, and normalized. A value of 1 was added to the cell containing the acting agent, in the allied minimap.

Only one unit type was used for each of the 4 categories, so there was no obvious need of adding more features such as damage, armor or defense values as they do not change during the course of the battle and can be absorbed into the unit type.

B. Experimental Setting

For simplicity, small flat maps were used, without terrain features such as hills, forests or river crossings. There was no unit vision limitation or "fog of war" mechanism, but the proposed method could be extended to work with imperfect information, and several ideas for doing so are mentioned in Section V.

Battles in TW are driven by units: groups of specific troop types that can be deployed in formation. In our experiments, up to 6 units were used per army, each unit containing 40 to 100 soldiers depending on its type—infantry units usually contain more soldiers than cavalry, for example. In TW, battle orders are given to units and the game engine deals with the movement and attack animations of the individual soldiers within the unit, to maintain coherence.

Actions were emitted every 3 seconds for all units, which was also the default setting for the built-in AI. With these settings battles were often concluded in around 5 minutes or real-time play. To avoid dithering (e.g., running away from the enemy) games longer than 7.5 minutes of real-time play—which translates to up to 150 orders per unit—were stopped and treated as defeats. Because team reward signals do not directly relate to individual agent's actions, individual reward functions are commonly used for IQL and were used here as well. Reward shaping via potential-like functions was chosen, as it is one of the few methods that guarantee to preserve optimality w.r.t. the initial objective [37], [38]. The score function ϕ is a weighted sum of the unit's current health, morale and stamina values as well as the damage inflicted upon enemy units to both health and morale from the start of the game and a bonus term for each routed or killed enemy unit. The reward signal at time step t for unit i was $R_t^i = \phi(S_t^i) - \phi(S_{t-1}^i) - r_{\text{step}} + r_{\text{outcome}}$ where r_{step} is a small positive term to discourage time wasting and r_{outcome} is 1 if the battle ended with a victory and 0 otherwise.

C. Results

The obtained results were very good for scenarios where the DQN player controlled 1 or 2 units, the DQN agents defeating the built-in AI in over 90% of the games. Complex behaviors often used by human players were learned, such as cavalry units taking advantage of their impact bonus by repeatedly charging and retreating. When controlling a weaker melee unit and a cavalry unit against a stronger enemy melee unit, the system learned to use the melee unit to pin down the enemy and only then use the cavalry troop to maneuver and charge with the cavalry from the enemy's flank or rear, an example shown in video[1]. In this and all following videos shown in this subsection, the AI controls the highlighted troops with green health bars, while the enemy has red health bars.

When using more than 3 units per side, learning with DQN was unstable, probably due to the non-stationary aspect of the environment. The learning and exploration of other agents adds noise and instability, and the environment is changing so fast that games in the experience replay quickly lose relevance. This problem is discussed in more detail in other publications [39], and one of the proposed solutions is to switch to A3C, an on-policy actor-critic framework. With the new algorithm learning was more stable, even though less efficient w.r.t. the number of games played due to the lack of experience replay. In the next paragraphs, we report final win rates and standard deviations after training with 3 runs of A3C (with different starting seeds) for each scenario, computed using 200 games per seed. For the mixed scenarios, at the start of each game the army composition is generated randomly using

[1]Video: https://drive.google.com/file/d/1ZnJWhU3FiUYZ7ZIblmp_jT-FPItuuJGP/view

3 unit types, one each for cavalry, melee and missile. Both players use the same composition. The learning rate is 10^{-4}, discount factor 0.99, and the entropy coefficient $3{\cdot}10^{-2}$. Each worker commits a gradient update to the central parameter server every 20 gradient steps.

On a symmetric 3v3 scenario with mixed units, after 70k games played the algorithm reached 82% (±3%) win rate. When scaling up and training from scratch on 6v6 scenarios, the learning required more than twice the time and plateaued after 150k games, at 73% (±5%) win rate. Further increases of army sizes seemed achievable, but were not attempted due to the limited time and computational resources available.

