Grandmaster level in StarCraft II using multi-agent reinforcement learning

Oriol Vinyals[*1†], Igor Babuschkin[*1], Wojciech M. Czarnecki[*1], Michaël Mathieu[*1], Andrew Dudzik[*1], Junyoung Chung[*1], David H. Choi[*1], Richard Powell[*1], Timo Ewalds[*1], Petko Georgiev[*1], Junhyuk Oh[*1], Dan Horgan[*1], Manuel Kroiss[*1], Ivo Danihelka[*1], Aja Huang[*1], Laurent Sifre[*1], Trevor Cai[*1], John P. Agapiou[*1], Max Jaderberg[1], Alexander S. Vezhnevets[1], Rémi Leblond[1], Tobias Pohlen[1], Valentin Dalibard[1], David Budden[1], Yury Sulsky[1], James Molloy[1], Tom L. Paine[1], Caglar Gulcehre[1], Ziyu Wang[1], Tobias Pfaff[1], Yuhuai Wu[1], Roman Ring[1], Dani Yogatama[1], Dario Wünsch[2], Katrina McKinney[1], Oliver Smith[1], Tom Schaul[1], Timothy Lillicrap[1], Koray Kavukcuoglu[1], Demis Hassabis[1], Chris Apps[*1], David Silver[*1†]

[1] DeepMind, 6 Pancras Square, N1C4AG, London, United Kingdom
[2] Team Liquid, Huis te Zuylenlaan 75, 3554 JE, Utrecht, Netherlands
[*] equal contributions
[†] corresponding author

Many real-world applications require artificial agents to compete and coordinate with other agents in complex environments. As a stepping stone to this goal, the domain of StarCraft has emerged as an important challenge for artificial intelligence research, owing to its iconic and enduring status among the most difficult professional esports, and its relevance to the real world in terms of its raw complexity and multi-agent challenges. Over the course of a decade and numerous competitions[1–3], the strongest agents have simplified important aspects of the game, utilised superhuman capabilities, or employed hand-crafted subsystems. Despite these advantages, no previous agent has come close to matching the overall skill of top StarCraft players. We chose to address the challenge of StarCraft using general-purpose learning methods that are in principle applicable to other complex domains: a multi-agent reinforcement learning algorithm that uses data from both human and agent games within a diverse league of continually adapting strategies and counter-strategies, each represented by deep neural networks[5,6]. We evaluated our agent, AlphaStar, in the full game of StarCraft II, through a series of online games against human players. AlphaStar was rated at Grandmaster level for all three StarCraft races and above 99.8% of officially ranked human players.

StarCraft is a real-time strategy game[a] in which players balance high-level economic decisions with individual control of hundreds of units. This domain raises important game-theoretic challenges: it features a vast space of cyclic, non-transitive strategies and counter-strategies; discovering novel strategies is intractable with naive self-play exploration methods; and those strategies may not be effective when deployed in real-world play with humans. Furthermore, StarCraft has a combinatorial action space, a planning horizon that extends over thousands of real-time decisions, and imperfect information[7].

Each game consists of tens of thousands of time-steps and thousands of actions, selected in real-time throughout approximately ten minutes of gameplay. At each step t, our agent AlphaStar receives an observation o_t that includes a list of all observable units and their

[a] StarCraft is a franchise from Blizzard Entertainment. The franchise comprises StarCraft: Brood War and StarCraft II. In this paper, we used StarCraft II.

attributes. This information is imperfect; the game includes opponent units seen by the player's own units, and excludes some opponent unit attributes outside the camera view.

Each action a_t is highly structured: it selects *what* action type, out of several hundred (for example, move or build worker); *who* to issue that action to, for any subset of the agent's units; *where* to target, among locations on the map or units within the camera view; and *when* to observe and act next (Fig. 1A). This representation of actions results in approximately 10^{26} possible choices at each step. Similar to human players, a special action is available to move the camera view, so as to gather more information.

Humans play StarCraft under physical constraints that limit their reaction time and the rate of their actions. The game was designed with those limitations in mind, and removing those constraints changes the nature of the game. We therefore chose to impose constraints upon AlphaStar: it suffers from delays due to network latency and computation time; and its actions per minute (APM) are limited, with peak statistics substantially lower than those of humans (Figs. 2C, 3G for performance analysis). AlphaStar's play with this interface and these constraints was approved by a professional player (see 'Professional player statement' in Methods).

Learning Algorithm

To address the complexity and game-theoretic challenges of StarCraft, AlphaStar uses a combination of new and existing general-purpose techniques for neural network architectures, imitation learning, reinforcement learning, and multi-agent learning. Further details about these techniques are given in the Methods.

Central to AlphaStar is a policy $\pi_\theta(a_t|s_t, z) = \mathbb{P}[a_t|s_t, z]$, represented by a neural network with parameters θ that receives all observations $s_t = (o_{1:t}, a_{1:t-1})$ from the start of the game as inputs, and selects actions as outputs. The policy is also conditioned on a statistic z that summarises a strategy sampled from human data (for example, a build order).

Our agent architecture consists of general-purpose neural networks components that handle StarCraft's raw complexity. Observations of player and opponent units are processed using a self-attention mechanism[8]. To integrate spatial and non-spatial information, we introduce scatter connections. To deal with partial observability, the temporal sequence of observations is processed by a deep long short-term memory (LSTM) system[9]. To manage the structured, combinatorial action space, the agent uses an auto-regressive policy[7,10,11] and recurrent pointer network[12]. Extended Data Fig. 3 summarizes the architecture and Fig. 3F shows an ablation of each component.

Agent parameters were initially trained by supervised learning. Games were sampled from a publicly available dataset of anonymized human replays. The policy was then trained to predict each action a_t, conditioned either solely on s_t, or also on z. This results in a diverse set of strategies that reflects the modes of human play.

The agent parameters were subsequently trained by a reinforcement learning algorithm that is designed to maximize the win rate (that is, compute a best response) against a mixture of opponents. The choice of opponent is determined by a multi-agent procedure, described below. AlphaStar's reinforcement learning algorithm is based on a policy gradient algorithm similar to advantage actor-critic[13]. Updates were applied asynchronously[14] on replayed

experiences[15]. This requires an approach known as off-policy learning[5], that is, updating the current policy from experience generated by a previous policy. Our solution is motivated by the observation that, in large action spaces, the current and previous policies are highly unlikely to match over many steps. We therefore use a combination of techniques that can learn effectively despite the mismatch: temporal difference learning (TD(λ))[16], clipped importance sampling (V-trace)[14], and a new self-imitation[17] algorithm (UPGO) that moves the policy towards trajectories with better-than-average reward. To reduce variance, during training only, the value function is estimated using information from both the player's and the opponent's perspectives. Figure 3I, K analyses the relative importance of these components.

One of the main challenges in StarCraft is to discover novel strategies. Consider a policy that has learned to build and utilize the micro-tactics of ground units. Any deviation that builds and naively uses air units will reduce performance. It is highly improbable that naive exploration will execute a precise sequence of instructions, over thousands of steps, that constructs air units and effectively utilizes their micro-tactics. To address this issue, and to encourage robust behaviour against likely human play, we utilize human data. Each agent is initialized to the parameters of the supervised learning agent. Subsequently, during reinforcement learning, we either condition the agent on a statistic z, in which case agents receive a reward for following the strategy corresponding to z, or train the agent unconditionally, in which case the agent is free to choose its own strategy. Agents also receive a penalty whenever their action probabilities differ from the supervised policy. This human exploration ensures that a wide variety of relevant modes of play continue to be explored throughout training. Figure 3E shows the importance of human data in AlphaStar.

To address the game-theoretic challenges, we introduce league training, an algorithm for multi-agent reinforcement learning (Fig. 1B, C). Self-play algorithms, similar to those used in chess and Go[18], learn rapidly but may chase cycles (for example, where A defeats B, and B defeats C, but A loses to C) indefinitely without making progress[19]. Fictitious self-play (FSP)[20–22] avoids cycles by computing a best response against a uniform mixture of all previous policies; the mixture converges to a Nash equilibrium in two-player zero-sum games[20]. We extend this approach to compute a best response against a non-uniform mixture of opponents. This league of potential opponents includes a diverse range of agents (Fig. 4D), as well as their policies from both current and previous iterations. At each iteration, each agent plays games against opponents sampled from a mixture policy specific to that agent. The parameters of the agent are updated from the outcomes of those games by the actor-critic reinforcement learning procedure described above.

The league consists of three distinct types of agent, differing primarily in their mechanism for selecting the opponent mixture. First, the main agents utilize a prioritized fictitious self-play (PFSP) mechanism that adapts the mixture probabilities proportionally to the win rate of each opponent against the agent; this provides our agent with more opportunities to overcome the most problematic opponents. With fixed probability, a main agent is selected as an opponent; this recovers the rapid learning of self-play (Fig. 3C). Second, main exploiter agents play only against the current iteration of main agents. Their purpose is to identify potential exploits in the main agents; the main agents are thereby encouraged to address their weaknesses. Third, league exploiter agents use a similar PFSP mechanism to the main agents, but are not targeted by main exploiter agents. Their purpose is to find systemic weaknesses of the entire league. Both main exploiters and league exploiters are periodically reinitialized to

encourage more diversity and may rapidly discover specialist strategies that are not necessarily robust against exploitation. Figure 3B analyses the choice of agents within the league.

In StarCraft, each player chooses one of three races — Terran, Protoss or Zerg — each with distinct mechanics. We trained the league using three main agents (one for each StarCraft race), three main exploiter agents (one for each race), and six league exploiter agents (two for each race). Each agent was trained using 32 third-generation tensor processing units (TPUs[23]) over 44 days. During league training almost 900 distinct players were created.

Empirical Evaluation

We evaluated the three main Terran, Protoss and Zerg AlphaStar agents using the unconditional policy on the official online matchmaking system Battle.net. Each agent was assessed at three different snapshots during training: after supervised training only (AlphaStar Supervised), after 27 days of League training (AlphaStar Mid), and after 44 days of league training (AlphaStar Final). AlphaStar Supervised and AlphaStar Mid were evaluated starting from an unranked rating on Battle.net for 30 and 60 games, respectively, for each race; AlphaStar Final was evaluated from AlphaStar Mid's rating for an additional 30 games for each race. The Battle.net matchmaking procedure selected maps and opponents. Matches were played under blind conditions: AlphaStar was not provided with the opponent's identity, and played under an anonymous account. These conditions were selected to estimate AlphaStar's strength under approximately stationary conditions, but do not directly measure AlphaStar's susceptibility to exploitation under repeated play.

AlphaStar Final achieved ratings of 6,275 Match Making Rating (MMR) for Protoss, 6,048 for Terran and 5,835 for Zerg, placing it above 99.8% of ranked human players, and at Grandmaster level for all three races (Fig. 2A and Extended Data Fig. 7 (analysis), Supplementary Data, Replays (game replays)). AlphaStar Supervised reached an average rating of 3,699, which places it above 84% of human players and shows the effectiveness of supervised learning.

