ARTICLE

Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts[1–4]. However, expert data sets are often expensive, unreliable or simply unavailable. Even when reliable data sets are available, they may impose a ceiling on the performance of systems trained in this manner[5]. By contrast, reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking. Recently, there has been rapid progress towards this goal, using deep neural networks trained by reinforcement learning. These systems have outperformed humans in computer games, such as Atari[6,7] and 3D virtual environments[8–10]. However, the most challenging domains in terms of human intellect—such as the game of Go, widely viewed as a grand challenge for artificial intelligence[11]—require a precise and sophisticated lookahead in vast search spaces. Fully general methods have not previously achieved human-level performance in these domains.

AlphaGo was the first program to achieve superhuman performance in Go. The published version[12], which we refer to as AlphaGo Fan, defeated the European champion Fan Hui in October 2015. AlphaGo Fan used two deep neural networks: a policy network that outputs move probabilities and a value network that outputs a position evaluation. The policy network was trained initially by supervised learning to accurately predict human expert moves, and was subsequently refined by policy-gradient reinforcement learning. The value network was trained to predict the winner of games played by the policy network against itself. Once trained, these networks were combined with a Monte Carlo tree search (MCTS)[13–15] to provide a lookahead search, using the policy network to narrow down the search to high-probability moves, and using the value network (in conjunction with Monte Carlo rollouts using a fast rollout policy) to evaluate positions in the tree. A subsequent version, which we refer to as AlphaGo Lee, used a similar approach (see Methods), and defeated Lee Sedol, the winner of 18 international titles, in March 2016.

Our program, AlphaGo Zero, differs from AlphaGo Fan and AlphaGo Lee[12] in several important aspects. First and foremost, it is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts. To achieve these results, we introduce a new reinforcement learning algorithm that incorporates lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning. Further technical differences in the search algorithm, training procedure and network architecture are described in Methods.
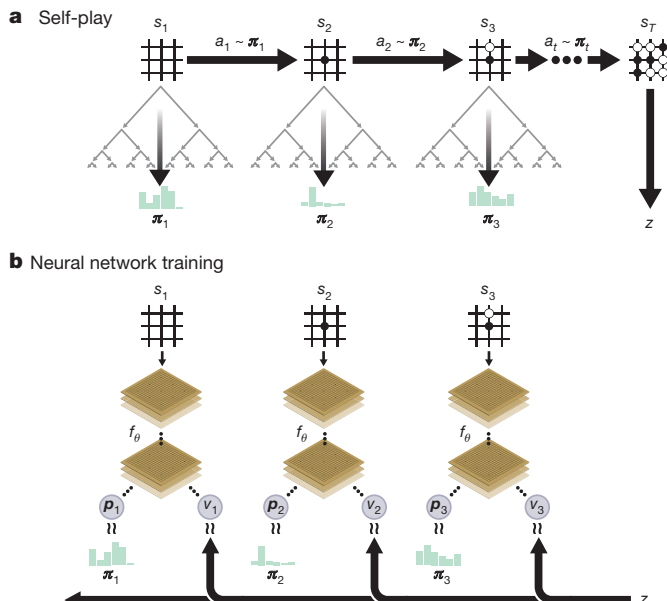
Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities $\boldsymbol{p}$ represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network[12] into a single architecture. The neural network consists of many residual blocks[4] of convolutional layers[16,17] with batch normalization[18] and rectifier nonlinearities[19] (see Methods).

The neural network in AlphaGo Zero is trained from games of self-play by a novel reinforcement learning algorithm. In each position s, an MCTS search is executed, guided by the neural network f_θ. The MCTS search outputs probabilities $\boldsymbol{\pi}$ of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities $\boldsymbol{p}$ of the neural network $f_\theta(s)$; MCTS may therefore be viewed as a powerful policy improvement operator[20,21]. Self-play with search—using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value—may be viewed as a powerful policy evaluation operator. The main idea of our reinforcement learning algorithm is to use these search operators

[1]DeepMind, 5 New Street Square, London EC4A 3TW, UK.
*These authors contributed equally to this work.

Figure 1 | Self-play reinforcement learning in AlphaGo Zero. a, The program plays a game $s_1, ..., s_T$ against itself. In each position s_t, an MCTS α_θ is executed (see Fig. 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z. **b,** Neural network training in AlphaGo Zero. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ, and outputs both a vector p_t, representing a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in position s_t. The neural network parameters θ are updated to maximize the similarity of the policy vector p_t to the search probabilities π_t, and to minimize the error between the predicted winner v_t and the game winner z (see equation (1)). The new parameters are used in the next iteration of self-play as in **a**.

repeatedly in a policy iteration procedure[22,23]: the neural network's parameters are updated to make the move probabilities and value (p, v) $= f_\theta(s)$ more closely match the improved search probabilities and self-play winner (π, z); these new parameters are used in the next iteration of self-play to make the search even stronger. Figure 1 illustrates the self-play training pipeline.

The MCTS uses the neural network f_θ to guide its simulations (see Fig. 2). Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximize

an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a) / (1 + N(s, a))$ (refs 12, 24), until a leaf node s' is encountered. This leaf position is expanded and evaluated only once by the network to generate both prior probabilities and evaluation, $(P(s', \cdot), V(s')) = f_\theta(s')$. Each edge ($s$, a) traversed in the simulation is updated to increment its visit count $N(s, a)$, and to update its action value to the mean evaluation over these simulations, $Q(s, a) = 1/N(s, a) \sum_{s'|s,a \to s'} V(s')$ where $s, a \to s'$ indicates that a simulation eventually reached s' after taking move a from position s.

MCTS may be viewed as a self-play algorithm that, given neural network parameters θ and a root position s, computes a vector of search probabilities recommending moves to play, $\pi = \alpha_\theta(s)$, proportional to the exponentiated visit count for each move, $\pi_a \propto N(s, a)^{1/\tau}$, where τ is a temperature parameter.

The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move. First, the neural network is initialized to random weights θ_0. At each subsequent iteration $i \geq 1$, games of self-play are generated (Fig. 1a). At each time-step t, an MCTS search $\pi_t = \alpha_{\theta_{i-1}}(s_t)$ is executed using the previous iteration of neural network $f_{\theta_{i-1}}$ and a move is played by sampling the search probabilities π_t. A game terminates at step T when both players pass, when the search value drops below a resignation threshold or when the game exceeds a maximum length; the game is then scored to give a final reward of $r_T \in \{-1, +1\}$ (see Methods for details). The data for each time-step t is stored as (s_t, π_t, z_t), where $z_t = \pm r_T$ is the game winner from the perspective of the current player at step t. In parallel (Fig. 1b), new network parameters θ_i are trained from data (s, π, z) sampled uniformly among all time-steps of the last iteration(s) of self-play. The neural network (p, v) $= f_{\theta_i}(s)$ is adjusted to minimize the error between the predicted value v and the self-play winner z, and to maximize the similarity of the neural network move probabilities p to the search probabilities π. Specifically, the parameters θ are adjusted by gradient descent on a loss function l that sums over the mean-squared error and cross-entropy losses, respectively:

$$(p, v) = f_\theta(s) \quad \text{and} \quad l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2 \quad (1)$$

where c is a parameter controlling the level of L2 weight regularization (to prevent overfitting).

Empirical analysis of AlphaGo Zero training

We applied our reinforcement learning pipeline to train our program AlphaGo Zero. Training started from completely random behaviour and continued without human intervention for approximately three days.

Over the course of training, 4.9 million games of self-play were generated, using 1,600 simulations for each MCTS, which corresponds to approximately 0.4 s thinking time per move. Parameters were updated
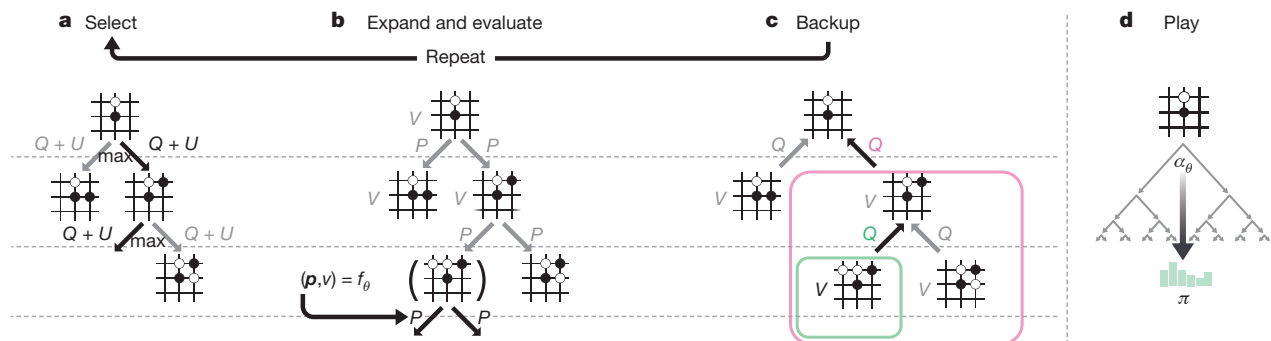
Figure 2 | MCTS in AlphaGo Zero. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus an upper confidence bound U that depends on a stored prior probability P and visit count N for that edge (which is incremented once traversed). **b,** The leaf node is expanded and the associated position s is evaluated by the neural network ($P(s, \cdot), V(s)) = f_\theta(s)$; the vector of P values are stored in

the outgoing edges from s. **c,** Action value Q is updated to track the mean of all evaluations V in the subtree below that action. **d,** Once the search is complete, search probabilities π are returned, proportional to $N^{1/\tau}$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature.

