AlphaZero Chess: How It Works, What Sets It Apart, and What It Can Tell Us

A deep dive into the most revolutionary phenomenon in computer chess in the 21st century

Maxim Khovanskiy
Published in Towards Data Science
15 min read · May 5, 2022

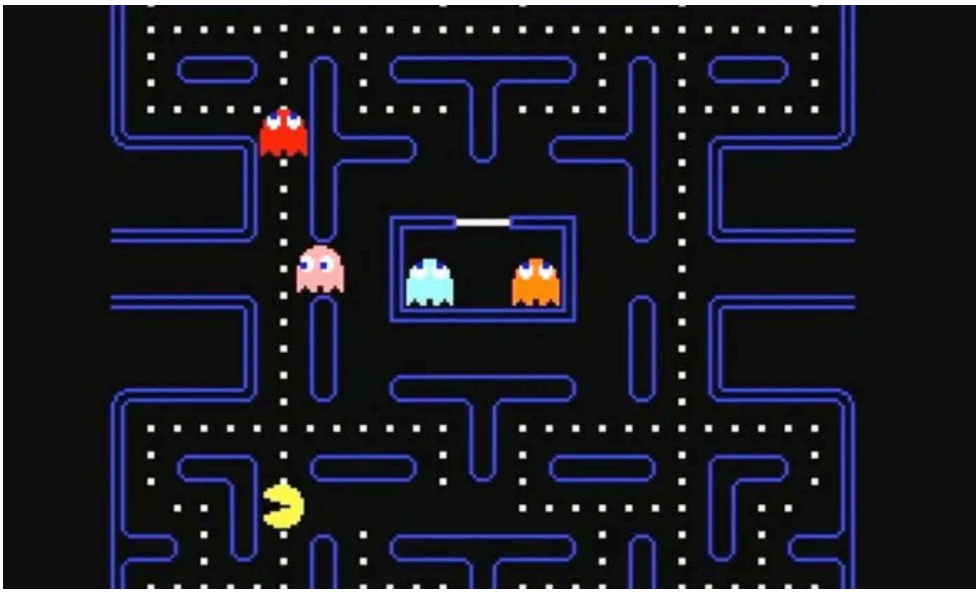
By Fotomek. Source: Istockphoto

Introduction

To those of you who have an interest in chess — or who have been monitoring recent developments in artificial intelligence — the name "AlphaZero" will be instantly recognisable; its victory over the then-leading chess engine in the world, Stockfish, had revolutionised the way that chess is played by both computers and, indeed, humans.

However, if you aren't a chess aficionado or have missed the news a couple of years ago, you might be wondering what exactly this AlphaZero really is, and what makes it worth writing an entire blog post about. For you, I will explain.

In short, AlphaZero is a game-playing program that, through a combination of self-play and neural network reinforcement learning (more on that later), is able to learn to play games such as chess and Go from scratch — that is, after being fed nothing more than the rules of said games. In fact, a newer derivative of AlphaZero, called MuZero, isn't limited to only board games such as chess, but can also learn to play a range of simple video games from the Atari collection. Both AlphaZero and MuZero were designed by DeepMind, a subsidiary of Alphabet Inc., the parent company of Google.



The Atari classic Pac-Man is one of the games that MuZero learnt to play. Source: Screenshot, Midway Games.

Okay, so we have something known as a General Game Playing (GGP) artificial intelligence on our hands. We have seen those before; for one, Stanford University has been running an annual GGP contest for over a decade, with entries such as WoodStock in 2016 performing particularly well. So what makes AlphaZero so special? It's damn good at what it does. So good, in fact, that not even human professionals at the games that AlphaZero is tasked with learning or — get this — even state-of-the-art, purpose-built computer programs developed specifically to play only one of these games are able to match its performance*. It's a jack of all trades and a master of *all*. As a case study into AlphaZero's GGP ability, I will be focusing on chess, being an avid chess player myself.