To further analyze AlphaStar we also ran several internal ablations (Fig. 3) and evaluations (Fig. 4). For multi-agent dynamics, we ran a round-robin tournament of all players throughout League training, and a second tournament of main agents against held-out validation agents trained to follow specific human strategies. The main agent performance improved steadily across all three races. The performance of the main exploiters actually reduced over time and main agents performed better against the held-out validation agents, both of which suggest that the main agents grew increasingly robust. The league Nash equilibrium over all players at each point in time assigns small probabilities to players from previous iterations, suggesting that the learning algorithm does not cycle or regress. Finally, the unit composition changed throughout league training, which indicates a diverse strategic progression.

Conclusion

AlphaStar is the first agent to achieve Grandmaster level in StarCraft II, and the first to reach the highest league of human players in a widespread professional esport without simplification of the game. Like StarCraft, real-world domains such as personal assistants,

self-driving cars, or robotics require real-time decisions, over combinatorial or structured action spaces, given imperfectly observed information. Furthermore, similar to StarCraft, many applications have complex strategy spaces that contain cycles or hard exploration landscapes, and agents may encounter unexpected strategies or complex edge cases when deployed in the real world. The success of AlphaStar in StarCraft II suggests that general-purpose machine learning algorithms may have a substantial effect on complex real-world problems.

References

1. AIIDE StarCraft AI Competition. Available at: https://www.cs.mun.ca/~dchurchill/starcraftaicomp/.
2. Student StarCraft AI Tournament and Ladder. Available at: https://sscaitournament.com/.
3. Starcraft 2 AI ladder. Available at: https://sc2ai.net/.
4. Churchill, D., Lin, Z. & Synnaeve, G. An Analysis of Model-Based Heuristic Search Techniques for StarCraft Combat Scenarios. in *Artificial Intelligence and Interactive Digital Entertainment Conference* (2017).
5. Sutton, R. & Barto, A. *Reinforcement Learning: An Introduction*. (MIT Press, 1998).
6. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* **521**, 436 (2015).
7. Vinyals, O. *et al.* StarCraft II: A New Challenge for Reinforcement Learning. *arXiv Prepr. arXiv1708.04782* (2017).
8. Vaswani, A. *et al.* Attention Is All You Need. in *Advances in Neural Information Processing Systems* 5998–6008 (2017).
9. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural Comput.* **9**, 1735–1780 (1997).
10. Mikolov, T., Karafiat, M., Burget, L., Cernocky, J. & Khudanpur, S. Recurrent Neural Network based Language Model. in *INTERSPEECH* 1045–1048 (2010).
11. Metz, L., Ibarz, J., Jaitly, N. & Davidson, J. Discrete Sequential Prediction of Continuous Actions for Deep RL. *arXiv Prepr. arXiv1705.05035* (2017).
12. Vinyals, O., Fortunato, M. & Jaitly, N. Pointer Networks. in *Advances in Neural Information Processing Systems* 2692–2700 (2015).
13. Mnih, V. *et al.* Asynchronous Methods for Deep Reinforcement Learning. in *International Conference on Machine Learning* 1928–1937 (2016).
14. Espeholt, L. *et al.* IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. in *International Conference on Machine Learning* 1406–1415 (2018).
15. Wang, Z. *et al.* Sample Efficient Actor-Critic with Experience Replay. in *International Conference on Learning Representations* (2017).

16. Sutton, R. Learning to predict by the method of temporal differences. *Mach. Learn.* **3**, 9–44 (1988).

17. Oh, J., Guo, Y., Singh, S. & Lee, H. Self-Imitation Learning. in *International Conference on Machine Learning* 3875–3884 (2018).

18. Silver, D. *et al.* A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* **362**, 1140–1144 (2018).

19. Balduzzi, D. *et al.* Open-ended Learning in Symmetric Zero-sum Games. in *International Conference on Machine Learning* 434–443 (2019).

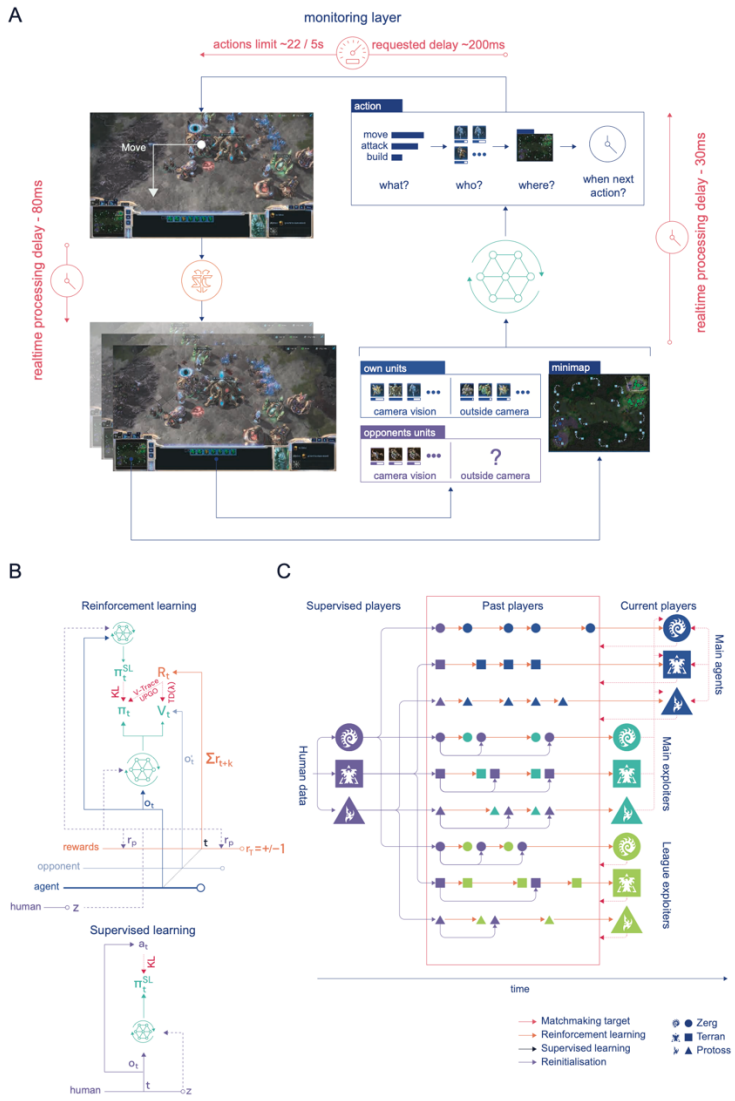
20. Brown, G. W. Iterative solution of games by fictitious play. *Act. Anal. Prod. Alloc.* **13**, 374–376 (1951).

21. Leslie, D. S. & Collins, E. J. Generalised weakened fictitious play. *Games Econ. Behav.* **56**, 285–298 (2006).

22. Heinrich, J., Lanctot, M. & Silver, D. Fictitious Self-Play in Extensive-Form Games. in *International Conference on Machine Learning* 805–813 (2015).

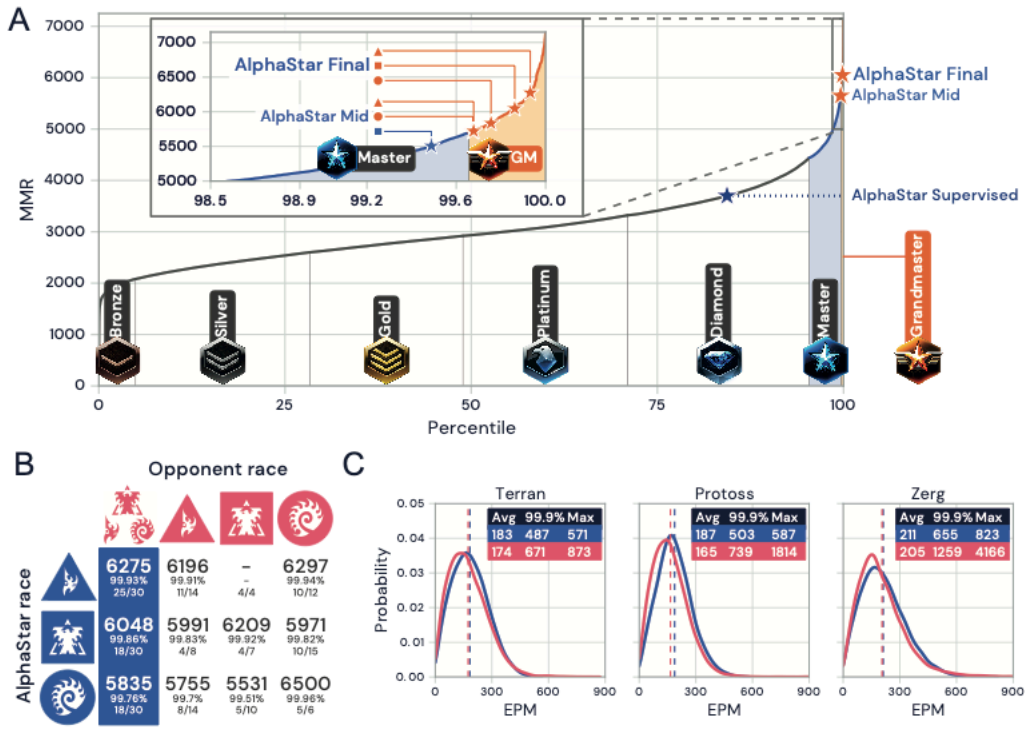
23. Jouppi, N. P., Young, C., Patil, N. & others. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. (2017).