Figure 3 | Empirical evaluation of AlphaGo Zero. a, Performance of self-play reinforcement learning. The plot shows the performance of each MCTS player α_{θ_i} from each iteration i of reinforcement learning in AlphaGo Zero. Elo ratings were computed from evaluation games between different players, using 0.4 s of thinking time per move (see Methods). For comparison, a similar player trained by supervised learning from human data, using the KGS dataset, is also shown. **b**, Prediction accuracy on human professional moves. The plot shows the accuracy of the neural network f_{θ_i}, at each iteration of self-play i, in predicting human professional moves from the GoKifu dataset. The accuracy measures the

percentage of positions in which the neural network assigns the highest probability to the human move. The accuracy of a neural network trained by supervised learning is also shown. **c**, Mean-squared error (MSE) of human professional game outcomes. The plot shows the MSE of the neural network f_{θ_i}, at each iteration of self-play i, in predicting the outcome of human professional games from the GoKifu dataset. The MSE is between the actual outcome $z \in \{-1, +1\}$ and the neural network value v, scaled by a factor of $\frac{1}{4}$ to the range of 0–1. The MSE of a neural network trained by supervised learning is also shown.

from 700,000 mini-batches of 2,048 positions. The neural network contained 20 residual blocks (see Methods for further details).

Figure 3a shows the performance of AlphaGo Zero during self-play reinforcement learning, as a function of training time, on an Elo scale[25]. Learning progressed smoothly throughout training, and did not suffer from the oscillations or catastrophic forgetting that have been suggested in previous literature[26–28]. Surprisingly, AlphaGo Zero outperformed AlphaGo Lee after just 36 h. In comparison, AlphaGo Lee was trained over several months. After 72 h, we evaluated AlphaGo Zero against the exact version of AlphaGo Lee that defeated Lee Sedol, under the same 2 h time controls and match conditions that were used in the man–machine match in Seoul (see Methods). AlphaGo Zero used a single machine with 4 tensor processing units (TPUs)[29], whereas AlphaGo Lee was distributed over many machines and used 48 TPUs. AlphaGo Zero defeated AlphaGo Lee by 100 games to 0 (see Extended Data Fig. 1 and Supplementary Information).

To assess the merits of self-play reinforcement learning, compared to learning from human data, we trained a second neural network (using the same architecture) to predict expert moves in the KGS Server dataset; this achieved state-of-the-art prediction accuracy compared to previous work[12,30–33] (see Extended Data Tables 1 and 2 for current and previous results, respectively). Supervised learning achieved a better initial performance, and was better at predicting human professional moves (Fig. 3). Notably, although supervised learning achieved higher move prediction accuracy, the self-learned player performed much better overall, defeating the human-trained player within the first 24 h of training. This suggests that AlphaGo Zero may be learning a strategy that is qualitatively different to human play.

To separate the contributions of architecture and algorithm, we compared the performance of the neural network architecture in AlphaGo Zero with the previous neural network architecture used in AlphaGo Lee (see Fig. 4). Four neural networks were created, using

Figure 4 | Comparison of neural network architectures in AlphaGo Zero and AlphaGo Lee. Comparison of neural network architectures using either separate (sep) or combined policy and value (dual) networks, and using either convolutional (conv) or residual (res) networks. The combinations 'dual–res' and 'sep–conv' correspond to the neural network architectures used in AlphaGo Zero and AlphaGo Lee, respectively. Each network was trained on a fixed dataset generated by a previous run of

AlphaGo Zero. **a**, Each trained network was combined with AlphaGo Zero's search to obtain a different player. Elo ratings were computed from evaluation games between these different players, using 5 s of thinking time per move. **b**, Prediction accuracy on human professional moves (from the GoKifu dataset) for each network architecture. **c** MSE of human professional game outcomes (from the GoKifu dataset) for each network architecture.

Figure 5 | Go knowledge learned by AlphaGo Zero. a, Five human *joseki* (common corner sequences) discovered during AlphaGo Zero training. The associated timestamps indicate the first time each sequence occurred (taking account of rotation and reflection) during self-play training. Extended Data Figure 2 provides the frequency of occurence over training for each sequence. **b**, Five *joseki* favoured at different stages of self-play training. Each displayed corner sequence was played with the greatest frequency, among all corner sequences, during an iteration of self-play training. The timestamp of that iteration is indicated on the timeline. At 10 h a weak corner move was preferred. At 47 h the 3–3 invasion was most frequently played. This *joseki* is also common in human professional play;

however AlphaGo Zero later discovered and preferred a new variation. Extended Data Figure 3 provides the frequency of occurence over time for all five sequences and the new variation. **c**, The first 80 moves of three self-play games that were played at different stages of training, using 1,600 simulations (around 0.4 s) per search. At 3 h, the game focuses greedily on capturing stones, much like a human beginner. At 19 h, the game exhibits the fundamentals of life-and-death, influence and territory. At 70 h, the game is remarkably balanced, involving multiple battles and a complicated *ko* fight, eventually resolving into a half-point win for white. See Supplementary Information for the full games.

either separate policy and value networks, as were used in AlphaGo Lee, or combined policy and value networks, as used in AlphaGo Zero; and using either the convolutional network architecture from AlphaGo Lee or the residual network architecture from AlphaGo Zero. Each network was trained to minimize the same loss function (equation (1)), using a fixed dataset of self-play games generated by AlphaGo Zero after 72 h of self-play training. Using a residual network was more accurate, achieved lower error and improved performance in AlphaGo by over 600 Elo. Combining policy and value together into a single network slightly reduced the move prediction accuracy, but reduced the value error and boosted playing performance in AlphaGo by around

another 600 Elo. This is partly due to improved computational efficiency, but more importantly the dual objective regularizes the network to a common representation that supports multiple use cases.

Knowledge learned by AlphaGo Zero

AlphaGo Zero discovered a remarkable level of Go knowledge during its self-play training process. This included not only fundamental elements of human Go knowledge, but also non-standard strategies beyond the scope of traditional Go knowledge.

Figure 5 shows a timeline indicating when professional *joseki* (corner sequences) were discovered (Fig. 5a and Extended Data

Figure 6 | Performance of AlphaGo Zero. a, Learning curve for AlphaGo Zero using a larger 40-block residual network over 40 days. The plot shows the performance of each player α_{θ_i} from each iteration i of our reinforcement learning algorithm. Elo ratings were computed from evaluation games between different players, using 0.4 s per search (see Methods). **b**, Final performance of AlphaGo Zero. AlphaGo Zero was trained for 40 days using a 40-block residual neural network. The plot shows the results of a tournament between: AlphaGo Zero, AlphaGo Master (defeated top human professionals 60–0 in online games), AlphaGo

Lee (defeated Lee Sedol), AlphaGo Fan (defeated Fan Hui), as well as previous Go programs Crazy Stone, Pachi and GnuGo. Each program was given 5 s of thinking time per move. AlphaGo Zero and AlphaGo Master played on a single machine on the Google Cloud; AlphaGo Fan and AlphaGo Lee were distributed over many machines. The raw neural network from AlphaGo Zero is also included, which directly selects the move a with maximum probability p_a, without using MCTS. Programs were evaluated on an Elo scale[25]: a 200-point gap corresponds to a 75% probability of winning.

Fig. 2); ultimately AlphaGo Zero preferred new *joseki* variants that were previously unknown (Fig. 5b and Extended Data Fig. 3). Figure 5c shows several fast self-play games played at different stages of training (see Supplementary Information). Tournament length games played at regular intervals throughout training are shown in Extended Data Fig. 4 and in the Supplementary Information. AlphaGo Zero rapidly progressed from entirely random moves towards a sophisticated understanding of Go concepts, including *fuseki* (opening), *tesuji* (tactics), life-and-death, *ko* (repeated board situations), *yose* (endgame), capturing races, *sente* (initiative), shape, influence and territory, all discovered from first principles. Surprisingly, *shicho* ('ladder' capture sequences that may span the whole board)—one of the first elements of Go knowledge learned by humans—were only understood by AlphaGo Zero much later in training.

Final performance of AlphaGo Zero
We subsequently applied our reinforcement learning pipeline to a second instance of AlphaGo Zero using a larger neural network and over a longer duration. Training again started from completely random behaviour and continued for approximately 40 days.

Over the course of training, 29 million games of self-play were generated. Parameters were updated from 3.1 million mini-batches of 2,048 positions each. The neural network contained 40 residual blocks. The learning curve is shown in Fig. 6a. Games played at regular intervals throughout training are shown in Extended Data Fig. 5 and in the Supplementary Information.

We evaluated the fully trained AlphaGo Zero using an internal tournament against AlphaGo Fan, AlphaGo Lee and several previous Go programs. We also played games against the strongest existing program, AlphaGo Master—a program based on the algorithm and architecture presented in this paper but using human data and features (see Methods)—which defeated the strongest human professional players 60–0 in online games in January 2017[34]. In our evaluation, all programs were allowed 5 s of thinking time per move; AlphaGo Zero and AlphaGo Master each played on a single machine with 4 TPUs; AlphaGo Fan and AlphaGo Lee were distributed over 176 GPUs and 48 TPUs, respectively. We also included a player based solely on the raw neural network of AlphaGo Zero; this player simply selected the move with maximum probability.

Figure 6b shows the performance of each program on an Elo scale. The raw neural network, without using any lookahead, achieved an Elo rating of 3,055. AlphaGo Zero achieved a rating of 5,185, compared

to 4,858 for AlphaGo Master, 3,739 for AlphaGo Lee and 3,144 for AlphaGo Fan.

Finally, we evaluated AlphaGo Zero head to head against AlphaGo Master in a 100-game match with 2-h time controls. AlphaGo Zero won by 89 games to 11 (see Extended Data Fig. 6 and Supplementary Information).

Conclusion
Our results comprehensively demonstrate that a pure reinforcement learning approach is fully feasible, even in the most challenging of domains: it is possible to train to superhuman level, without human examples or guidance, given no knowledge of the domain beyond basic rules. Furthermore, a pure reinforcement learning approach requires just a few more hours to train, and achieves much better asymptotic performance, compared to training on human expert data. Using this approach, AlphaGo Zero defeated the strongest previous versions of AlphaGo, which were trained from human data using handcrafted features, by a large margin.

Humankind has accumulated Go knowledge from millions of games played over thousands of years, collectively distilled into patterns, proverbs and books. In the space of a few days, starting *tabula rasa*, AlphaGo Zero was able to rediscover much of this Go knowledge, as well as novel strategies that provide new insights into the oldest of games.