The following examples show battles with agents trained via A3C, the next two using agents that learned on 3v3 scenarios with mixed units. In the second video[2] each side controls 3 identical melee units and the agents display a coordinated unit front while closing the distance to fight. They use flank attacks and better positioning to bring more troops in contact, fighting on the frontline. In the third video[3] there are 2 melee units and 1 ranged unit per side. The ranged agent avoids the incoming melee units and takes out the enemy ranged unit, while the melee units surround the enemy melee and finish them off with subsequent help from the ranged agent. Finally, the last example[4] shows agents trained on the 6v6 scenarios controlling 4 melee and 2 ranged units. The melee units out-manoeuver the enemy and obtain a better front, while the ranged units help focus damage on the same melee units. The leftover enemy ranged are easy to deal with afterwards.

Compared to training directly on the last scenario, learning was faster when using a curriculum based approach. Army size was extended progressively from 1 to 6, and for each scenario the hit points of enemy units were gradually increased from 60% to 100%. Scenario switching was done when the algorithm reached 70% win rate on the current task, up until reaching the final task. With these settings a higher win rate of 77% (±3%) was achieved for the 6v6 scenarios, using a small number of games for training – 110k.

D. Hierarchical Reinforcement Learning

The results described above confirmed that the proposed method, even with basic architecture choices and without extensive hyper-parameter optimization, worked well for the intended purposes. During the last part of the project we investigated scaling towards scenarios with more agents and reducing the number of games required for training. An approach based on hierarchical reinforcement learning (HRL) was chosen, as it is a promising approach to expand traditional RL methods to work with more difficult and complex tasks.

A two-layer hierarchical structure was used, with the high-level policy learning to plan over an extended time period by choosing goals to be implemented by the low-level policy. It observed the current state every $k = 10$ steps and chose a goal to be achieved from {approach, outflank, attack, disengage} and a target enemy unit for the goal. The low-level policies observed the state and the goal every time step, and produced an in-game action—from the same action set as in previous experiments—which was applied directly to the environment. Intrinsic rewards provided by the high-level controller were used to train the low-level policies, based only on how well the different goals were accomplished. For the high-level policy, the default rewards provided by the environment were summed up over the 10 steps and used as feedback.

We did not obtain good results when training both high- and low-level policies jointly, the win rate never going over 40% in the 6v6 scenario even after 200k games. When the low-level policies were trained separately, the disengage and approach objectives were learnt less than 5k games, the outflank one in 8k games and the attack objective in 25k games. However, even with pre-trained and then fixed low-level policies, the high-level policy was not able to surpass 50% win rate. This suggests that the designed goals and their hand-crafted auxiliary rewards did not map very well to the main task to be solved—winning the battle. Moreover, game results are very sensitive to one bad high-level decision: a poorly timed withdraw order can expose one's flank to being rear-charged, which may lead to morale dropping in a chain reaction and a subsequent defeat. To minimize this effect, we suggest learning or using fewer but more comprehensive low-level policies: for example reducing our goal set to 4 by not using target individual enemy units, or using complete action scrips as in Puppet Search, introduced in Subsection II-A. Another approach is to use an extended set of simpler, lower impact behaviors. For example, a HRL approach using a set of 165 hard-coded macro actions as the low-level has recently been successful in a full-game StarCraft II AI [40]. Even though time constraints prevented reaching good results with HRL in this project, we believe that spatial and/or temporal abstractions are needed for complex or difficult environments such as RTS games.

V. Conclusions and Future Work

Deep convolutional neural networks are powerful non-linear function approximators that can be trained from observed or generated data. In this paper we first showed examples of how they can be used to evaluate RTS game states and select high-level strategies quickly and accurately—speeding up computationally demanding high-level search tasks and using the gained time to improve tactical searches, thereby increasing the overall decision quality. The availability of powerful machine learning frameworks that can be applied to vast game replay data, in addition to accelerated research efforts in this area, will likely lead to game AI breakthroughs in the near future that will revolutionize game design and playing.