Figure 1



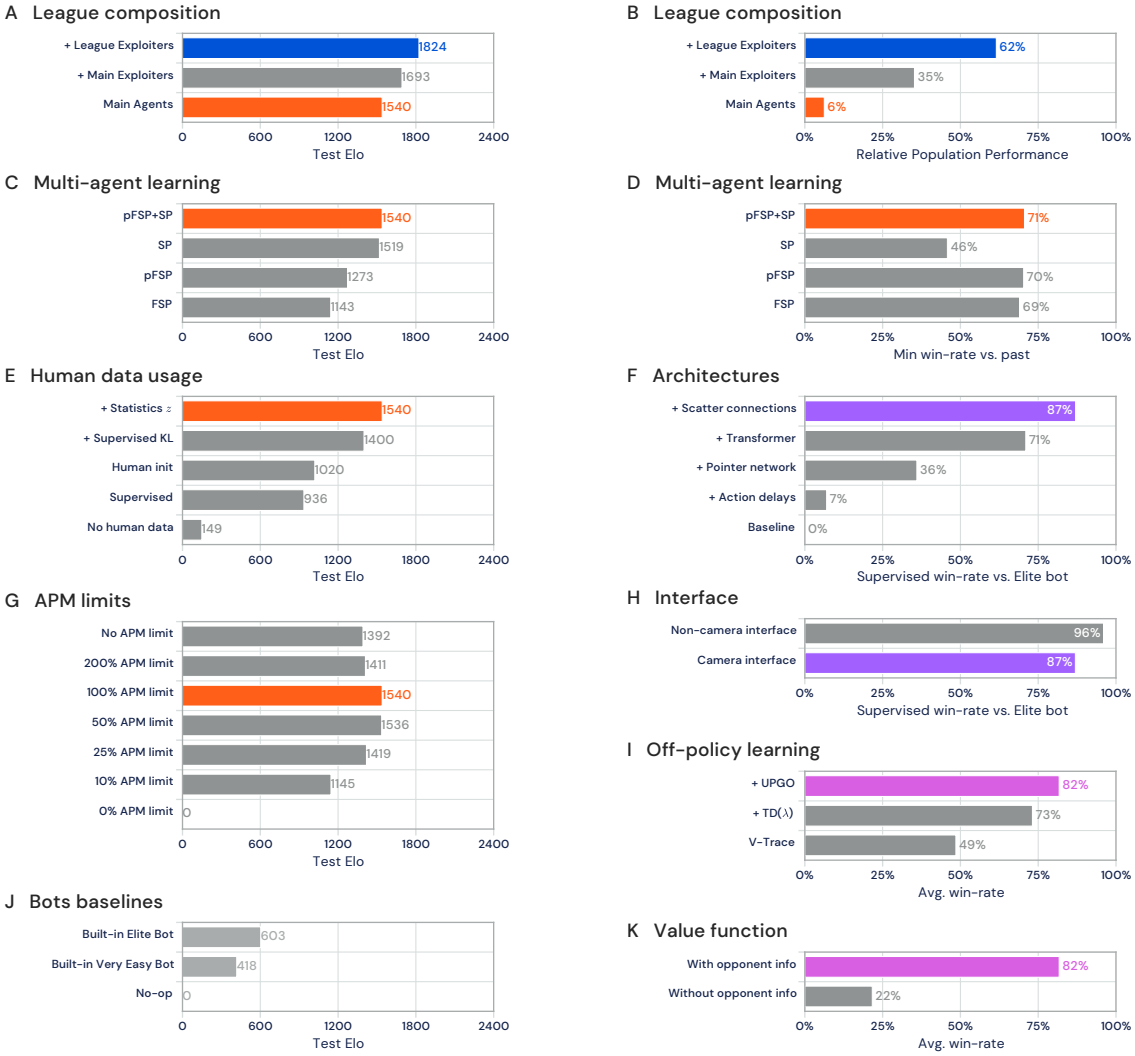
Training setup. A: AlphaStar observes the game through an overview map and list of units. To act, the agent outputs *what* action type to issue (e.g. build), *who* it is applied to, *where* it targets, and *when* the next action will be issued. Actions are sent to the game through a monitoring layer that limits action rate. AlphaStar contends with delays from network latency and processing time. **B:** AlphaStar is trained both via supervised learning and reinforcement learning. In supervised learning (bottom), the parameters are updated to optimise KL divergence between its output and human actions sampled from a collection of replays. In reinforcement learning (top), human data is used to sample the statistic z, and agent experience is collected to update the policy and value outputs via reinforcement learning (TD(λ), V-trace, UPGO) combined with a KL loss towards the supervised agent. **C:** Three pools of agents, each initialised by supervised learning, were subsequently trained with reinforcement learning. As they train, these agents intermittently add copies of themselves to the League. The main agents train against all past players, as well as themselves. The League exploiters train against all past players. The main exploiters train against the main agents. Main exploiters and League exploiters can be reset to the supervised agent when they add a player to the League.

Figure 2



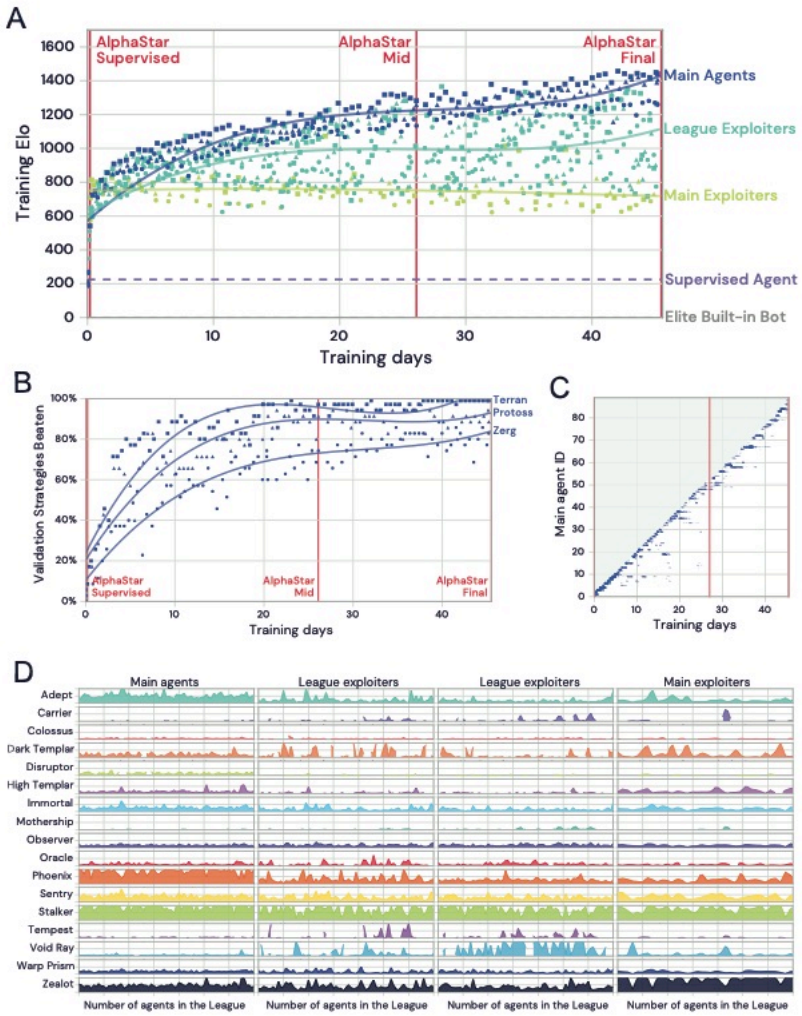
Results. A: On Battle.net, StarCraft II players are divided into 7 leagues, from Bronze to Grandmaster, based on their ratings (MMR). We played three variants of AlphaStar on Battle.net: (1) AlphaStar Supervised (2) AlphaStar Mid, and (3) AlphaStar Final. The supervised agent was rated in the top 16% of human players, the midpoint agent within the top 0.5%, and the final agent, on average, within the top 0.15%, achieving a Grandmaster level rating for all three races. **B:** MMR rating of AlphaStar Final per race (from top to bottom: Protoss, Terran, Zerg) versus opponents encountered on Battle.net (from left to right: all races combined, Protoss, Terran, Zerg). Note that per-race data is limited; AlphaStar won all Protoss versus Terran games. **C:** Distribution of effective actions per minute (EPM) as reported by StarCraft II for both AlphaStar Final (blue) and human players (red). Averages are marked as dashed lines.

Figure 3



Ablations for key components of AlphaStar. These experiments use a simplified setup: one map (Kairos Junction), one race match-up (Protoss versus Protoss), RL and League experiments limited to 10^{10} steps, only main agents, and a 50%-50% mix of self-play and PFSP, unless stated otherwise. More details in Methods. The first column shows Elo ratings against Ablation Test Agents. **A, B:** Comparing different League compositions using Elo of the Main agents (left) and Relative Population Performance of the whole Leagues (right), which measures exploitability. **C, D:** Comparing different multi-agent learning algorithms using Elo (left) and a proxy for forgetting: the minimum win-rate against all past versions, averaged over time (right). Naive self-play has high Elo, but is more forgetful. See Extended Data Fig. 5 for more in-depth comparison. **E:** Ablation study of the different mechanisms to use human data. **G:** APM limits relative to those used in AlphaStar. Reducing APM significantly reduces performance. Surprisingly, increasing APM also reduces performance, possibly because the agent spends more effort refining micro-tactics than learning diverse strategies. **F, H:** Comparison of architectures using the win-rate of supervised agents (trained in Protoss vs All) against the built-in Elite bot. **J:** Elo score of StarCraft II built-in bots. Ratings are anchored by a bot that never acts. **I, K:** Reinforcement learning ablations, measured by training a best response against Fixed Opponents to avoid multi-agent dynamics.

Figure 4



AlphaStar training progression. A: Training Elo of agents in the League during the 44 days of training. Each point represents a player, evaluated against the entire League and the Elite built-in bot (whose Elo is set to 0). **B:** Proportion of Validation Agents that beat the main agents in more than 80 out of 160 games. It increased steadily over time, which shows the robustness of League training to unseen strategies. **C:** The Nash distribution (mixture of the least exploitable players) of the players in the League, as training progressed. It puts the most weight on recent players, suggesting that latest strategies largely dominate the previous ones, without much forgetting or cycling. For example, player 40 was part of the Nash since its creation at day 20, until 5 days later, when it was completely dominated by newer agents. **D:** Average number of each unit built by the Protoss agents over the course of League training, normalised by the most common unit. Unlike the main agents, the exploiters rapidly explore different unit compositions. Worker units are removed for clarity.

Methods

Game and Interface

Game Environment: StarCraft is a real-time strategy game that takes place in a science fiction universe. Since StarCraft was released by *Blizzard Entertainment* in 1998, there has been a strong competitive community with tens of millions of dollars of prize money. The most common competitive setting of StarCraft II is 1v1, where each player chooses one of the three available races, Terran, Protoss, and Zerg[b] which all have distinct units and buildings, exhibit different mechanics, and necessitate different strategies when playing for and against. Players begin with a small base and a few worker units, which gather resources to build additional units and buildings, scout the opponent, and research new technologies. A player is defeated if they lose all buildings.

There is no universally accepted notion of fairness in real-time human-computer matches, so our match conditions, interface, camera view, action rate limits, and delays were developed in consultation with professional StarCraft II players and Blizzard employees. AlphaStar's play under these conditions was professional-player approved (Supplementary Data Professional Player Statement). At each agent step, the policy receives an observation o_t and issues an action a_t (detailed in Extended Data Tables 1 and 2) through the game interface. There can be several game time-steps (each 45 ms) per agent step.

Camera View: Humans play StarCraft through a screen that displays only part of the map along with a high-level view of the entire map, to e.g. avoid information overload. The agent interacts with the game through a similar camera-like interface, which naturally imposes an economy of attention, so that the agent chooses which area it fully sees and interacts with. The agent can move the camera as an action.