1. Friedman, J., Hastie, T. & Tibshirani, R. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (Springer, 2009).
2. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *Nature* **521,** 436–444 (2015).
3. Krizhevsky, A., Sutskever, I. & Hinton, G. ImageNet classification with deep convolutional neural networks. In *Adv. Neural Inf. Process. Syst.* Vol. 25 (eds Pereira, F., Burges, C. J. C., Bottou, L. & Weinberger, K. Q.) 1097–1105 (2012).
4. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition. In *Proc. 29th IEEE Conf. Comput. Vis. Pattern Recognit.* 770–778 (2016).
5. Hayes-Roth, F., Waterman, D. & Lenat, D. *Building Expert Systems* (Addison-Wesley, 1984).
6. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529–533 (2015).
7. Guo, X., Singh, S. P., Lee, H., Lewis, R. L. & Wang, X. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In *Adv. Neural Inf. Process. Syst.* Vol. 27 (eds Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D. & Weinberger, K. Q.) 3338–3346 (2014).

8. Mnih, V. *et al.* Asynchronous methods for deep reinforcement learning. In *Proc. 33rd Int. Conf. Mach. Learn.* Vol. 48 (eds Balcan, M. F. & Weinberger, K. Q.) 1928–1937 (2016).
9. Jaderberg, M. *et al.* Reinforcement learning with unsupervised auxiliary tasks. In *5th Int. Conf. Learn. Representations* (2017).
10. Dosovitskiy, A. & Koltun, V. Learning to act by predicting the future. In *5th Int. Conf. Learn. Representations* (2017).
11. Mańdziuk, J. in *Challenges for Computational Intelligence* (Duch, W. & Mańdziuk, J.) 407–442 (Springer, 2007).
12. Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *Nature* **529,** 484–489 (2016).
13. Coulom, R. Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th Int. Conf. Computers and Games* (eds Ciancarini, P. & van den Herik, H. J.) 72–83 (2006).
14. Kocsis, L. & Szepesvári, C. Bandit based Monte-Carlo planning. In *15th Eu. Conf. Mach. Learn.* 282–293 (2006).
15. Browne, C. *et al.* A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games* **4,** 1–49 (2012).
16. Fukushima, K. Neocognitron: a self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol. Cybern.* **36,** 193–202 (1980).
17. LeCun, Y. & Bengio, Y. in *The Handbook of Brain Theory and Neural Networks* Ch. 3 (ed. Arbib, M.) 276–278 (MIT Press, 1995).
18. Ioffe, S. & Szegedy, C. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proc. 32nd Int. Conf. Mach. Learn.* Vol. 37 448–456 (2015).
19. Hahnloser, R. H. R., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J. & Seung, H. S. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature* **405,** 947–951 (2000).
20. Howard, R. *Dynamic Programming and Markov Processes* (MIT Press, 1960).
21. Sutton, R. & Barto, A. *Reinforcement Learning: an Introduction* (MIT Press, 1998).
22. Bertsekas, D. P. Approximate policy iteration: a survey and some new methods. *J. Control Theory Appl.* **9,** 310–335 (2011).
23. Scherrer, B. Approximate policy iteration schemes: a comparison. In *Proc. 31st Int. Conf. Mach. Learn.* Vol. 32 1314–1322 (2014).
24. Rosin, C. D. Multi-armed bandits with episode context. *Ann. Math. Artif. Intell.* **61,** 203–230 (2011).
25. Coulom, R. Whole-history rating: a Bayesian rating system for players of time-varying strength. In *Int. Conf. Comput. Games* (eds van den Herik, H. J., Xu, X. Ma, Z. & Winands, M. H. M.) Vol. 5131 113–124 (Springer, 2008).
26. Laurent, G. J., Matignon, L. & Le Fort-Piat, N. The world of independent learners is not Markovian. *Int. J. Knowledge-Based Intelligent Engineering Systems* **15,** 55–64 (2011).
27. Foerster, J. N. *et al.* Stabilising experience replay for deep multi-agent reinforcement learning. In *Proc. 34th Int. Conf. Mach. Learn.* Vol. 70 1146–1155 (2017).
28. Heinrich, J. & Silver, D. Deep reinforcement learning from self-play in imperfect-information games. In *NIPS Deep Reinforcement Learning Workshop* (2016).
29. Jouppi, N. P. *et al.* In-datacenter performance analysis of a Tensor Processing Unit. *Proc. 44th Annu. Int. Symp. Comp. Architecture* Vol. 17 1–12 (2017).
30. Maddison, C. J., Huang, A., Sutskever, I. & Silver, D. Move evaluation in Go using deep convolutional neural networks. In *3rd Int. Conf. Learn. Representations.* (2015).
31. Clark, C. & Storkey, A. J. Training deep convolutional neural networks to play Go. In *Proc. 32nd Int. Conf. Mach. Learn.* Vol. 37 1766–1774 (2015).
32. Tian, Y. & Zhu, Y. Better computer Go player with neural network and long-term prediction. In *4th Int. Conf. Learn. Representations* (2016).
33. Cazenave, T. Residual networks for computer Go. *IEEE Trans. Comput. Intell. AI Games* https://doi.org/10.1109/TCIAIG.2017.2681042 (2017).
34. Huang, A. AlphaGo master online series of games. https://deepmind.com/research/AlphaGo/match-archive/master (2017).

METHODS

Reinforcement learning. Policy iteration[20,21] is a classic algorithm that generates a sequence of improving policies, by alternating between policy evaluation—estimating the value function of the current policy—and policy improvement—using the current value function to generate a better policy. A simple approach to policy evaluation is to estimate the value function from the outcomes of sampled trajectories[35,36]. A simple approach to policy improvement is to select actions greedily with respect to the value function[20]. In large state spaces, approximations are necessary to evaluate each policy and to represent its improvement[22,23].

Classification-based reinforcement learning[37] improves the policy using a simple Monte Carlo search. Many rollouts are executed for each action; the action with the maximum mean value provides a positive training example, while all other actions provide negative training examples; a policy is then trained to classify actions as positive or negative, and used in subsequent rollouts. This may be viewed as a precursor to the policy component of AlphaGo Zero's training algorithm when $\tau \rightarrow 0$.

A more recent instantiation, classification-based modified policy iteration (CBMPI), also performs policy evaluation by regressing a value function towards truncated rollout values, similar to the value component of AlphaGo Zero; this achieved state-of-the-art results in the game of Tetris[38]. However, this previous work was limited to simple rollouts and linear function approximation using hand-crafted features.

The AlphaGo Zero self-play algorithm can similarly be understood as an approximate policy iteration scheme in which MCTS is used for both policy improvement and policy evaluation. Policy improvement starts with a neural network policy, executes an MCTS based on that policy's recommendations, and then projects the (much stronger) search policy back into the function space of the neural network. Policy evaluation is applied to the (much stronger) search policy: the outcomes of self-play games are also projected back into the function space of the neural network. These projection steps are achieved by training the neural network parameters to match the search probabilities and self-play game outcome respectively.

Guo *et al.*[7] also project the output of MCTS into a neural network, either by regressing a value network towards the search value, or by classifying the action selected by MCTS. This approach was used to train a neural network for playing Atari games; however, the MCTS was fixed—there was no policy iteration—and did not make any use of the trained networks.

Self-play reinforcement learning in games. Our approach is most directly applicable to Zero-sum games of perfect information. We follow the formalism of alternating Markov games described in previous work[12], noting that algorithms based on value or policy iteration extend naturally to this setting[39].

Self-play reinforcement learning has previously been applied to the game of Go. NeuroGo[40,41] used a neural network to represent a value function, using a sophisticated architecture based on Go knowledge regarding connectivity, territory and eyes. This neural network was trained by temporal-difference learning[42] to predict territory in games of self-play, building on previous work[43]. A related approach, RLGO[44], represented the value function instead by a linear combination of features, exhaustively enumerating all 3×3 patterns of stones; it was trained by temporal-difference learning to predict the winner in games of self-play. Both NeuroGo and RLGO achieved a weak amateur level of play.

MCTS may also be viewed as a form of self-play reinforcement learning[45]. The nodes of the search tree contain the value function for the positions encountered during search; these values are updated to predict the winner of simulated games of self-play. MCTS programs have previously achieved strong amateur level in Go[46,47], but used substantial domain expertise: a fast rollout policy, based on handcrafted features[13,48], that evaluates positions by running simulations until the end of the game; and a tree policy, also based on handcrafted features, that selects moves within the search tree[47].

Self-play reinforcement learning approaches have achieved high levels of performance in other games: chess[49–51], checkers[52], backgammon[53], othello[54], Scrabble[55] and most recently poker[56]. In all of these examples, a value function was trained by regression[54–56] or temporal-difference learning[49–53] from training data generated by self-play. The trained value function was used as an evaluation function in an alpha–beta search[49–54], a simple Monte Carlo search[55,57] or counterfactual regret minimization[56]. However, these methods used handcrafted input features[49–53,56] or handcrafted feature templates[54,55]. In addition, the learning process used supervised learning to initialize weights[58], hand-selected weights for piece values[49,51,52], handcrafted restrictions on the action space[56] or used pre-existing computer programs as training opponents[49,50], or to generate game records[51].

Many of the most successful and widely used reinforcement learning methods were first introduced in the context of Zero-sum games: temporal-difference learning was first introduced for a checkers-playing program[59], while MCTS was introduced for the game of Go[13]. However, very similar algorithms have subsequently proven highly effective in video games[6–8,10], robotics[60], industrial control[61–63] and online recommendation systems[64,65].

AlphaGo versions. We compare three distinct versions of AlphaGo:

(1) AlphaGo Fan is the previously published program[12] that played against Fan Hui in October 2015. This program was distributed over many machines using 176 GPUs.

(2) AlphaGo Lee is the program that defeated Lee Sedol 4–1 in March 2016. It was previously unpublished, but is similar in most regards to AlphaGo Fan[12]. However, we highlight several key differences to facilitate a fair comparison. First, the value network was trained from the outcomes of fast games of self-play by AlphaGo, rather than games of self-play by the policy network; this procedure was iterated several times—an initial step towards the *tabula rasa* algorithm presented in this paper. Second, the policy and value networks were larger than those described in the original paper—using 12 convolutional layers of 256 planes—and were trained for more iterations. This player was also distributed over many machines using 48 TPUs, rather than GPUs, enabling it to evaluate neural networks faster during search.