[2]Video: https://drive.google.com/file/d/1SiQH5FNEEVWyl1T3wUoK8YlwTTL-5tXt/view

[3]Video: https://drive.google.com/file/d/1dd8On9o3d_gxkWjYbRPgs4_QrnBZm1ow/view

[4]Video: https://drive.google.com/file/d/1etkbNw0XEqz1XScTtQmFkSFDvey68jA-/view

We then described how to eliminate the need for simulators or forward models by using reinforcement learning to learn autonomous, self-improving behaviors for cooperative-competitive environments with sparse rewards such as Total War battles. We showed how a basic architecture design can be used with techniques such as curriculum learning and reward shaping to learn diverse and complex behaviors for scenarios with up to 6v6 units of mixed types. The highest difficulty built-in AI was defeated in 77% of the games. To increase data efficiency and to scale better to scenarios with more units, we attempted a hierarchical RL approach, but obtained mixed results, likely because of poor goal abstraction choices. A few options to be explored in future work have been mentioned in Subsection IV-D.

In addition, other data efficient methods should be investigated. One idea is to use 'world models', which learn compressed spatial and temporal representations of the environment that can be trained relatively quickly in an unsupervised manner [41]. Then agents can be trained, even entirely based on "hallucinated" action trajectories generated by the world model. It has been shown that such policies can transfer surprisingly well to actual environments. We found that designing good individual reward functions is very important to the overall learning stability, sometimes requiring extensive tuning to obtain convincing results. Using team reward signals and then learning how to assign the credit is a more elegant solution which proved successful in other cooperative multi-agent environments [42], [43].

VI. Acknowledgments

This work is based on material presented in two conference papers [27] and [28] and internship work at Creative Assembly.