Opponent units outside the camera have certain information hidden, and the agent can only target within the camera for certain actions (e.g. building structures). AlphaStar can target locations more accurately than humans outside the camera, although less accurately within it because target locations (selected on a 256x256 grid) are treated the same inside and outside the camera. Agents can also select sets of units anywhere, which humans can do less flexibly using control groups. In practice, the agent does not seem to exploit these extra capabilities (Supplementary Data Professional Player Statement), because of the human prior. Ablation Fig. 3H shows that using this camera view reduces performance.

APM Limits: Humans are physically limited in the number of actions per minute (APM) they can execute. Our agent has a monitoring layer which enforces APM limitations. This introduces an action economy that requires prioritising actions. Agents are limited to executing at most 22 non-duplicate actions per five second window. Converting between actions and the APM measured by the game is non-trivial, and agent actions are hard to compare with human actions (computers can precisely execute different actions from step to step). See Fig. 2C and Extended Data Fig. 1 for APM details.

Delays: Humans are limited in how quickly they react to new information; AlphaStar has two sources of delays. First, in real-time evaluation (not training), AlphaStar has a delay of about 110 ms between when a frame is observed and when an action is executed due to latency,

[b] There is also a Random race, where the game selects the race at random.

observation processing, and inference. Second, because agents decide ahead of time when to observe next (on average 370 ms, but possibly multiple seconds), they may react late to unexpected situations. The distribution of these delays is shown in Extended Data Fig. 2.

Related Work

Games have been a focus of artificial intelligence research for decades as a stepping stone towards more general applications. Classic board games such as chess[24] and Go[25] have been mastered using general-purpose reinforcement learning and planning algorithms[18]. Reinforcement learning methods have achieved significant successes in video games such as Atari[26], Mario[27], Quake III Arena Capture the Flag[28], and Dota 2[29].

Real-time strategy (RTS) games are recognised for their game-theoretic and domain complexities[30]. Many sub-problems of RTS games, e.g. micro-management, base economy, or build order optimisation, have been studied in depth in the literature[7,31–34], often in small-scale environments[35,36]. For the combined challenge, the StarCraft domain has emerged by consensus as a research focus[1,7]. StarCraft: Brood War has an active competitive AI research community[37], and most bots combine rule-based heuristics with other AI techniques such as search[4,38], data driven build order selection[39], and simulation[40]. Reinforcement learning has also been studied to control units in the game[7,33,41–43], and imitation learning has been proposed to learn unit and building compositions[44]. Most recently, deep learning has been used to predict future game states[45]. StarCraft II similarly has an active bot community[3] since the release of a public API[7]. No StarCraft bots have defeated professional players, or even high-level casual players[46], and the most successful bots used superhuman capabilities, such as executing tens of thousands of actions per minute or viewing the entire map at once. These capabilities make comparing against humans hard, and invalidate certain strategies. Some of the most recent approaches use reinforcement learning to play the full game, with hand-crafted, high-level actions[47], or rule-based systems with machine learning incrementally replacing components[42]. In contrast, AlphaStar proposes a model free, end-to-end learning approach to playing StarCraft II which sidesteps the difficulties of search-based methods due to imperfect models, and is applicable to any domain that shares some of the challenges present in StarCraft.

Dota 2 is a modern competitive team game that shares some complexities of RTS games such as StarCraft (like imperfect information and large time horizons). Recently, OpenAI Five defeated a team of professional Dota 2 players and 99.4% of online players[29]. The hero units of OpenAI Five are controlled by a team of agents, trained together with a scaled up version of PPO[48], based on handcrafted rewards. However, unlike AlphaStar, some game rules were simplified, players were restricted to a subset of heroes, agents used hard-coded sub-systems for certain aspects of the game, and agents did not limit their perception to a camera view.

AlphaStar relies on imitation learning combined with reinforcement learning, which has been used several times in the past. Similarly to the training pipeline of AlphaStar, the original AlphaGo initialised a policy network by supervised learning from human games, which was then used as a prior in Monte-Carlo tree search[25]. Similar to our statistic z, other work attempted to train reward functions from human preferences and use them to guide reinforcement learning[49,50] or learned goals from human intervention[51].

Related to the League, recent progress in multi-agent research led to agents performing at human-level in the Capture the Flag team mode of Quake III Arena[28]. These results were obtained using population-based training of several agents competing with each other, which used pseudo-reward evolution to deal with the hard credit assignment problem. Similarly, the Policy Space Response Oracle framework[52] is related to League training, although League training specifies unique targets for approximate best responses (i.e. PFSP and exploiters).

Architecture

The policy of AlphaStar is a function $\pi_\theta(a_t|s_t, z)$, mapping all previous observations and actions $s_t = o_{1:t}, a_{1:t-1}$ (defined in Extended Data Tables 1 and 2) and z (representing strategy statistics) to a probability distribution over actions a_t for the current step. π_θ is implemented as a deep neural network with the following structure.

The observations o_t are encoded into vector representations, combined, and processed by a deep LSTM[9] which maintains memory between steps. The action arguments a_t are sampled auto-regressively[10], conditioned on the outputs of the LSTM and the observation encoders. There is a value function for each of the possible rewards (see Reinforcement Learning).

Architecture components were chosen and tuned with respect to their performance in supervised learning, and include many recent advances in deep learning architectures[7,8,12,53,54]. A high-level overview of the agent architecture is given in Extended Data Fig. 3, with more detailed descriptions in Supplementary Data Architecture. AlphaStar has 139 million weights, but only 55 million weights are required during inference. Ablation Fig. 3F compares the impact of scatter connections, transformer, and pointer network.

Supervised Learning

Each agent is initially trained through supervised learning on replays to imitate human actions. Supervised learning is used both to initialise the agent and to maintain diverse exploration[55]. Because of this, the primary goal is to produce a diverse policy that captures StarCraft's complexities.

We use a dataset of 971,000 replays[c] played on StarCraft II versions 4.8.2 to 4.8.6 by players with MMR (Blizzard's metric similar to Elo) greater than 3500, i.e. from the top 22% of players. The observations and actions are returned by the game's raw interface (described in Extended Data Tables 1 and 2). We train one policy for each race, with the same architecture as the one used during reinforcement learning.

From each replay, we extract a statistic *z* encoding each player's *build order*, defined as the first 20 constructed buildings and units, and *cumulative statistics*, defined as the units, buildings, effects, and upgrades that were present during a game. We condition the policy on *z* in both supervised and reinforcement learning, and in supervised learning we set *z* to zero 10% of the time.

To train the policy, at each step we input the current observations and output a probability distribution over each action argument (Extended Data Table 2). For these arguments, we compute the KL divergence between human actions and the policy's outputs, and apply

[c] Instructions for downloading replays can be found at https://github.com/Blizzard/s2client-proto

updates using the Adam optimiser[56]. We also apply L_2 regularisation[57]. The pseudocode of the supervised training algorithm can be found in Supplementary Data Pseudocode.

We further fine-tune the policy using only winning replays with MMR above 6200 (16,000 games). Fine-tuning improved the win-rate against the built-in Elite bot from 87% to 96% in Protoss versus Protoss. The fine-tuned supervised agents were rated at 3947, 3607, 3544 MMR for Terran, Protoss, and Zerg, respectively. They are capable of building all units in the game, and are qualitatively diverse from game to game (Extended Data Fig. 4).

Reinforcement Learning

We apply reinforcement learning to improve the performance of AlphaStar based on agent versus agent games. We use the match outcome (-1 on a loss, 0 on a draw and $+1$ on a win) as the terminal reward r_T, without a discount to accurately reflect the true goal of winning games. Following the actor-critic paradigm[14], a value function $V_\theta(s_t, z)$ is trained to predict r_t, and used to update the policy $\pi_\theta(a_t|s_t, z)$.

StarCraft poses several challenges when viewed as a reinforcement learning problem: exploration is difficult due to domain complexity and reward sparsity; policies need to be capable of executing diverse strategies throughout training; and off-policy learning is difficult due to large time horizons and the complex action space.

Exploration and diversity

We use human data to aid in exploration and to preserve strategic diversity throughout training. First, we initialise the policy parameters to the supervised policy and continually minimise the KL divergence between the supervised and current policy[58,59]. Second, we train the main agents with pseudo-rewards to follow a strategy statistic z, which we randomly sample from human data. These pseudo-rewards measure the edit distance between sampled and executed *build orders*, and the Hamming distance between sampled and executed *cumulative statistics* (see Supplementary Data Detailed Architecture). Each type of pseudo-reward is active (i.e. non-zero) with probability 25%, and separate value functions and losses are computed for each pseudo-reward. We found our use of human data to be critical in achieving good performance with reinforcement learning (Fig. 3E).

Value and policy updates

New trajectories are generated by actors. Asynchronously, model parameters are updated by learners, using a replay buffer that stores trajectories. Because of this, AlphaStar is subject to off-policy data, which potentially requires off-policy corrections. We found that off-policy correction methods like V-trace[14] can be inefficient in large, structured action spaces like the one we used for StarCraft, because distinct actions can result in similar (or even identical) behaviour. We address this by using a hybrid approach. The policy is updated using V-trace and the value estimates are updated using TD(λ)[5], which does not apply off-policy corrections (ablation in Fig. 3I). To decrease variance of the value estimates, we also use the opponent's observations as input to the value functions (ablation in Fig. 3K). Note that these are only used during training, as value functions are unnecessary during evaluation.

For the policy $\pi_\theta(a_t|s_t,z)$, using V-trace off-policy corrections improved learning stability. To mitigate early trace cutting due to the large action space, we assume independence between the action type, delay, and all other arguments, and so update them separately.

In addition to the V-trace policy update, we introduce an *upgoing* policy update (UPGO), which updates the policy parameters in the direction of

$$\rho_t \left(G_t^{\mathrm{U}} - V_\theta(s_t,z)\right) \nabla_\theta \log \pi_\theta(a_t|s_t,z)\,,$$

where

$$G_t^{\mathrm{U}} = \begin{cases} r_t + G_{t+1}^{\mathrm{U}} & \text{if } Q(s_{t+1},a_{t+1},z) \geq V_\theta(s_{t+1},z) \\ r_t + V_\theta(s_{t+1},z) & \text{otherwise} \end{cases}$$

is an upgoing return, $Q(s_t,a_t,z)$ is an action-value estimate, $\rho_t = \min\left(\frac{\pi_\theta(a_t|s_t,z)}{\pi_{\theta'}(a_t|s_t,z)},1\right)$ is a clipped importance ratio, and $\pi_{\theta'}$ is the policy that generated the trajectory in the actor. Similar to self-imitation learning[17], the idea is to update the policy from partial trajectories with better-than-expected returns by bootstrapping when the behaviour policy takes a worse-than-average action (ablation in Fig. 3I). Due to the difficulty of approximating $Q(s_t,a_t,z)$ over the large action space of StarCraft, we estimate action-values with a one-step target, $Q(s_t,a_t,z) = r_t + V_\theta(s_{t+1},z)$.