(3) AlphaGo Master is the program that defeated top human players by 60–0 in January 2017[34]. It was previously unpublished, but uses the same neural network architecture, reinforcement learning algorithm, and MCTS algorithm as described in this paper. However, it uses the same handcrafted features and rollouts as AlphaGo Lee[12] and training was initialized by supervised learning from human data.

(4) AlphaGo Zero is the program described in this paper. It learns from self-play reinforcement learning, starting from random initial weights, without using rollouts, with no human supervision and using only the raw board history as input features. It uses just a single machine in the Google Cloud with 4 TPUs (AlphaGo Zero could also be distributed, but we chose to use the simplest possible search algorithm).

Domain knowledge. Our primary contribution is to demonstrate that superhuman performance can be achieved without human domain knowledge. To clarify this contribution, we enumerate the domain knowledge that AlphaGo Zero uses, explicitly or implicitly, either in its training procedure or its MCTS; these are the items of knowledge that would need to be replaced for AlphaGo Zero to learn a different (alternating Markov) game.

(1) AlphaGo Zero is provided with perfect knowledge of the game rules. These are used during MCTS, to simulate the positions resulting from a sequence of moves, and to score any simulations that reach a terminal state. Games terminate when both players pass or after $19 \times 19 \times 2 = 722$ moves. In addition, the player is provided with the set of legal moves in each position.

(2) AlphaGo Zero uses Tromp–Taylor scoring[66] during MCTS simulations and self-play training. This is because human scores (Chinese, Japanese or Korean rules) are not well-defined if the game terminates before territorial boundaries are resolved. However, all tournament and evaluation games were scored using Chinese rules.

(3) The input features describing the position are structured as a 19×19 image; that is, the neural network architecture is matched to the grid-structure of the board.

(4) The rules of Go are invariant under rotation and reflection; this knowledge has been used in AlphaGo Zero both by augmenting the dataset during training to include rotations and reflections of each position, and to sample random rotations or reflections of the position during MCTS (see Search algorithm). Aside from *komi*, the rules of Go are also invariant to colour transposition; this knowledge is exploited by representing the board from the perspective of the current player (see Neural network architecture).

AlphaGo Zero does not use any form of domain knowledge beyond the points listed above. It only uses its deep neural network to evaluate leaf nodes and to select moves (see 'Search algorithm'). It does not use any rollout policy or tree policy, and the MCTS is not augmented by any other heuristics or domain-specific rules. No legal moves are excluded—even those filling in the player's own eyes (a standard heuristic used in all previous programs[67]).

The algorithm was started with random initial parameters for the neural network. The neural network architecture (see 'Neural network architecture') is based on the current state of the art in image recognition[4,18], and hyperparameters for training were chosen accordingly (see 'Self-play training pipeline'). MCTS search parameters were selected by Gaussian process optimization[68], so as to optimize self-play performance of AlphaGo Zero using a neural network trained in a preliminary run. For the larger run (40 blocks, 40 days), MCTS search parameters were re-optimized using the neural network trained in the smaller run (20 blocks, 3 days). The training algorithm was executed autonomously without human intervention.

Self-play training pipeline. AlphaGo Zero's self-play training pipeline consists of three main components, all executed asynchronously in parallel. Neural network parameters θ_i are continually optimized from recent self-play data; AlphaGo Zero

players α_{θ_i} are continually evaluated; and the best performing player so far, α_{θ_*}, is used to generate new self-play data.

Optimization. Each neural network f_{θ_i} is optimized on the Google Cloud using TensorFlow, with 64 GPU workers and 19 CPU parameter servers. The batch-size is 32 per worker, for a total mini-batch size of 2,048. Each mini-batch of data is sampled uniformly at random from all positions of the most recent 500,000 games of self-play. Neural network parameters are optimized by stochastic gradient descent with momentum and learning rate annealing, using the loss in equation (1). The learning rate is annealed according to the standard schedule in Extended Data Table 3. The momentum parameter is set to 0.9. The cross-entropy and MSE losses are weighted equally (this is reasonable because rewards are unit scaled, $r \in \{-1, +1\}$) and the L2 regularization parameter is set to $c = 10^{-4}$. The optimization process produces a new checkpoint every 1,000 training steps. This checkpoint is evaluated by the evaluator and it may be used for generating the next batch of self-play games, as we explain next.

Evaluator. To ensure we always generate the best quality data, we evaluate each new neural network checkpoint against the current best network f_{θ_*} before using it for data generation. The neural network f_{θ_i} is evaluated by the performance of an MCTS search α_{θ_i} that uses f_{θ_i} to evaluate leaf positions and prior probabilities (see Search algorithm). Each evaluation consists of 400 games, using an MCTS with 1,600 simulations to select each move, using an infinitesimal temperature $\tau \rightarrow 0$ (that is, we deterministically select the move with maximum visit count, to give the strongest possible play). If the new player wins by a margin of $>55\%$ (to avoid selecting on noise alone) then it becomes the best player α_{θ_*}, and is subsequently used for self-play generation, and also becomes the baseline for subsequent comparisons.

Self-play. The best current player α_{θ_*}, as selected by the evaluator, is used to generate data. In each iteration, α_{θ_*} plays 25,000 games of self-play, using 1,600 simulations of MCTS to select each move (this requires approximately 0.4 s per search). For the first 30 moves of each game, the temperature is set to $\tau = 1$; this selects moves proportionally to their visit count in MCTS, and ensures a diverse set of positions are encountered. For the remainder of the game, an infinitesimal temperature is used, $\tau \rightarrow 0$. Additional exploration is achieved by adding Dirichlet noise to the prior probabilities in the root node s_0, specifically $P(s, a) = (1 - \varepsilon)p_a + \varepsilon \eta_a$, where $\eta \sim \text{Dir}(0.03)$ and $\varepsilon = 0.25$; this noise ensures that all moves may be tried, but the search may still overrule bad moves. In order to save computation, clearly lost games are resigned. The resignation threshold v_{resign} is selected automatically to keep the fraction of false positives (games that could have been won if AlphaGo had not resigned) below 5%. To measure false positives, we disable resignation in 10% of self-play games and play until termination.

Supervised learning. For comparison, we also trained neural network parameters θ_{SL} by supervised learning. The neural network architecture was identical to AlphaGo Zero. Mini-batches of data $(s, \boldsymbol{\pi}, z)$ were sampled at random from the KGS dataset, setting $\pi_a = 1$ for the human expert move a. Parameters were optimized by stochastic gradient descent with momentum and learning rate annealing, using the same loss as in equation (1), but weighting the MSE component by a factor of 0.01. The learning rate was annealed according to the standard schedule in Extended Data Table 3. The momentum parameter was set to 0.9, and the L2 regularization parameter was set to $c = 10^{-4}$.

By using a combined policy and value network architecture, and by using a low weight on the value component, it was possible to avoid overfitting to the values (a problem described in previous work[12]). After 72 h the move prediction accuracy exceeded the state of the art reported in previous work[12,30–33], reaching 60.4% on the KGS test set; the value prediction error was also substantially better than previously reported[12]. The validation set was composed of professional games from GoKifu. Accuracies and MSEs are reported in Extended Data Table 1 and Extended Data Table 2, respectively.

Search algorithm. AlphaGo Zero uses a much simpler variant of the asynchronous policy and value MCTS algorithm (APV-MCTS) used in AlphaGo Fan and AlphaGo Lee.

Each node s in the search tree contains edges (s, a) for all legal actions $a \in \mathcal{A}(s)$. Each edge stores a set of statistics,

$$\{N(s, a), W(s, a), Q(s, a), P(s, a)\}$$

where $N(s, a)$ is the visit count, $W(s, a)$ is the total action value, $Q(s, a)$ is the mean action value and $P(s, a)$ is the prior probability of selecting that edge. Multiple simulations are executed in parallel on separate search threads. The algorithm proceeds by iterating over three phases (Fig. 2a–c), and then selects a move to play (Fig. 2d).

Select (Fig. 2a). The selection phase is almost identical to AlphaGo Fan[12]; we recapitulate here for completeness. The first in-tree phase of each simulation begins at the root node of the search tree, s_0, and finishes when the simulation reaches a leaf node s_L at time-step L. At each of these time-steps, $t < L$, an action is selected according to the statistics in the search tree, $a_t = \underset{a}{\text{argmax}}(Q(s_t, a) + U(s_t, a))$, using a variant of the PUCT algorithm[24],

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

where c_{puct} is a constant determining the level of exploration; this search control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

Expand and evaluate (Fig. 2b). The leaf node s_L is added to a queue for neural network evaluation, $(d_i(\boldsymbol{p}), v) = f_\theta(d_i(s_L))$, where d_i is a dihedral reflection or rotation selected uniformly at random from i in [1..8]. Positions in the queue are evaluated by the neural network using a mini-batch size of 8; the search thread is locked until evaluation completes. The leaf node is expanded and each edge (s_L, a) is initialized to $\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a\}$; the value v is then backed up.

Backup (Fig. 2c). The edge statistics are updated in a backward pass through each step $t \leq L$. The visit counts are incremented, $N(s_t, a_t) = N(s_t, a_t) + 1$, and the action value is updated to the mean value, $W(s_t, a_t) = W(s_t, a_t) + v$, $Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$. We use virtual loss to ensure each thread evaluates different nodes[12,69].

Play (Fig. 2d). At the end of the search AlphaGo Zero selects a move a to play in the root position s_0, proportional to its exponentiated visit count, $\pi(a|s_0) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau}$, where τ is a temperature parameter that controls the level of exploration. The search tree is reused at subsequent time-steps: the child node corresponding to the played action becomes the new root node; the subtree below this child is retained along with all its statistics, while the remainder of the tree is discarded. AlphaGo Zero resigns if its root value and best child value are lower than a threshold value v_{resign}.

Compared to the MCTS in AlphaGo Fan and AlphaGo Lee, the principal differences are that AlphaGo Zero does not use any rollouts; it uses a single neural network instead of separate policy and value networks; leaf nodes are always expanded, rather than using dynamic expansion; each search thread simply waits for the neural network evaluation, rather than performing evaluation and backup asynchronously; and there is no tree policy. A transposition table was also used in the large (40 blocks, 40 days) instance of AlphaGo Zero.