References

[1] R. Koster and W. Wright, *A Theory of Fun for Game Design*. Paraglyph Press, 2004.
[2] D. Silver et al., "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, Oct. 2017.
[3] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to RTS games," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment*, Oct. 2013, pp. 58–64.
[4] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for RTS game combat scenarios," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Oct. 2012, pp. 112–117.
[5] A. Uriarte and S. Ontanón, "Single believe state generation for partially observable real-time strategy games," in *Proc. IEEE Conf. Computational Intelligence and Games (CIG)*, Aug. 2017, pp. 296–303.
[6] D. Churchill and M. Buro, "Incorporating search algorithms into RTS game agents," in *Proc. AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*, Oct. 2012.
[7] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in StarCraft," in *Proc. IEEE Conf. Computational Intelligence in Games (CIG)*, Oct. 2013, pp. 1–8.
[8] N. A. Barriga, M. Stanescu, and M. Buro, "Game tree search based on nondeterministic action scripts in real-time strategy games," *IEEE Trans. Games*, vol. 10, no. 1, pp. 69–77, Mar. 2018.
[9] S. Ontañón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Proc. Int. Joint Conf. Artificial Intelligence (IJCAI)*, July 2015, pp. 1652–1658.
[10] A. Uriarte and S. Ontañón, "Game-tree search over high-level game states in RTS games," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Oct. 2014, pp. 73–79.
[11] D. Churchill and M. Buro, "Incorporating search algorithms into RTS game agents," in *Proc. AIIDE Workshop on Artificial Intelligence in Adversarial Real-Time Games*, 2012.
[12] S. Ontañón, "Combinatorial multi-armed bandits for real-time strategy games," *J. Artif. Intell. Res.*, vol. 58, pp. 665–702, Mar. 2017.
[13] L. H. Lelis, "Stratified strategy selection for unit control in real-time strategy games," in *Proc. Int. Joint Conf. Artificial Intelligence (IJCAI)*, Aug. 2017, pp. 3735–3741.
[14] R. O. Moraes, J. R. Mariño, L. H. Lelis, and M. A. Nascimento, "Action abstractions for combinatorial multi-armed bandit tree search," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Nov. 2018, pp. 74–80.
[15] R. O. Moraes and L. H. Lelis, "Asymmetric action abstractions for multi-unit control in adversarial real-time games," in *Proc. AAAI Conf. Artificial Intelligence (AAAI)*, Feb. 2017, pp. 876–883.
[16] A. Kovarsky and M. Buro, "Heuristic search applied to abstract combat games," in *Proc. Advances in Artificial Intelligence*, May 2005, pp. 66–78.
[17] M. Stanescu, S. P. Hernandez, G. Erickson, R. Greiner, and M. Buro, "Predicting army combat outcomes in StarCraft," in *Proc. AAAI Artificial Intelligence and Interactive Digital Entertainment Conf. (AIIDE)*, Oct. 2013, pp. 86–92.
[18] M. Stanescu, N. A. Barriga, and M. Buro, "Using Lanchester attrition laws for combat prediction in StarCraft," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Nov. 2015, pp. 86–92.
[19] G. Erickson and M. Buro, "Global state evaluation in StarCraft," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, AAAI Press, Oct. 2014, pp. 112–118.
[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Advances in Neural Information Processing Systems (NIPS)*, Dec. 2012, pp. 1097–1105.
[21] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.
[22] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.
[23] M. Campbell, A. J. Hoane, and F.-H. Hsu, "Deep Blue," *Artif. Intell.*, vol. 134, no. 1, pp. 57–83, Jan. 2002.
[24] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, Mar. 1998.
[25] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in *Proc. Int. Conf. Machine Learning (ICML)*, June 2016, pp. 1928–1937.
[26] Y. Li, "Deep reinforcement learning," *CoRR*, vol. abs/1810.06339, 2018. [Online]. Available: http://arxiv.org/abs/1810.06339
[27] N. A. Barriga, M. Stanescu, and M. Buro, "Combining strategic learning and tactical search in real-time strategy games," in *Proc. AAAI Conf. Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, Oct. 2017, pp. 9–15.
[28] M. Stanescu, N. A. Barriga, A. Hess, and M. Buro, "Evaluating real-time strategy game states using convolutional neural networks," in *Proc. IEEE Conf. Computational Intelligence and Games (CIG)*, Sept. 2016.
[29] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artificial Intelligence (AAAI)*, Feb. 2016, pp. 2094–2100.
[30] V. Mnih et al., "Playing Atari with deep reinforcement learning," in *Proc. NIPS Deep Learning Workshop*, Dec. 2013.
[31] O. Vinyals et al., "StarCraft II: A new challenge for reinforcement learning," DeepMind, Blizzard, Tech. Rep., 2017.
[32] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online]. Available: http://arxiv.org/abs/1511.06581
[33] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proc. Int. Conf. Machine Learning (ICML)*, June 1993, pp. 330–337.
[34] J. N. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients," *CoRR*, vol. abs/1705.08926, 2017. [Online]. Available: http://arxiv.org/abs/1705.08926
[35] J. Z. Leibo, V. F. Zambaldi, M. Lanctot, J. Marecki, and T. Graepel, "Multi-agent reinforcement learning in sequential social dilemmas," *CoRR*, vol. abs/1702.03037, 2017. [Online]. Available: http://arxiv.org/abs/1702.03037
[36] P. Peng et al., "Multiagent bidirectionally-coordinated nets for learning to play StarCraft combat games," *CoRR*, vol. abs/1703.10069, 2017. [Online]. Available: http://arxiv.org/abs/1703.10069
[37] S. Devlin, L. Yliniemi, D. Kudenko, and K. Tumer, "Potential-based difference rewards for multiagent reinforcement learning," in *Proc. Int. Conf. Autonomous Agents and Multi-Agent Systems*, May 2014, pp. 165–172.
[38] A. Eck, L.-K. Soh, S. Devlin, and D. Kudenko, "Potential-based reward shaping for finite horizon online POMDP planning," *Auton. Agents Multi-Agent Syst.*, vol. 30, no. 3, pp. 403–445, Mar. 2016.
[39] J. Foerster, N. Nardelli, G. Farquhar, P. H. S. Torr, P. Kohli, and S. Whiteson, "Stabilising experience replay for deep multi-agent reinforcement learning," in *Proc. Int. Conf. Machine Learning (ICML)*, Aug. 2017.
[40] P. Sun et al., "TStarBots: Defeating the cheating level builtin AI in StarCraft II in the full game," arXiv Preprint, arXiv:1809.07193, 2018.
[41] D. Ha and J. Schmidhuber, "World models," *CoRR*, vol. abs/1803.10122, 2018. [Online]. Available: http://arxiv.org/abs/1803.10122
[42] P. Sunehag et al., "Value-decomposition networks for cooperative multi-agent learning," *CoRR*, vol. abs/1706.05296, 2017. [Online]. Available: http://arxiv.org/abs/1706.05296
[43] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. N. Foerster, and S. Whiteson, "QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning," *CoRR*, vol. abs/1803.11485, 2018. [Online]. Available: http://arxiv.org/abs/1803.11485