The overall loss is a weighted sum of the policy and value function losses described above, corresponding to the win-loss reward r_t as well as pseudo-rewards based on human data, the KL divergence loss with respect to the supervised policy, and the standard entropy regularisation loss[13]. We optimise the overall loss with Adam[56]. The pseudocode of the reinforcement learning algorithm can be found in Supplementary Data Pseudocode.

Multi-agent Learning

League training is a multi-agent reinforcement learning algorithm that is designed both to address the cycles commonly encountered during self-play training, and to integrate a diverse range of strategies. During training, we populate the League by regularly saving the parameters from our agents (that are being trained by the RL algorithm) as new players (which have fixed, frozen parameters). We also continuously re-evaluate the internal payoff estimation, giving agents up-to-date information about their performance against all players in the League (see Evaluators in the Extended Data Fig. 6).

Prioritised Fictitious Self-Play

Our Self-Play (SP) algorithm plays games between the latest agents for all three races. This approach may chase cycles in strategy space and does not work well in isolation (Fig. 3D). Fictitious Self-Play (FSP)[20–22] avoids cycles by playing against all previous players in the League. However, many games are wasted against players that are defeated in almost 100% of games. Consequently, we introduce Prioritised Fictitious Self-Play (PFSP). Instead of uniformly sampling opponents in the League, we use a matchmaking mechanism to provide a good learning signal. Given a learning agent A, we sample the frozen opponent B from a candidate set $\mathcal{C}$ with probability

$$\frac{f(\mathbb{P}[A \text{ beats } B])}{\sum_{C \in \mathcal{C}} f(\mathbb{P}[A \text{ beats } C])},$$

where $f: [0,1] \to [0, \infty)$ is some weighting function.

Choosing $f_{\text{hard}}(x) = (1-x)^p$ makes PFSP focus on the hardest players, where $p \in \mathbb{R}_+$ controls how entropic the resulting distribution is. Since $f_{\text{hard}}(1) = 0$, no games are played against opponents that the agent already beats. By focusing on the hardest players, the agent must beat everyone in the League rather than maximising average performance, which is even more important in highly non-transitive games like StarCraft, where the pursuit of the mean win-rate might lead to policies that are easy to exploit. This scheme is used as the default weighting of PFSP. Consequently, on the theoretical side, one can view f_{hard} as a form of smooth approximation of max-min optimisation, as opposed to max-avg that FSP imposes. In particular, this helps with integrating information from exploits, as these are strong but rare counter strategies, and a uniform mixture would be able to just ignore them (Extended Data Fig. 5).

Only playing against the hardest opponents can waste games against much stronger opponents, so PFSP also uses an alternative curriculum, $f_{\text{var}}(x) = x(1-x)$, where the agent preferentially plays against opponents around its own level. We use this curriculum for main exploiters and struggling main agents.

Populating the League

During training we used three agent types that differ only in the distribution of opponents they train against, when they are snapshotted to create a new player, and the probability of resetting to the supervised parameters.

Main Agents are trained with a proportion of 35% SP, 50% PFSP against all past players in the League, and an additional 15% of PFSP matches against forgotten main players the agent can no longer beat and past main exploiters. If there are no forgotten players or strong exploiters, the 15% is used for self-play instead. Every $2 \cdot 10^9$ steps, a copy of the agent is added as a new player to the League. Main agents never reset.

League Exploiters are trained using PFSP and their frozen copies are added to the League when they defeat all players in the League in more than 70% of games, or after a timeout of $2 \cdot 10^9$ steps. At this point there is a 25% probability that the agent is reset to the supervised parameters. The intuition is that League Exploiters identify global blind spots in the League (strategies that no player in the League can beat, but that are not necessarily robust themselves).

Main Exploiters play against main agents. Half of the time, and if the current probability of winning is lower than 20%, exploiters use PFSP with f_{var} weighting over players created by the main agents. This forms a curriculum which facilitates learning. Otherwise there is enough learning signal and it plays against the current main agents. These agents are added to the League whenever all three main agents are defeated in more than 70% of games, or after a timeout of $4 \cdot 10^9$steps. They are then reset to the supervised parameters. Main Exploiters identify weaknesses of main agents, and consequently make them more robust.

For more details refer to the pseudocode in Supplementary Materials.

Infrastructure

In order to train the League, we run a large number of StarCraft II matches in parallel and update the parameters of the agents based on data from those games. To manage this, we developed a highly scalable training setup with different types of distributed workers.

For every training agent in the League, we run 16,000 concurrent StarCraft II matches and 16 actor tasks (each using a TPU v3 device with 8 TPU cores[23]) to perform inference. The game instances progress asynchronously on preemptible CPUs (roughly equivalent to 150 processors with 28 physical cores each), but requests for agent steps are batched together dynamically to make efficient use of the TPU. Utilising TPUs for batched inference provides large efficiency gains over prior work[14,28].

Actors send sequences of observations, actions, and rewards over the network to a central 128-core TPU learner worker, which updates the parameters of the training agent. The received data is buffered in memory and replayed twice. The learner worker performs large-batch synchronous updates. Each TPU core processes a mini-batch of 4 sequences, for a total batch size of 512. The learner processes about 50,000 agent steps per second. The actors update their copy of the parameters from the learner every 10 seconds.

We instantiate 12 separate copies of this actor-learner setup: one main agent, one main exploiter and two League exploiter agents for each StarCraft race. One central coordinator maintains an estimate of the payoff matrix, samples new matches on request, and resets main and league exploiters. Additional evaluator workers (running on CPU) are used to supplement the payoff estimates. See Extended Data Fig. 6 for an overview of the training setup.

Evaluation

AlphaStar Battle.net Evaluation

AlphaStar agents were evaluated against humans on Battle.net, Blizzard's online matchmaking system based on MMR ratings, on StarCraft II balance patch 4.9.3. AlphaStar Final was rated at Grandmaster level, above 99.8% of human players who were active enough in the past months to be placed into a league in the European server (about 90,000 players).

AlphaStar only played opponents who opted to participate in the experiment (the majority of players opted in)[60], used an anonymous account name, and played on four maps: Cyber Forest, Kairos Junction, King's Cove, and New Repugnancy[d]. Humans also must select at least four maps and frequently play under anonymous account names. Each agent ran on a single high-end consumer GPU. We evaluated at three points during training: supervised, midpoint, and final.

For the supervised and midpoint evaluation, each agent began with a fresh, unranked account. Their MMR was updated on Battle.net as for humans. The supervised and midpoint

[d] Blizzard updated the map pool a few weeks before testing. Instead of retraining AlphaStar, we simply played on the four common maps that were kept in the pool of seven available maps.

evaluation played 30 and 60 games respectively. The midpoint evaluation was halted while still increasing because the anonymity constraint was compromised after 50 games.

For the final Battle.net evaluation, we used several accounts to parallelise the games and help avoid identification. The MMRs of our accounts were seeded randomly from the distribution of combined, estimated, midpoint MMRs. Consequently, we no longer used the iterative MMR estimation provided in Battle.net, and instead used the underlying probabilistic model provided by Blizzard: given our rating r with uncertainty u, and opponent rating r_i with uncertainty $u_i \in [0.1, 1.0]$, the probability of the outcome $o_i \in \{-1, 1\}$ is

$$\mathbb{P}[o_i = 1|r,u,r_i,u_i] = 1 - \mathbb{P}[o_i = -1|r,u,r_i,u_i] = \Phi\left(\frac{r - r_i}{400\sqrt{2 + u^2 + u_i^2}}\right) \approx \Phi\left(\frac{r - r_i}{568}\right)$$

where ϕ is the CDF of a standard Gaussian distribution, and where we used Battle.net's minimum uncertainties $u = u_i = 0.1$.

Under i.i.d. assumptions of match results and a uniform prior over MMRs, we can compute our rating as

$$\text{argmax}_{r\in\mathbb{N}}\mathbb{P}[r|\text{results}] = \text{argmax}_{r\in\mathbb{N}}\mathbb{P}[\text{results}|r]U(r) = \text{argmax}_{r\in\mathbb{N}}\prod_{i=1}^{N}\mathbb{P}[o_i|r,r_i].$$

We validated our MMR computation on the 200 most recent matches of Dario "TLO" Wünsch, a professional StarCraft II player, and obtained an MMR estimate of 6334, while the average MMR reported by Battle.net was 6336.

StarCraft Demonstration Evaluation

In December 2018, we played two 5-game series against StarCraft II professional players Grzegorz "MaNa" Komincz and Dario "TLO" Wünsch, though TLO did not play the same StarCraft II race that he plays professionally. These games took place with a different, preliminary version of AlphaStar[61]. In particular, the agent did not have a limited camera, was less restricted in how often it could act, and played for and against a single StarCraft II race on a single map. AlphaStar won all 10 games in both 5 game series, though an early camera prototype lost a follow-up game against MaNa.

Analysis

Agent sets

Validation Agents: We validated League robustness against a set of 17 strategies trained using only main agents and no exploiters, and fixing *z* to a hand-curated set of interesting strategies (e.g. a cannon rush or early flying units).

Ablation Test Agents: Ablation test agents include the validation agents, and the first (i.e. weaker) 20 main and 20 League exploiter Protoss agents created by full League Training.

Fixed Opponents: To evaluate our RL algorithms, we computed the best response against a uniform mixed strategy composed of the first 10 League exploiter Protoss agents created by League Training.

Metrics used in Figures

Elo rating: To compute internal Elo ratings of the League, we added the built-in bots, and used it to estimate Elo with the following model:

$$\mathbb{P}[\mathrm{r}_1 \text{ beats } \mathrm{r}_2] = \frac{1}{1 + e^{-(\mathrm{r}_1 - \mathrm{r}_2)/400}} \approx \Phi\left(\frac{\mathrm{r}_1 - \mathrm{r}_2}{400}\right)$$

where r_1 and r_2 are the Elo ratings of both players. Since the Elo rating has no intrinsic absolute scale, we ground it by setting the rating of the built-in Elite bot to 0.

Relative Population Performance (RPP) is the expected outcome of the meta-game between two populations after they reach the Nash equilibrium[19]. Given a payoff matrix between all agents in the Leagues A and B of sizes N and M respectively $P_{AB} \in [0,1]^{N \times M}$:

$$\mathrm{RPP}(P_{AB}) = \mathrm{Nash}(P_{AB})^T \, P_{AB} \, \mathrm{Nash}(P_{BA})$$

where $\mathrm{Nash}(X) \in [0,1]^K$ is a vector of probabilities assigned to playing each agent, in League X of size K, in the Nash equilibrium. High RPP means that League A consists of agents that can form a mixed strategy that can exploit agents from B, while not being too exploitable by any mixed strategy from B.