Neural network architecture. The input to the neural network is a $19 \times 19 \times 17$ image stack comprising 17 binary feature planes. Eight feature planes, X_t, consist of binary values indicating the presence of the current player's stones ($X_t^i = 1$ if intersection i contains a stone of the player's colour at time-step t; 0 if the intersection is empty, contains an opponent stone, or if $t < 0$). A further 8 feature planes, Y_t, represent the corresponding features for the opponent's stones. The final feature plane, C, represents the colour to play, and has a constant value of either 1 if black is to play or 0 if white is to play. These planes are concatenated together to give input features $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, ..., X_{t-7}, Y_{t-7}, C]$. History features X_t, Y_t are necessary, because Go is not fully observable solely from the current stones, as repetitions are forbidden; similarly, the colour feature C is necessary, because the komi is not observable.

The input features s_t are processed by a residual tower that consists of a single convolutional block followed by either 19 or 39 residual blocks[4].

The convolutional block applies the following modules:

(1) A convolution of 256 filters of kernel size 3×3 with stride 1
(2) Batch normalization[18]
(3) A rectifier nonlinearity

Each residual block applies the following modules sequentially to its input:

(1) A convolution of 256 filters of kernel size 3×3 with stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A convolution of 256 filters of kernel size 3×3 with stride 1
(5) Batch normalization
(6) A skip connection that adds the input to the block
(7) A rectifier nonlinearity

The output of the residual tower is passed into two separate 'heads' for computing the policy and value. The policy head applies the following modules:

(1) A convolution of 2 filters of kernel size 1×1 with stride 1
(2) Batch normalization
(3) A rectifier nonlinearity
(4) A fully connected linear layer that outputs a vector of size $19^2 + 1 = 362$, corresponding to logit probabilities for all intersections and the pass move

The value head applies the following modules:

(1) A convolution of 1 filter of kernel size 1×1 with stride 1
(2) Batch normalization
(3) A rectifier nonlinearity

(4) A fully connected linear layer to a hidden layer of size 256

(5) A rectifier nonlinearity

(6) A fully connected linear layer to a scalar

(7) A tanh nonlinearity outputting a scalar in the range $[-1, 1]$

The overall network depth, in the 20- or 40-block network, is 39 or 79 parameterized layers, respectively, for the residual tower, plus an additional 2 layers for the policy head and 3 layers for the value head.

We note that a different variant of residual networks was simultaneously applied to computer Go[33] and achieved an amateur dan-level performance; however, this was restricted to a single-headed policy network trained solely by supervised learning.

Neural network architecture comparison. Figure 4 shows the results of a comparison between network architectures. Specifically, we compared four different neural networks:

(1) dual–res: the network contains a 20-block residual tower, as described above, followed by both a policy head and a value head. This is the architecture used in AlphaGo Zero.

(2) sep–res: the network contains two 20-block residual towers. The first tower is followed by a policy head and the second tower is followed by a value head.

(3) dual–conv: the network contains a non-residual tower of 12 convolutional blocks, followed by both a policy head and a value head.

(4) sep–conv: the network contains two non-residual towers of 12 convolutional blocks. The first tower is followed by a policy head and the second tower is followed by a value head. This is the architecture used in AlphaGo Lee.

Each network was trained on a fixed dataset containing the final 2 million games of self-play data generated by a previous run of AlphaGo Zero, using stochastic gradient descent with the annealing rate, momentum and regularization hyperparameters described for the supervised learning experiment; however, cross-entropy and MSE components were weighted equally, since more data was available.

Evaluation. We evaluated the relative strength of AlphaGo Zero (Figs 3a, 6) by measuring the Elo rating of each player. We estimate the probability that player a will defeat player b by a logistic function $P(a \text{ defeats } b) = \frac{1}{1 + \exp(c_{\text{elo}}(e(b) - e(a)))}$, and estimate the ratings $e(\cdot)$ by Bayesian logistic regression, computed by the BayesElo program[25] using the standard constant $c_{\text{elo}} = 1/400$.

Elo ratings were computed from the results of a 5 s per move tournament between AlphaGo Zero, AlphaGo Master, AlphaGo Lee and AlphaGo Fan. The raw neural network from AlphaGo Zero was also included in the tournament. The Elo ratings of AlphaGo Fan, Crazy Stone, Pachi and GnuGo were anchored to the tournament values from previous work[12], and correspond to the players reported in that work. The results of the matches of AlphaGo Fan against Fan Hui and AlphaGo Lee against Lee Sedol were also included to ground the scale to human references, as otherwise the Elo ratings of AlphaGo are unrealistically high due to self-play bias.

The Elo ratings in Figs 3a, 4a, 6a were computed from the results of evaluation games between each iteration of player α_{θ_i} during self-play training. Further evaluations were also performed against baseline players with Elo ratings anchored to the previously published values[12].

We measured the head-to-head performance of AlphaGo Zero against AlphaGo Lee, and the 40-block instance of AlphaGo Zero against AlphaGo Master, using the same player and match conditions that were used against Lee Sedol in Seoul, 2016. Each player received 2 h of thinking time plus 3 byoyomi periods of 60 s per move. All games were scored using Chinese rules with a *komi* of 7.5 points.

Data availability. The datasets used for validation and testing are the GoKifu dataset (available from http://gokifu.com/) and the KGS dataset (available from https://u-go.net/gamerecords/).

35. Barto, A. G. & Duff, M. Monte Carlo matrix inversion and reinforcement learning. *Adv. Neural Inf. Process. Syst.* **6,** 687–694 (1994).
36. Singh, S. P. & Sutton, R. S. Reinforcement learning with replacing eligibility traces. *Mach. Learn.* **22,** 123–158 (1996).
37. Lagoudakis, M. G. & Parr, R. Reinforcement learning as classification: leveraging modern classifiers. In *Proc. 20th Int. Conf. Mach. Learn.* 424–431 (2003).
38. Scherrer, B., Ghavamzadeh, M., Gabillon, V., Lesner, B. & Geist, M. Approximate modified policy iteration and its application to the game of Tetris. *J. Mach. Learn. Res.* **16,** 1629–1676 (2015).
39. Littman, M. L. Markov games as a framework for multi-agent reinforcement learning. In *Proc. 11th Int. Conf. Mach. Learn.* 157–163 (1994).
40. Enzenberger, M. The integration of a priori knowledge into a Go playing neural network. http://www.cgl.ucsf.edu/go/Programs/neurogo-html/neurogo.html (1996).
41. Enzenberger, M. in *Advances in Computer Games* (eds Van Den Herik, H. J., Iida, H. & Heinz, E. A.) 97–108 (2003).
42. Sutton, R. Learning to predict by the method of temporal differences. *Mach. Learn.* **3,** 9–44 (1988).
43. Schraudolph, N. N., Dayan, P. & Sejnowski, T. J. Temporal difference learning of position evaluation in the game of Go. *Adv. Neural Inf. Process. Syst.* **6,** 817–824 (1994).
44. Silver, D., Sutton, R. & Müller, M. Temporal-difference search in computer Go. *Mach. Learn.* **87,** 183–219 (2012).
45. Silver, D. *Reinforcement Learning and Simulation-Based Search in Computer Go.* PhD thesis, Univ. Alberta, Edmonton, Canada (2009).
46. Gelly, S. & Silver, D. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artif. Intell.* **175,** 1856–1875 (2011).
47. Coulom, R. Computing Elo ratings of move patterns in the game of Go. *Int. Comput. Games Assoc. J.* **30,** 198–208 (2007).
48. Gelly, S., Wang, Y., Munos, R. & Teytaud, O. Modification of UCT with patterns in Monte-Carlo Go. Report No. 6062 (INRIA, 2006).
49. Baxter, J., Tridgell, A. & Weaver, L. Learning to play chess using temporal differences. *Mach. Learn.* **40,** 243–263 (2000).
50. Veness, J., Silver, D., Blair, A. & Uther, W. Bootstrapping from game tree search. In *Adv. Neural Inf. Process. Syst.* 1937–1945 (2009).
51. Lai, M. *Giraffe: Using Deep Reinforcement Learning to Play Chess.* MSc thesis, Imperial College London (2015).
52. Schaeffer, J., Hlynka, M. & Jussila, V. Temporal difference learning applied to a high-performance game-playing program. In *Proc. 17th Int. Jt Conf. Artif. Intell.* Vol. 1 529–534 (2001).
53. Tesauro, G. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.* **6,** 215–219 (1994).
54. Buro, M. From simple features to sophisticated evaluation functions. In *Proc. 1st Int. Conf. Comput. Games* 126–145 (1999).
55. Sheppard, B. World-championship-caliber Scrabble. *Artif. Intell.* **134,** 241–275 (2002).
56. Moravčík, M. *et al.* DeepStack: expert-level artificial intelligence in heads-up no-limit poker. *Science* **356,** 508–513 (2017).
57. Tesauro, G & Galperin, G. On-line policy improvement using Monte-Carlo search. In *Adv. Neural Inf. Process. Syst.* 1068–1074 (1996).
58. Tesauro, G. Neurogammon: a neural-network backgammon program. In *Proc. Int. Jt Conf. Neural Netw.* Vol. 3, 33–39 (1990).
59. Samuel, A. L. Some studies in machine learning using the game of checkers II - recent progress. *IBM J. Res. Develop.* **11,** 601–617 (1967).
60. Kober, J., Bagnell, J. A. & Peters, J. Reinforcement learning in robotics: a survey. *Int. J. Robot. Res.* **32,** 1238–1274 (2013).
61. Zhang, W. & Dietterich, T. G. A reinforcement learning approach to job-shop scheduling. In *Proc. 14th Int. Jt Conf. Artif. Intell.* 1114–1120 (1995).
62. Cazenave, T., Balbo, F. & Pinson, S. Using a Monte-Carlo approach for bus regulation. In *Int. IEEE Conf. Intell. Transport. Syst.* 1–6 (2009).
63. Evans, R. & Gao, J. Deepmind AI reduces Google data centre cooling bill by 40%. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/ (2016).
64. Abe, N. *et al.* Empirical comparison of various reinforcement learning strategies for sequential targeted marketing. In *IEEE Int. Conf. Data Mining* 3–10 (2002).
65. Silver, D., Newnham, L., Barker, D., Weller, S. & McFall, J. Concurrent reinforcement learning from customer interactions. In *Proc. 30th Int. Conf. Mach. Learn.* Vol. 28 924–932 (2013).
66. Tromp, J. Tromp–Taylor rules. http://tromp.github.io/go.html (1995).
67. Müller, M. Computer Go. *Artif. Intell.* **134,** 145–179 (2002).
68. Shahriari, B., Swersky, K., Wang, Z., Adams, R. P. & de Freitas, N. Taking the human out of the loop: a review of Bayesian optimization. *Proc. IEEE* **104,** 148–175 (2016).
69. Segal, R. B. On the scalability of parallel UCT. *Comput. Games* **6515,** 36–47 (2011).