AlphaStar Generality

To address the complexity and game-theoretic challenges of StarCraft, AlphaStar uses a combination of new and existing general-purpose techniques for neural network architectures, imitation learning, reinforcement learning, and multi-agent learning. These techniques and their combination are widely applicable.

The neural network architecture components, including the new scatter connections, are all generally applicable to any domain whose observations comprises a combination of images, lists, and sets, all of which are present in StarCraft.

AlphaStar's action space is defined as a set of functions with typed arguments. Any domain which defines a similar API can be tackled with the same decomposition of complex, structured action spaces, whose joint probability is decomposed via the chain rule (akin to e.g. language modelling[10] or theorem proving).

Imitation learning in AlphaStar requires a large amount of human demonstrations to be effective, and thus is only applicable to those domains which provide such a set of demonstrations. Using a latent variable z to induce exploration is not specific to StarCraft, but the particular choice of statistics required domain knowledge. In particular, we chose z to encode openings and units in StarCraft. Pseudo-rewards were based on appropriate distance metrics for these statistics, such as edit distance or Hamming distance.

AlphaStar's underlying reinforcement learning algorithm can be applied to any RL environment. Using an opponent's observations for a lower-variance baseline and new components, such as hybrid off-policy learning, UPGO, and distillation towards an imitation policy, are also widely applicable.

Lastly, we propose a new multi-agent training regime with different kinds of exploiters whose purpose is to strengthen the main agents. Together with prioritised FSP, these are all general-purpose techniques applicable to any multiplayer domain.

Code and Data Availability Statement

Professional Player Statement

The following quote describes our interface and limitations from StarCraft II professional player Dario "TLO" Wünsch (who is part of the team and an author of this paper).

The limitations that have been put in place for AlphaStar now mean that it feels very different from the initial show match in January. While AlphaStar has excellent and precise control it doesn't feel superhuman - certainly not on a level that a human couldn't theoretically achieve. It is better in some aspects than humans and then also worse in others, but of course there are going to be unavoidable differences between AlphaStar and human players.

I've had the pleasure of providing consultation to the AlphaStar team to help ensure that DeepMind's system does not have any unfair advantages over human players. Overall, it feels very fair, like it is playing a `real' game of StarCraft and doesn't completely throw the balance off by having unrealistic capabilities. Now that it has limited camera view, when I multi-task it doesn't always catch everything at the same time, so that aspect also feels very fair and more human-like.

Replay Data

All the games that AlphaStar played online can be found in the file *replays.zip* (Supplementary Data Replays), and the raw data from the Battle.net experiment can be found in *bnet.json* (Supplementary Data Battle.net).

Code

The StarCraft II environment was open sourced in 2017 by Blizzard and DeepMind[7]. All the human replays used for imitation learning can be found at https://github.com/Blizzard/s2client-proto. The pseudocode for the supervised learning, reinforcement learning, and multi-agent learning components of AlphaStar can be found in the file *pseudocode.zip* (Supplementary Data Pseudocode). All the neural architecture details and hyper-parameters can be found in the file *detailed-architecture.txt* (Supplementary Data Architecture).

Method References

24. Campbell, M., Hoane, A. & Hsu, F. Deep Blue. *Artif. Intell.* **134**, 57–83 (2002).

25. Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016).

26. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529–533 (2015).

27. Pathak, D., Agrawal, P., Efros, A. A. & Darrell, T. Curiosity-Driven Exploration by Self-Supervised Prediction. in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* 16–17 (2017).

28. Jaderberg, M. *et al.* Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science* **364**, 859–865 (2019).

29. OpenAI. OpenAI Five. (2018). Available at: https://blog.openai.com/openai-five/.

30. Buro, M. Real-Time Strategy Games: A New AI Research Challenge. in *International Joint Conference on Artificial Intelligence* 1534–1535 (2003).

31. Samvelyan, M. *et al.* The StarCraft Multi-Agent Challenge. in *International Conference on Autonomous Agents and MultiAgent Systems* 2186–2188 (2019).

32. Zambaldi, V. *et al.* Relational Deep Reinforcement Learning. in *International Conference on Learning Representations* (2018).

33. Usunier, N., Synnaeve, G., Lin, Z. & Chintala, S. Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks. in *International Conference on Learning Representations* (2017).

34. Weber, B. G. & Mateas, M. Case-based Reasoning for Build Order in Real-Time Strategy Games. in *Artificial Intelligence and Interactive Digital Entertainment Conference* (2009).

35. Buro, M. ORTS: A Hack-Free RTS Game Environment. in *International Conference on Computers and Games* 280–291 (2002).

36. Churchill, D. SparCraft: Open source StarCraft combat simulation. (2013). Available at: https://code.google.com/archive/p/sparcraft/.

37. Weber, B. G. AIIDE 2010 StarCraft Competition. in *Artificial Intelligence and Interactive Digital Entertainment Conference* (2010).

38. Uriarte, A. & Ontañón, S. Improving Monte Carlo Tree Search Policies in StarCraft via Probabilistic Models Learned from Replay Data. in *Artificial Intelligence and Interactive Digital Entertainment Conference* (2016).

39. Hsieh, J.-L. & Sun, C.-T. Building a Player Strategy Model by Analyzing Replays of Real-Time Strategy Games. in *IEEE International Joint Conference on Neural Networks* 3106–3111 (2008).

40. Synnaeve, G. & Bessiere, P. A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. in *Artificial Intelligence and Interactive Digital Entertainment Conference* (2011).

41. Shao, K., Zhu, Y. & Zhao, D. Starcraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning. *IEEE Trans. Emerg. Top. Comput. Intell.* **3**, 73–84 (2018).

42. Facebook CherryPi. Available at: https://torchcraft.github.io/TorchCraftAI/.

43. Berkeley Overmind. Available at: https://http//overmind.cs.berkeley.edu/.

44. Justesen, N. & Risi, S. Learning Macromanagement in StarCraft from Replays using Deep Learning. in *IEEE Conference on Computational Intelligence and Games (CIG)* 162–169 (2017).

45. Synnaeve, G. *et al.* Forward Modeling for Partial Observation Strategy Games - A StarCraft Defogger. in *Advances in Neural Information Processing Systems* 10738–

10748 (2018).

46. Farooq, S. S., Oh, I.-S., Kim, M.-J. & Kim, K. J. StarCraft AI Competition Report. *AI Mag.* **37**, 102–107 (2016).

47. Sun, P. *et al.* TStarBots: Defeating the Cheating Level Builin AI in StarCraft II in the Full Game. *arXiv Prepr. arXiv1809.07193* (2018).

48. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. Proximal Policy Optimization Algorithms. *arXiv Prepr. arXiv1707.06347* (2017).

49. Ibarz, B. *et al.* Reward learning from human preferences and demonstrations in Atari. in *Advances in Neural Information Processing Systems* 8011–8023 (2018).

50. Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W. & Abbeel, P. Overcoming Exploration in Reinforcement Learning with Demonstrations. in *IEEE International Conference on Robotics and Automation* 6292–6299 (2018).

51. Christiano, P. F. *et al.* Deep reinforcement learning from human preferences. in *Advances in Neural Information Processing Systems* 4299–4307 (2017).

52. Lanctot, M. *et al.* A Unified Game-Theoretic Approach to Multiagent Reinforcement Learning. in *Advances in Neural Information Processing Systems* 4190–4203 (2017).

53. Perez, E., Strub, F., De Vries, H., Dumoulin, V. & Courville, A. FiLM: Visual Reasoning with a General Conditioning Layer. in *AAAI Conference on Artificial Intelligence* (2018).

54. He, K., Zhang, X., Ren, S. & Sun, J. Deep Residual Learning for Image Recognition. in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* 770–778 (2016).

55. Hinton, G., Vinyals, O. & Dean, J. Distilling the Knowledge in a Neural Network. *arXiv Prepr. arXiv1503.02531* (2015).

56. Kingma, D. P. & Ba, J. Adam: A Method for Stochastic Optimization. *arXiv Prepr. arXiv1412.6980* (2014).

57. Bishop, C. M. *Pattern Recognition and Machine Learning*. (Springer, 2006).

58. Rusu, A. A. *et al.* Policy Distillation. in *International Conference on Learning Representations* (2016).

59. Parisotto, E., Ba, J. & Salakhutdinov, R. Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning. in *International Conference on Learning Representations* (2016).

60. DeepMind Research on Ladder. Available at: https://starcraft2.com/en-us/news/22933138.

61. Vinyals, O. *et al.* AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. (2019). Available at: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii.

Acknowledgements

We would like to thank Blizzard for creating StarCraft and acknowledge them for their continued support of the research environment, and enabling AlphaStar to participate in Battle.net. In particular, we would like to thank Austin Hudelson, Chris Lee, Kevin Calderone, and Tim Morten. We would also like to thank StarCraft II professional players Grzegorz “MaNa” Komincz and Diego “Kelazhur” Schwimer for their StarCraft expertise and advice.

We also wish to thank Adam Cain, Ali Razavi, Daniel Toyama, David Balduzzi, Doug Fritz, Eser Aygün, Florian Strub, Georg Ostrovski, Guillaume Alain, Haoran Tang, Jaume Sanchez, Jonathan Fildes, Julian Schrittwieser, Justin Novosad, Karen Simonyan, Karol Kurach, Philippe Hamel, Ricardo Barreira, Scott Reed, Sergey Bartunov, Shibl Mourad, Steve Gaffney, Thomas Hubert, the team that created PySC2 and the whole DeepMind Team, with special thanks to the research platform team, comms and events teams, for their support, ideas, and encouragement.

Author Contributions

O.V., I.B., W.M.C., M.M., A.D., J.C., D.C., R.P., T.E., P.G., J.O., D.Ho., M.K., I.D., A.H., L.S., T.C., J.A., C.A., and D.S. contributed equally.

O.V., I.B., W.M.C., M.M., A.D., J.C., D.C., R.P., T.E., P.G., J.O., D.Ho., M.K., I.D., A.H., L.S., T.C., J.A., C.A., R.L., M.J., V.D., Y.S., S.V., D.B., T.Pa., C.G., Z.W., T.Pf., T.Po., and D.S. designed and built AlphaStar with advice from T.S., and T.L.

J.M., and R.R. contributed to software engineering.

D.W. and D.Y. provided expertise in the StarCraft II domain.

K.K., D.Ha., K.M., O.S., and C.A. managed the project

D.S., W.M.C., O.V., J.O., I.B., and D.C. wrote the paper with contributions from M.M., J.C., D.Ho., L.S., R.L., T.C., T.S., and T.L.

O.V. and D.S. led the team.