Game 1, B: AG Lee, W: AG Zero, Result: W+R
Game 2, B: AG Lee, W: AG Zero, Result: W+R
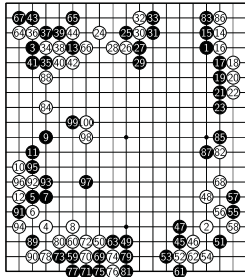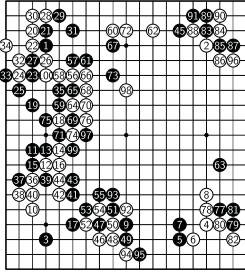Game 3, B: AG Lee, W: AG Zero, Result: W+R
Game 4, B: AG Lee, W: AG Zero, Result: W+0.50

Game 5, B: AG Lee, W: AG Zero, Result: W+R
Game 6, B: AG Lee, W: AG Zero, Result: W+0.50
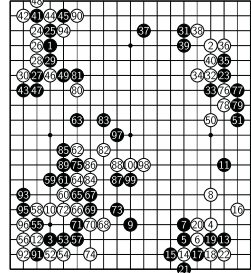Game 7, B: AG Lee, W: AG Zero, Result: W+R
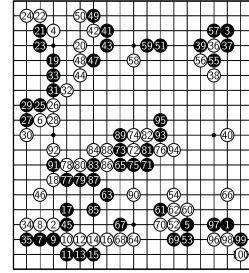Game 8, B: AG Lee, W: AG Zero, Result: W+R
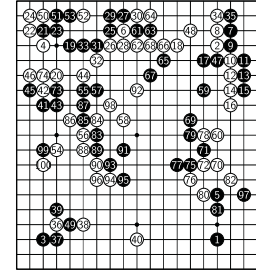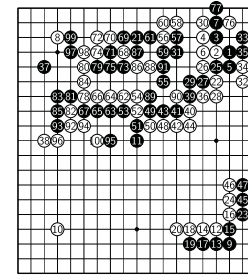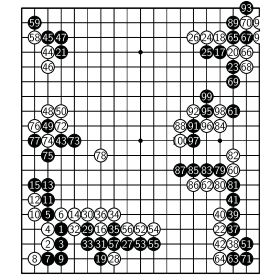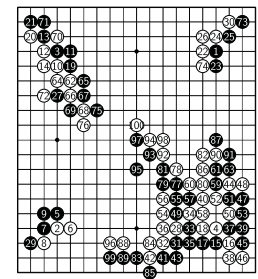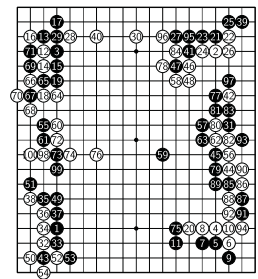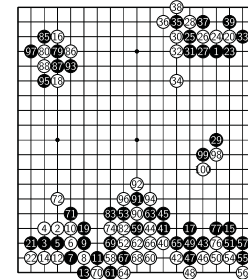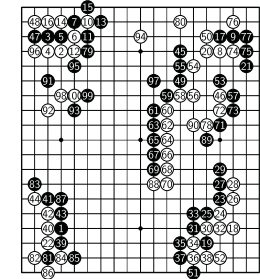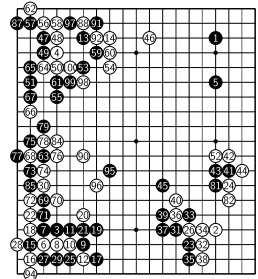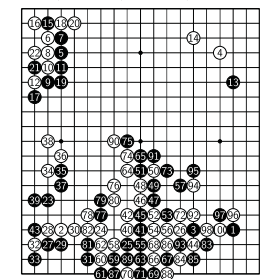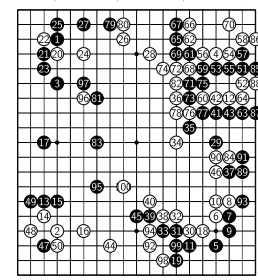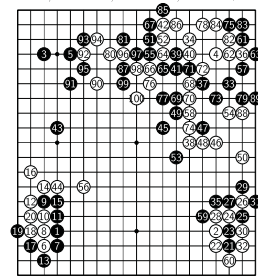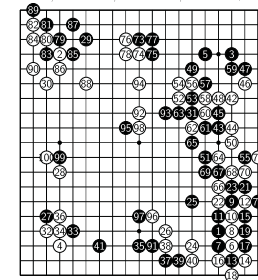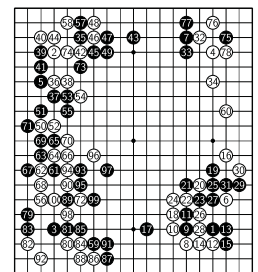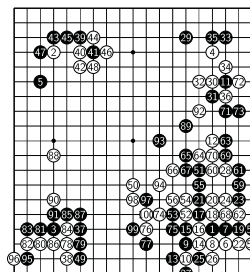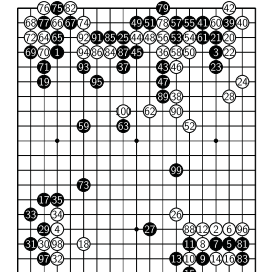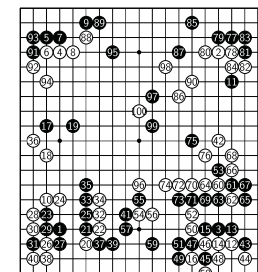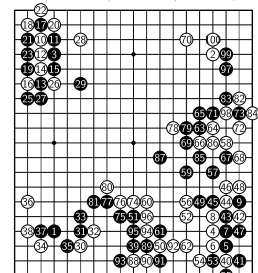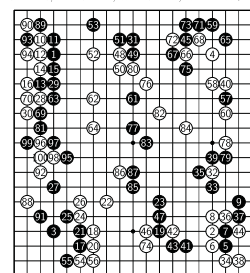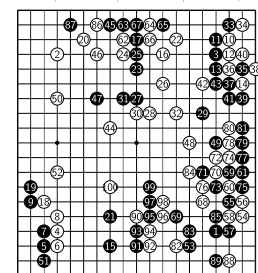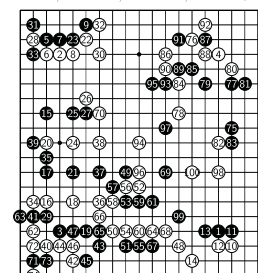
Game 9, B: AG Lee, W: AG Zero, Result: W+R
Game 10, B: AG Lee, W: AG Zero, Result: W+R
Game 11, B: AG Zero, W: AG Lee, Result: B+R
Game 12, B: AG Zero, W: AG Lee, Result: B+1.50

Game 13, B: AG Zero, W: AG Lee, Result: B+R
Game 14, B: AG Zero, W: AG Lee, Result: B+R
Game 15, B: AG Zero, W: AG Lee, Result: B+R
Game 16, B: AG Zero, W: AG Lee, Result: B+R
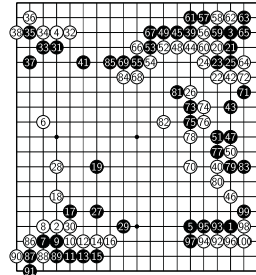
Game 17, B: AG Zero, W: AG Lee, Result: B+R
Game 18, B: AG Zero, W: AG Lee, Result: B+R
Game 19, B: AG Zero, W: AG Lee, Result: B+1.50
Game 20, B: AG Zero, W: AG Lee, Result: B+R
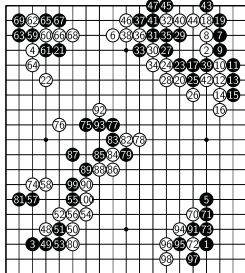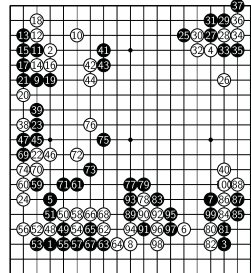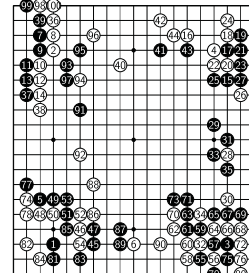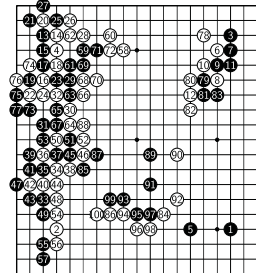
Extended Data Figure 1 | Tournament games between AlphaGo Zero (20 blocks, 3 days) versus AlphaGo Lee using 2 h time controls. One hundred moves of the first 20 games are shown; full games are provided in the Supplementary Information.