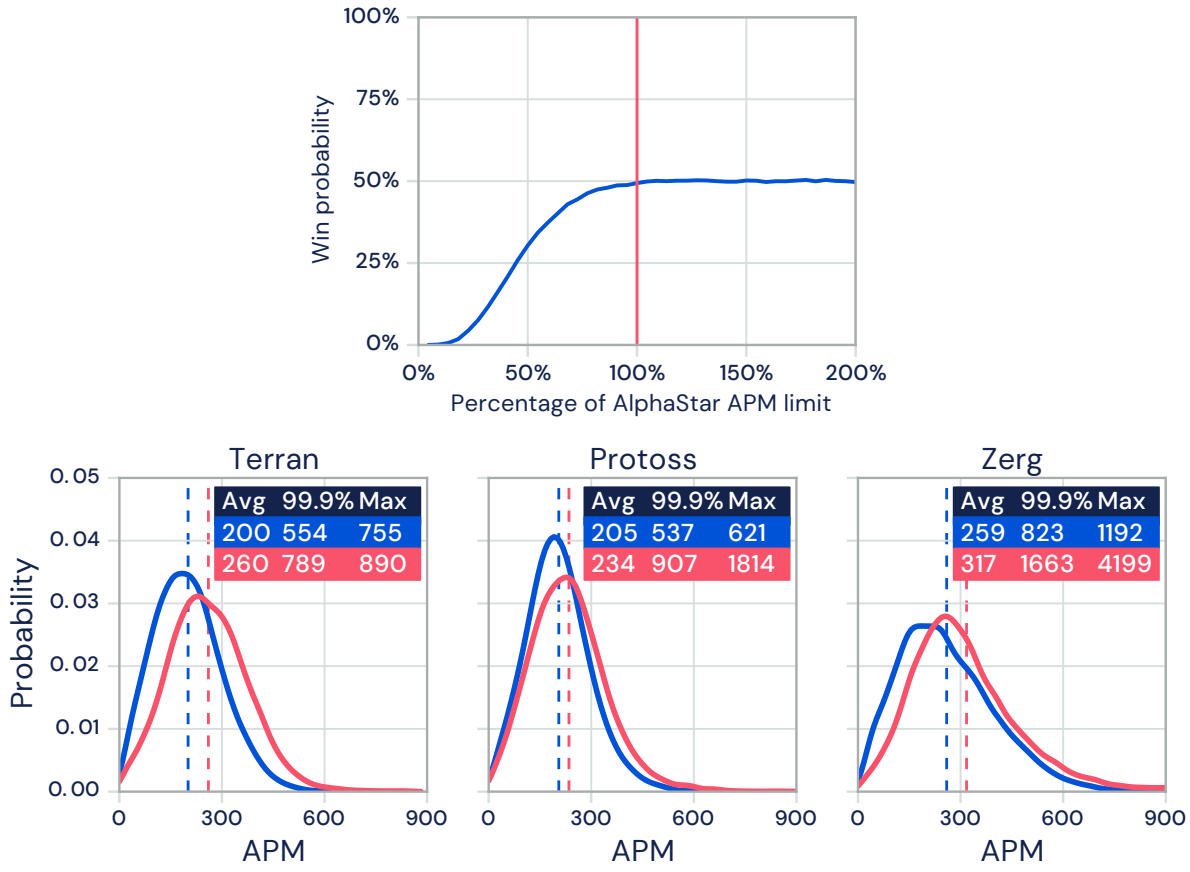
Competing Interests

M.J., W.M.C., O.V., and D.S. have filed provisional patent application 62/796,567 about the contents of this manuscript. The remaining authors declare no competing financial or conflicts of interest.

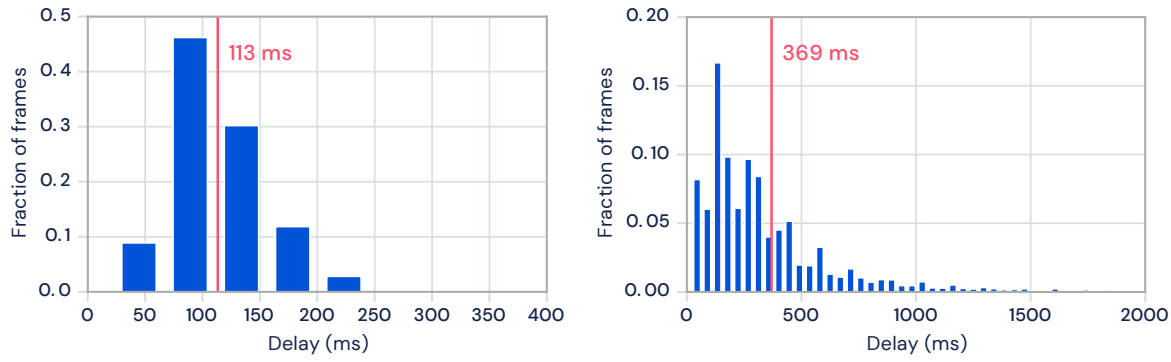
Materials and correspondence

Correspondence and material requests should be sent to Oriol Vinyals: vinyals@google.com.

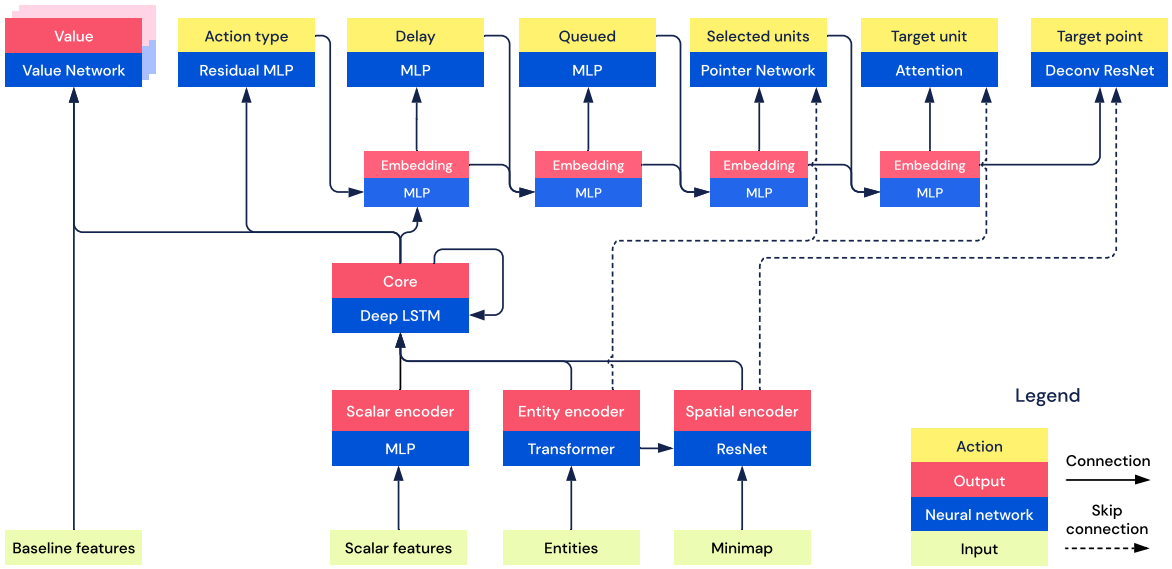
Extended Data



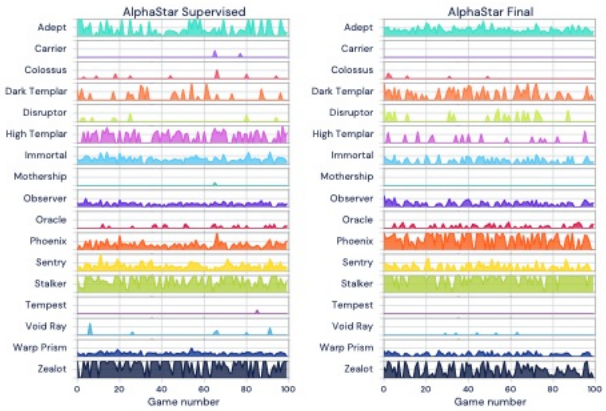
Extended Data Fig. 1 | APM limits. (**Top**) Win probability of AlphaStar Supervised against itself, when applying various agent action rate limits. Our limit does not affect supervised performance and is acceptable when compared to humans. (**Bottom**) Distributions of APMs of AlphaStar Final (blue) and humans (red) during games on *Battle.net*. Averages are marked as dashed lines.



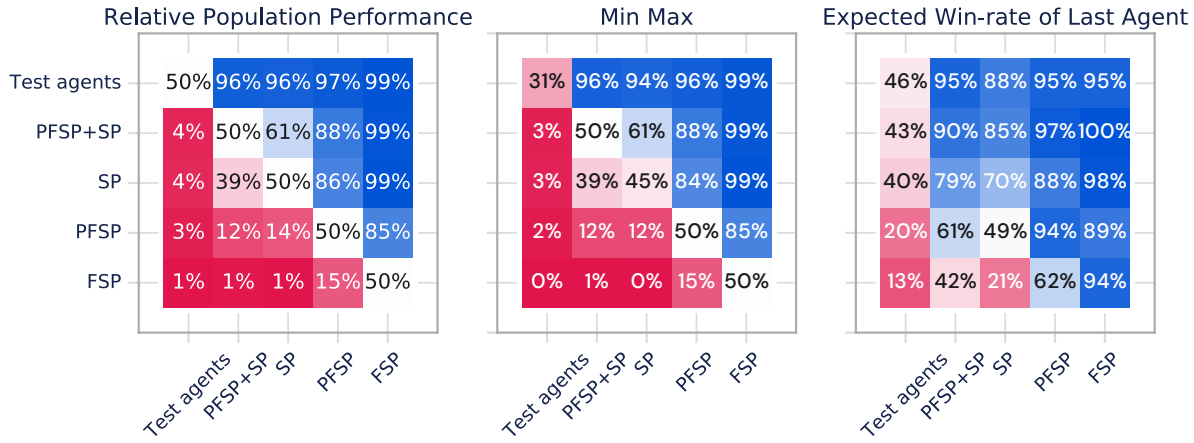
Extended Data Fig. 2 | Delays. (**Left**) Distribution of delays between when the game generates an observation and when the game executes the corresponding agent action. (**Right**) Distribution of how long agents request to wait without observing between observations.