Extended Data Figure 2 | Frequency of occurence over time during training, for each *joseki* from Fig. 5a (corner sequences common in professional play that were discovered by AlphaGo Zero). The corresponding *joseki* are shown on the right.
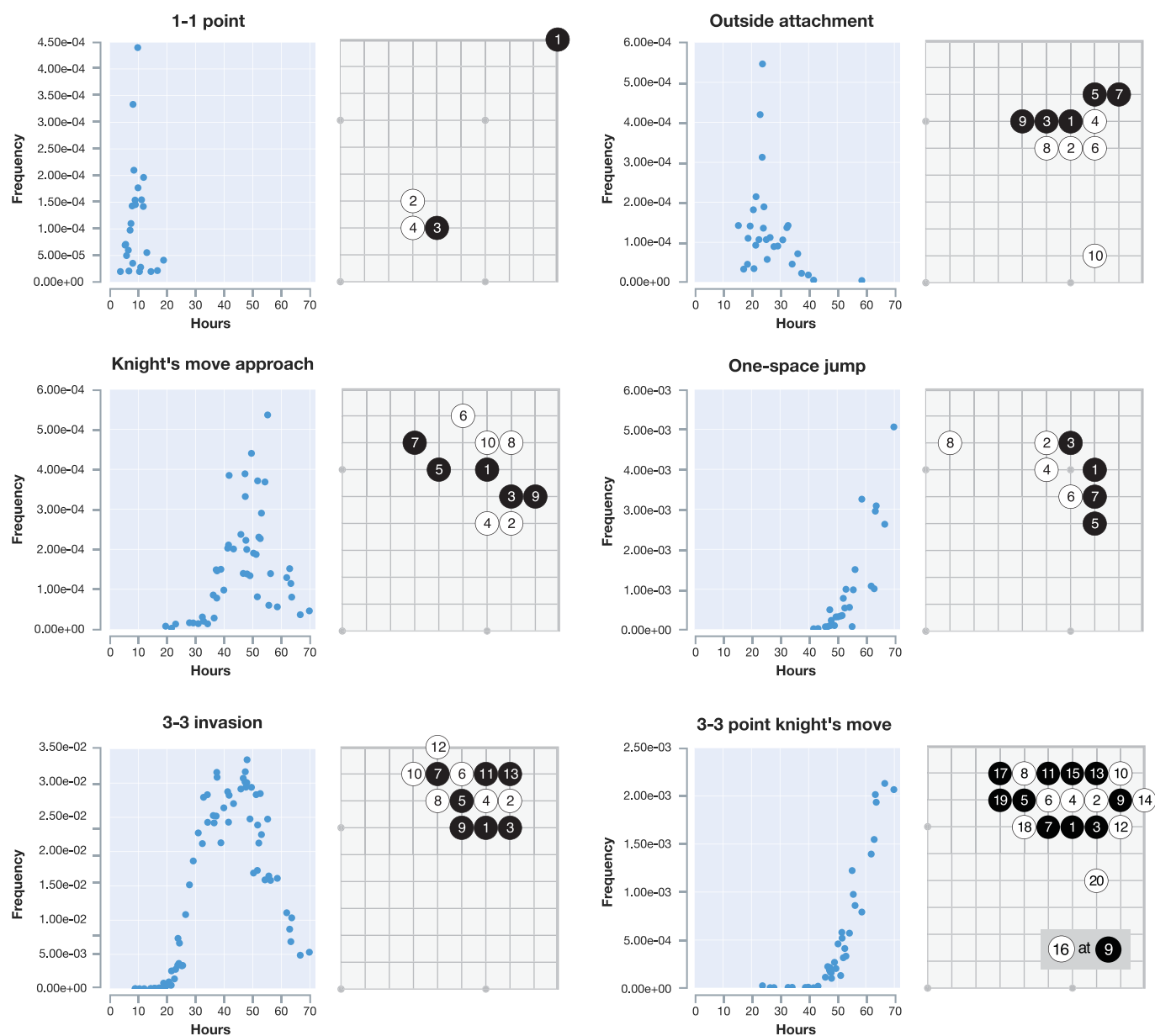
Extended Data Figure 3 | Frequency of occurence over time during training, for each *joseki* from Fig. 5b (corner sequences that AlphaGo Zero favoured for at least one iteration), and one additional variation. The corresponding *joseki* are shown on the right.
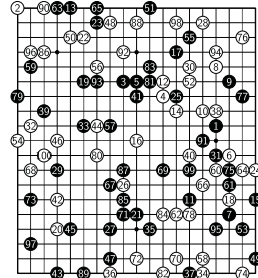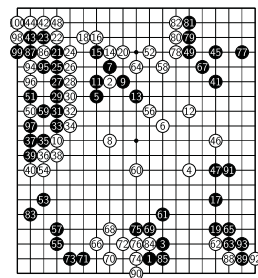
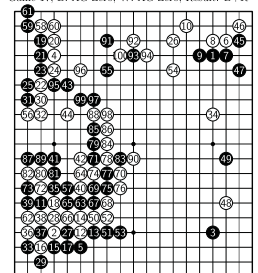Game 1, B: AG Zero, W: AG Zero, Result: B+R
Game 2, B: AG Zero, W: AG Zero, Result: B+R
Game 3, B: AG Zero, W: AG Zero, Result: W+R
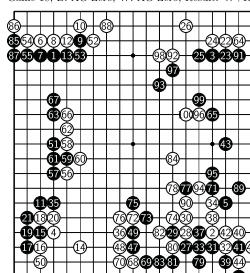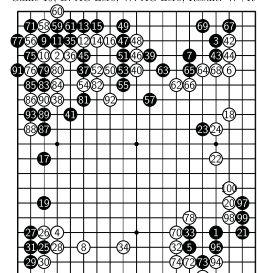Game 4, B: AG Zero, W: AG Zero, Result: B+R

Game 5, B: AG Zero, W: AG Zero, Result: B+R
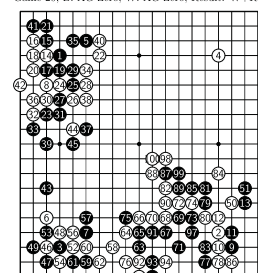Game 6, B: AG Zero, W: AG Zero, Result: W+R
Game 7, B: AG Zero, W: AG Zero, Result: B+R
Game 8, B: AG Zero, W: AG Zero, Result: B+R
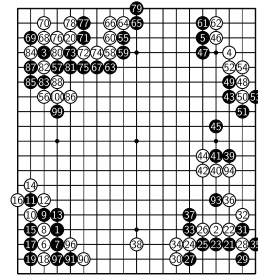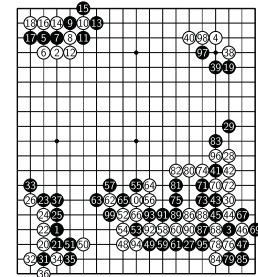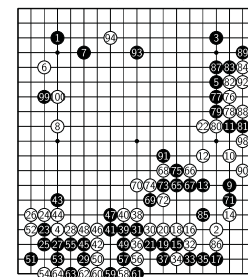
Game 9, B: AG Zero, W: AG Zero, Result: W+R
Game 10, B: AG Zero, W: AG Zero, Result: W+R
Game 11, B: AG Zero, W: AG Zero, Result: W+R
Game 12, B: AG Zero, W: AG Zero, Result: W+R
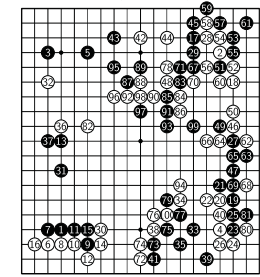
Game 13, B: AG Zero, W: AG Zero, Result: W+R
Game 14, B: AG Zero, W: AG Zero, Result: W+R
Game 15, B: AG Zero, W: AG Zero, Result: W+R
Game 16, B: AG Zero, W: AG Zero, Result: W+R
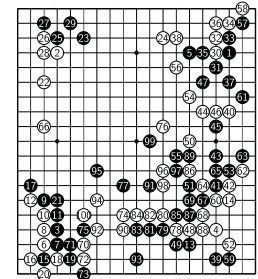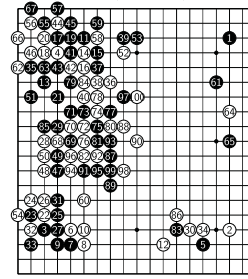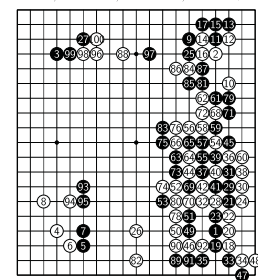
Game 17, B: AG Zero, W: AG Zero, Result: W+R
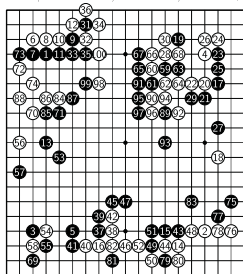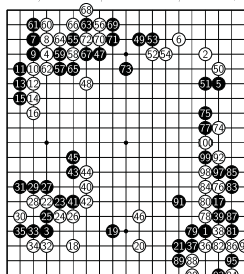Game 18, B: AG Zero, W: AG Zero, Result: B+R
Game 19, B: AG Zero, W: AG Zero, Result: B+R
Game 20, B: AG Zero, W: AG Zero, Result: W+R
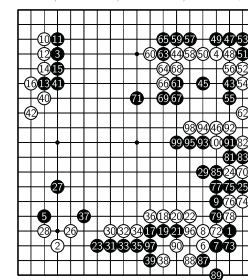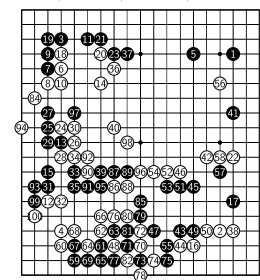
Extended Data Figure 4 | AlphaGo Zero (20 blocks) self-play games. The 3-day training run was subdivided into 20 periods. The best player from each period (as selected by the evaluator) played a single game against itself, with 2 h time controls. One hundred moves are shown for each game; full games are provided in the Supplementary Information.
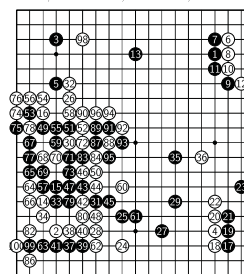
Game 1, B: AG Zero, W: AG Zero, Result: B+R
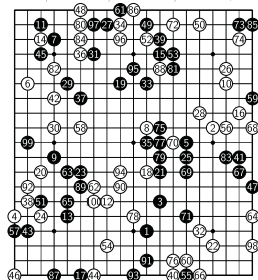
Game 2, B: AG Zero, W: AG Zero, Result: W+R
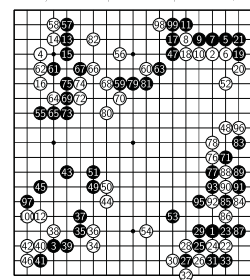
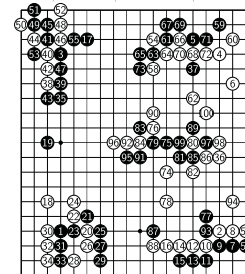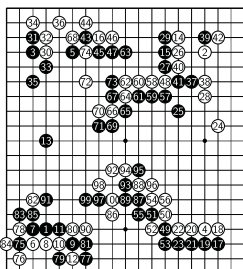Game 3, B: AG Zero, W: AG Zero, Result: B+R