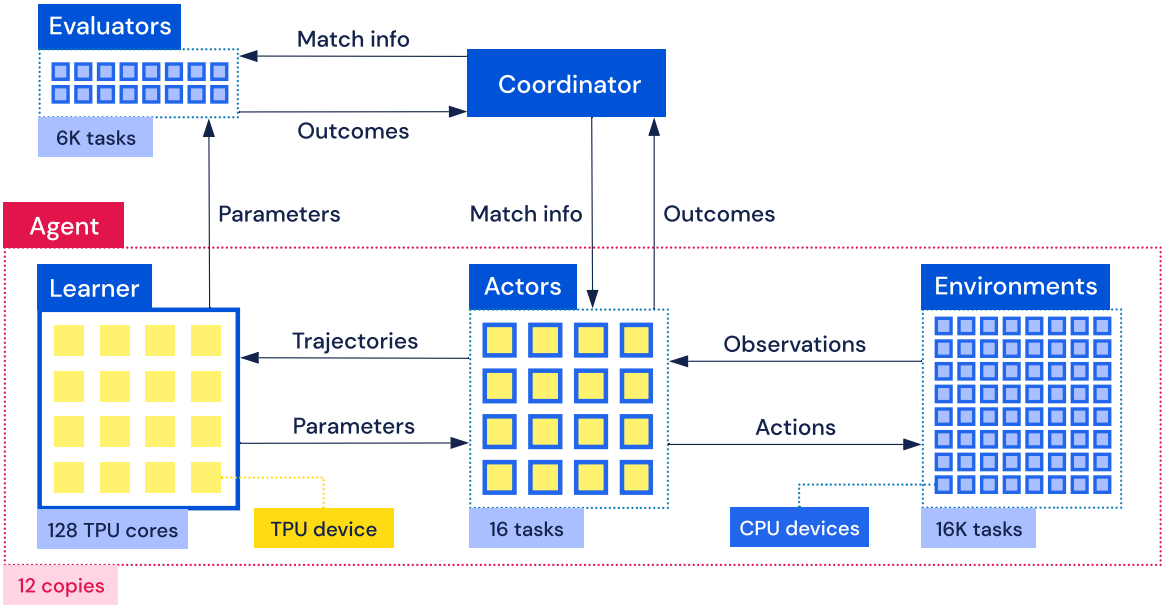
Extended Data Fig. 3 | Overview of the architecture of AlphaStar. A detailed description can be found in Supplementary Data Architecture.



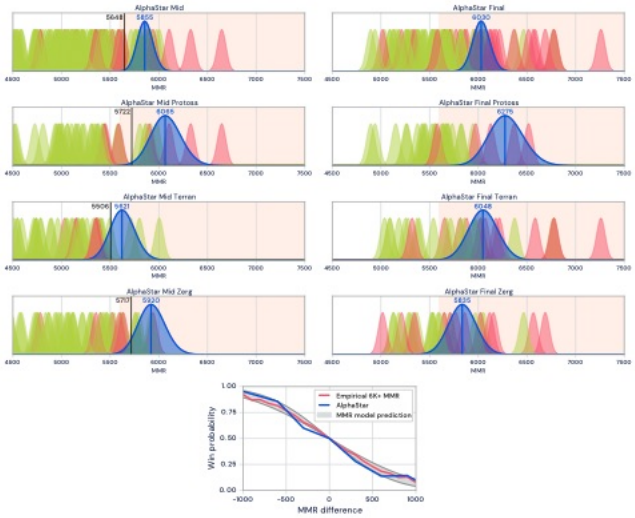
Extended Data Fig. 4 | Distribution of units built in a game. Units built by Protoss AlphaStar Supervised (left) and AlphaStar Final (right) over multiple self-play games. AlphaStar Supervised can build every unit.



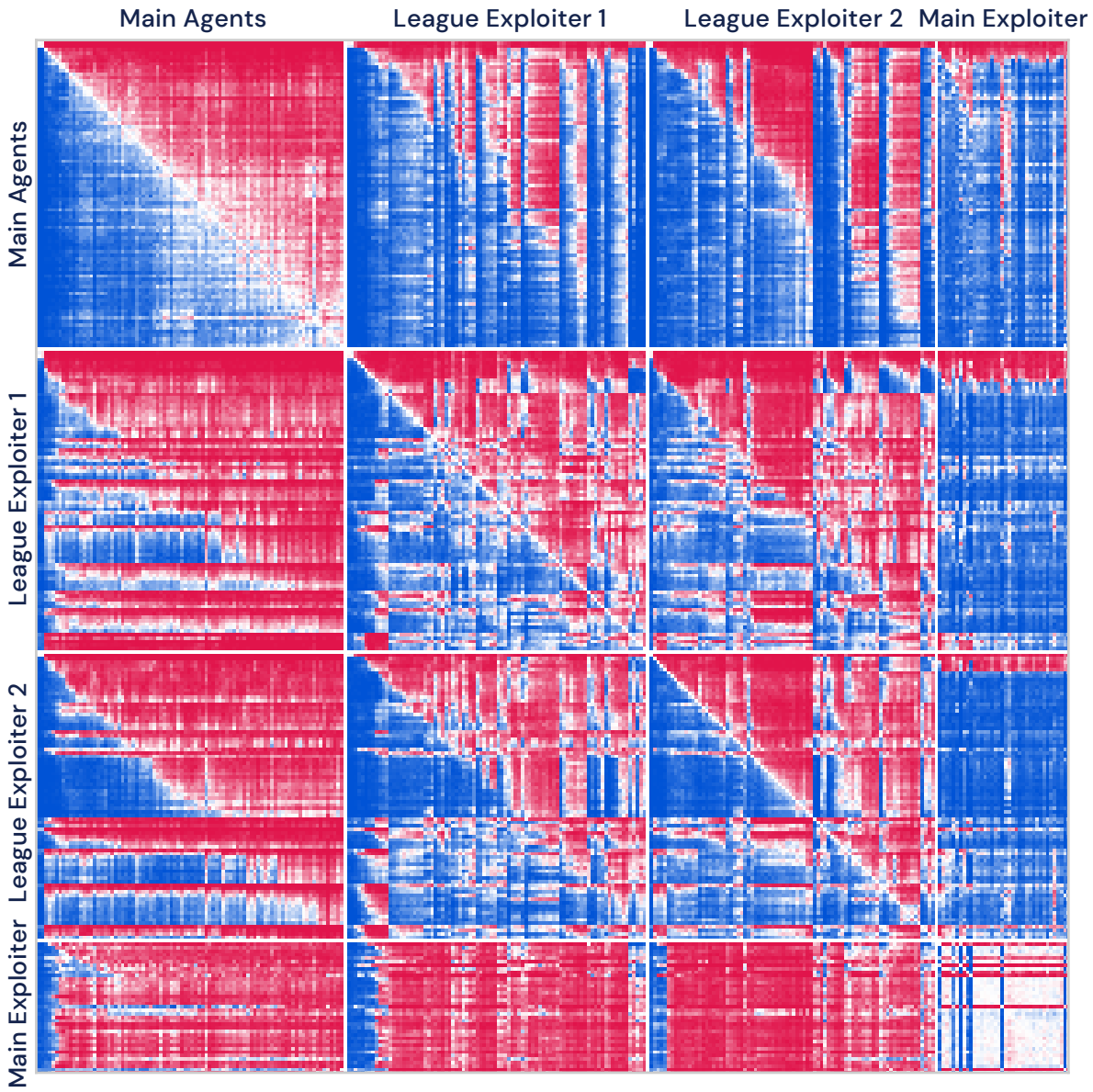
Extended Data Fig. 5 | A more detailed analysis of multi-agent ablations from Figure 3 C and D. PFSP based training outperforms FSP under all measures considered: it has a stronger population measured by relative population performance, provides a less exploitable solution, and has better final agent performance against the corresponding League.



Extended Data Fig. 6 | Training Infrastructure. Diagram of the training setup for the entire League.



Extended Data Fig. 7 | *Battle.net* performance details. (**Top**) Visualisation of all the matches played by AlphaStar Final (right) and matches against opponents above 4500 MMR of AlphaStar Mid (left). Each Gaussian represents an opponent MMR (with uncertainty): AlphaStar won against green opponents and lost to red. Blue is our MMR estimate, and black is the MMR reported by StarCraft II. The orange background is the Grandmaster league range. (**Bottom**) Win probability versus gap in MMR. The shaded grey region shows MMR model predictions when varying players' uncertainty. The red and blue line are empirical win-rates for players above 6000 MMR and AlphaStar Final respectively. Both human and AlphaStar win-rates closely follow the MMR model.



Extended Data Fig. 8 | Payoff matrix (limited to only Protoss vs. Protoss for simplicity), split into agent types of the League. Blue means a row agent wins, red loses, and white draws. The main agents behave transitively: the more recent agents win consistently against older main agents and exploiters. Interactions between exploiters are highly non-transitive – across the full payoff, there are around 3,000,000 rock-paper-scissor cycles (with requirement of at least 70% win rates to form a cycle) that involve at least one exploiter, and around 200 that involve only main agents.

Category	Field	Description
Entities: up to 512	Unit type	E.g. Drone or Forcefield
	Owner	Agent, opponent, or neutral
	Status	Current health, shields, energy
	Display type	E.g. Snapshot, for opponent buildings in the fog of war
	Position	Entity position
	Number of workers	For resource collecting base buildings
	Cooldowns	Attack cooldown
	Attributes	Invisible, powered, hallucination, active, in cargo, and/or on the screen
	Unit attributes	E.g. Biological or Armored
	Cargo status	Current and maximum amount of cargo space
	Building status	Build progress, build queue, and add-on type
	Resource status	Remaining resource contents
	Order status	Order queue and order progress
	Buff status	Buffs and buff durations
Map: 128x128 grid	Height	Heights of map locations
	Visibility	Whether map locations are currently visible
	Creep	Whether there is creep at a specific location
	Entity owners	Which player owns entities
	Alerts	Whether units are under attack
	Pathable	Which areas can be navigated over
	Buildable	Which areas can be built on
Player data	Race	Agent and opponent requested race, and agent actual race
	Upgrades	Agent upgrades and opponent upgrades, if they would be known to humans
	Agent statistics	Agent current resources, supply, army supply, worker supply, maximum supply, number of idle workers, number of Warp Gates, and number of Larva
Game statistics	Camera	Current camera position. The camera is a 32x20 game-unit sized rectangle
	Time	Current time in game

Extended Data Table 1 | Agent input space

The observations the agent receives through the raw interface. Information is hidden if it would be hidden to a human player. For example, AlphaStar will not see most information about invisible opponent units unless there is a detector; opponent units hidden by the fog of war will not appear in the list of units; opponent units outside of the agent's camera will only have the owner, display type, and position; and opponent's cloaked units will only appear in the list if they are within the agent's camera. Note that this interface displays information that must be inferred or remembered by humans, like the armour upgrades of a visible opponent unit, attack cool-downs, or entities occluded by other entities.

Field	Description
Action type	Which action to execute. Some examples of actions are moving a unit, training a unit from a building, moving the camera, or no-op. See PySC2 for a full list[6]
Selected units	Entities that will execute the action
Target	An entity or location in the map discretised to 256x256 targeted by the action
Queued	Whether to queue this action or execute it immediately
Repeat	Whether or not to issue this action multiple times
Delay	The number of game time-steps to wait until receiving the next observation

Extended Data Table 2 | Agent action space

The action arguments the agents can submit through the raw interface as part of an action. Some fields may be ignored depending on the action type.