Game 4, B: AG Zero, W: AG Zero, Result: W+R

94 at 85

20 at 13

Game 5, B: AG Zero, W: AG Zero, Result: W+R

Game 6, B: AG Zero, W: AG Zero, Result: W+R

Game 7, B: AG Zero, W: AG Zero, Result: W+R

Game 8, B: AG Zero, W: AG Zero, Result: B+R

80 at 56

Game 9, B: AG Zero, W: AG Zero, Result: W+R

Game 10, B: AG Zero, W: AG Zero, Result: W+R

Game 11, B: AG Zero, W: AG Zero, Result: B+R

Game 12, B: AG Zero, W: AG Zero, Result: W+R

24 at 17
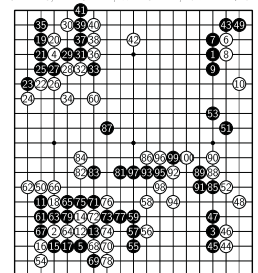
Game 13, B: AG Zero, W: AG Zero, Result: B+R

Game 14, B: AG Zero, W: AG Zero, Result: W+R
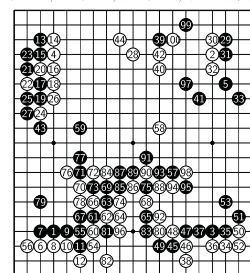
Game 15, B: AG Zero, W: AG Zero, Result: W+R

Game 16, B: AG Zero, W: AG Zero, Result: W+R

80 at 13

62 at 55
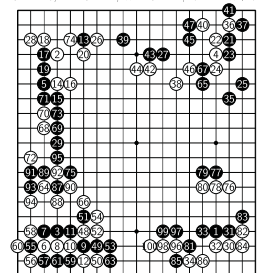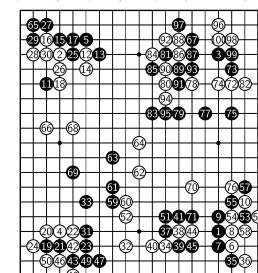
Game 17, B: AG Zero, W: AG Zero, Result: B+R

Game 18, B: AG Zero, W: AG Zero, Result: W+R

Game 19, B: AG Zero, W: AG Zero, Result: W+R

Game 20, B: AG Zero, W: AG Zero, Result: W+R

55 at 46

Extended Data Figure 5 | AlphaGo Zero (40 blocks) self-play games. The 40-day training run was subdivided into 20 periods. The best player from each period (as selected by the evaluator) played a single game against itself, with 2 h time controls. One hundred moves are shown for each game; full games are provided in the Supplementary Information.

Game 1, B: AG Master, W: AG Zero, Result: W+R

Game 2, B: AG Zero, W: AG Master, Result: B+R

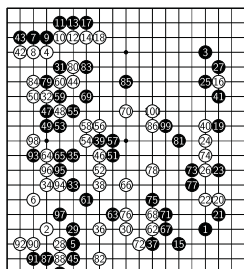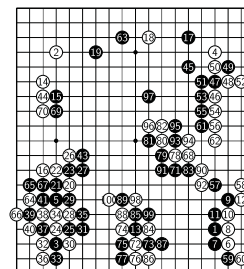Game 3, B: AG Master, W: AG Zero, Result: W+R
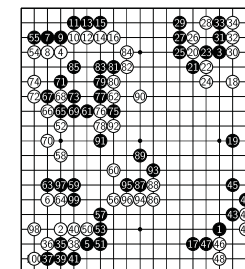
Game 4, B: AG Zero, W: AG Master, Result: B+R

⑧ at ㊳ ⑧ at ⑦ ㊼ at ㊼

Game 5, B: AG Master, W: AG Zero, Result: W+R

Game 6, B: AG Zero, W: AG Master, Result: B+R

Game 7, B: AG Master, W: AG Zero, Result: W+R

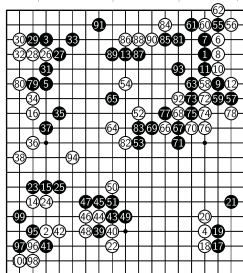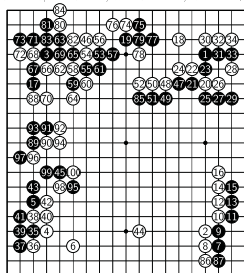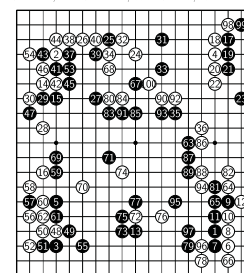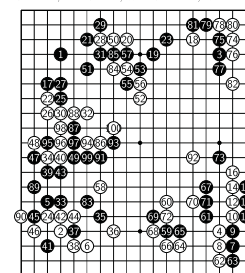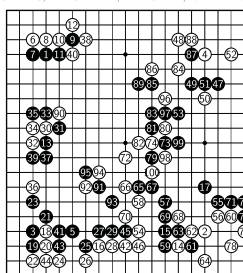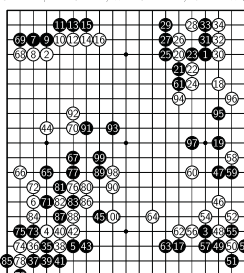Game 8, B: AG Zero, W: AG Master, Result: B+R

㊷ at ㊲

Game 9, B: AG Master, W: AG Zero, Result: W+R

Game 10, B: AG Zero, W: AG Master, Result: B+R

Game 11, B: AG Master, W: AG Zero, Result: B+R

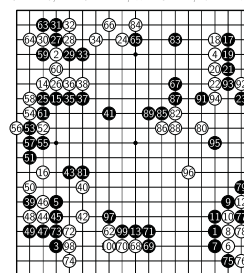Game 12, B: AG Zero, W: AG Master, Result: B+R
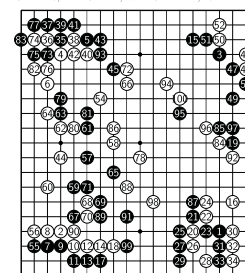
Game 13, B: AG Master, W: AG Zero, Result: W+R
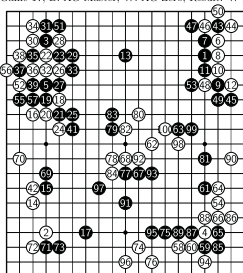
Game 14, B: AG Zero, W: AG Master, Result: W+R

Game 15, B: AG Master, W: AG Zero, Result: W+R

Game 16, B: AG Zero, W: AG Master, Result: W+R

Game 17, B: AG Master, W: AG Zero, Result: W+R

Game 18, B: AG Zero, W: AG Master, Result: B+R

Game 19, B: AG Master, W: AG Zero, Result: W+R

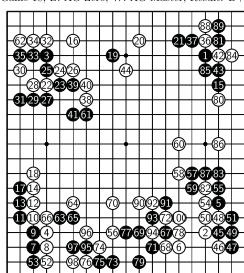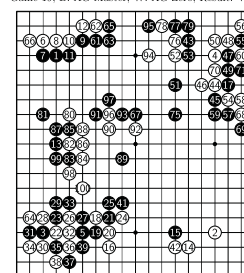Game 20, B: AG Zero, W: AG Master, Result: B+R

㊵ at ㉟

㊾ at ㊿

㊵ at ㉟

⑦ at ⑦ ⑥ at ⑧ ㊿ at ㊿ ⑨ at ㊷ ⑦ at ㊿
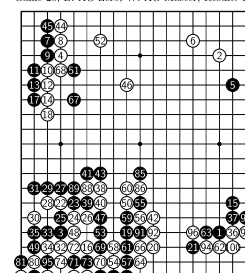
Extended Data Figure 6 | AlphaGo Zero (40 blocks, 40 days) versus AlphaGo Master tournament games using 2 h time controls. One hundred moves of the first 20 games are shown; full games are provided in the Supplementary Information.

Extended Data Table 1 | Move prediction accuracy

	KGS train	*KGS* test	*GoKifu* validation
Supervised learning (20 block)	62.0	60.4	54.3
Supervised learning (12 layer [12])	59.1	55.9	-
Reinforcement learning (20 block)	-	-	49.0
Reinforcement learning (40 block)	-	-	51.3

Percentage accuracies of move prediction for neural networks trained by reinforcement learning (that is, AlphaGo Zero) or supervised learning. For supervised learning, the network was trained for 3 days on KGS data (amateur games); comparative results are also shown from ref. 12. For reinforcement learning, the 20-block network was trained for 3 days and the 40-block network was trained for 40 days. Networks were also evaluated on a validation set based on professional games from the GoKifu dataset.

Extended Data Table 2 | Game outcome prediction error

	KGS train	*KGS* test	*GoKifu* validation
Supervised learning (20 block)	0.177	0.185	0.207
Supervised learning (12 layer [12])	0.19	0.37	-
Reinforcement learning (20 block)	-	-	0.177
Reinforcement learning (40 block)	-	-	0.180

Mean squared error on game outcome predictions for neural networks trained by reinforcement learning (that is, AlphaGo Zero) or supervised learning. For supervised learning, the network was trained for 3 days on KGS data (amateur games); comparative results are also shown from ref. 12. For reinforcement learning, the 20 block network was trained for 3 days and the 40 block network was trained for 40 days. Networks were also evaluated on a validation set based on professional games from the GoKifu dataset.

Extended Data Table 3 | Learning rate schedule

Thousands of steps	Reinforcement learning	Supervised learning
0–200	10^{-2}	10^{-1}
200–400	10^{-2}	10^{-2}
400–600	10^{-3}	10^{-3}
600–700	10^{-4}	10^{-4}
700–800	10^{-4}	10^{-5}
>800	10^{-4}	-

Learning rate used during reinforcement learning and supervised learning experiments, measured in thousands of steps (mini-batch updates).