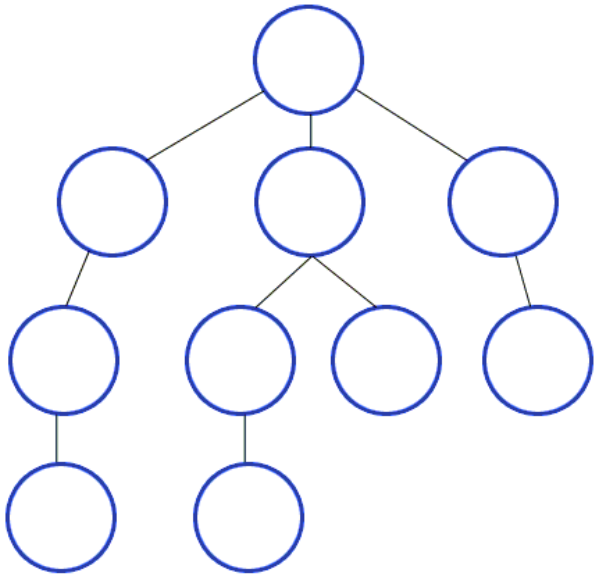
The premise behind this blog post is that I, as a master-strength chess player, am primed to provide some external insight into the field of artificial intelligence from a chess-based perspective. To this end, I will first detail how traditional engines and AlphaZero work, before explaining what that can tell us about the two types of engine's playing styles and exploring the implications of that on the broader field of artificial intelligence.

**In chess, as of 2022, newer engines are generally agreed to have surpassed AlphaZero's playing strength; however, at the time, its performance was unprecedented, although there is some controversy around the matter regarding the improved hardware that AlphaZero was run on.*

Traditional engines

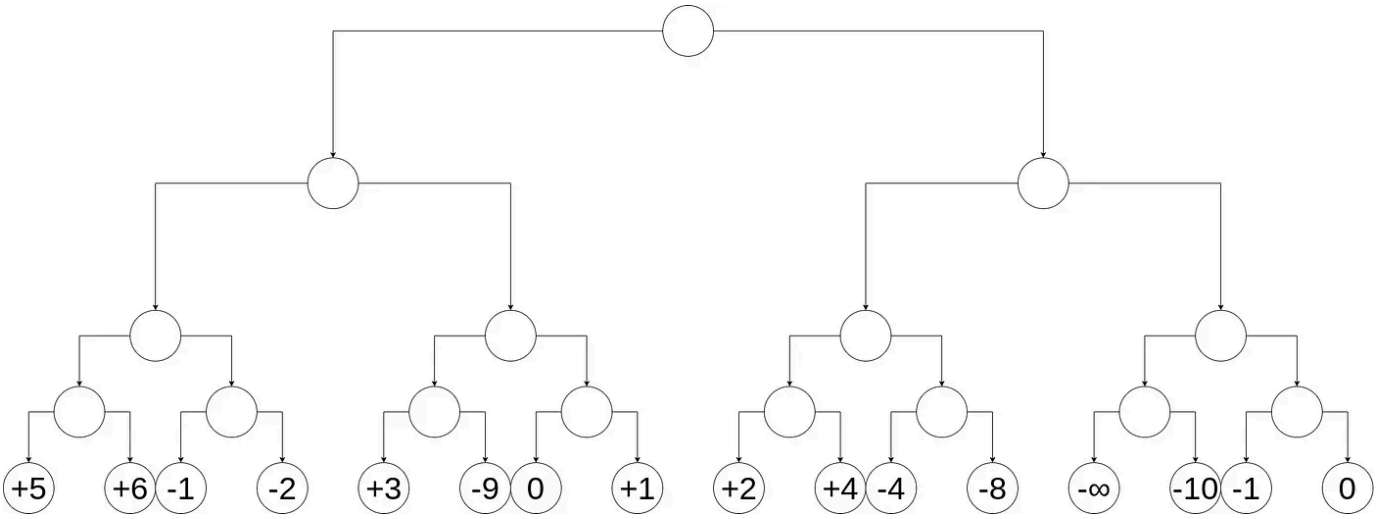
Since the field of computer chess emerged in the 1970s and until AlphaZero's inception, the basic architecture of almost all chess engines has been virtually the same.

On every move, a depth-first search (DFS) of the game-tree, up until a specified depth, is performed: that is, every single legal sequence of a chosen number of moves from the current position is computed. Since no legal moves are left out, this class of search algorithms is known as *brute-force search*.



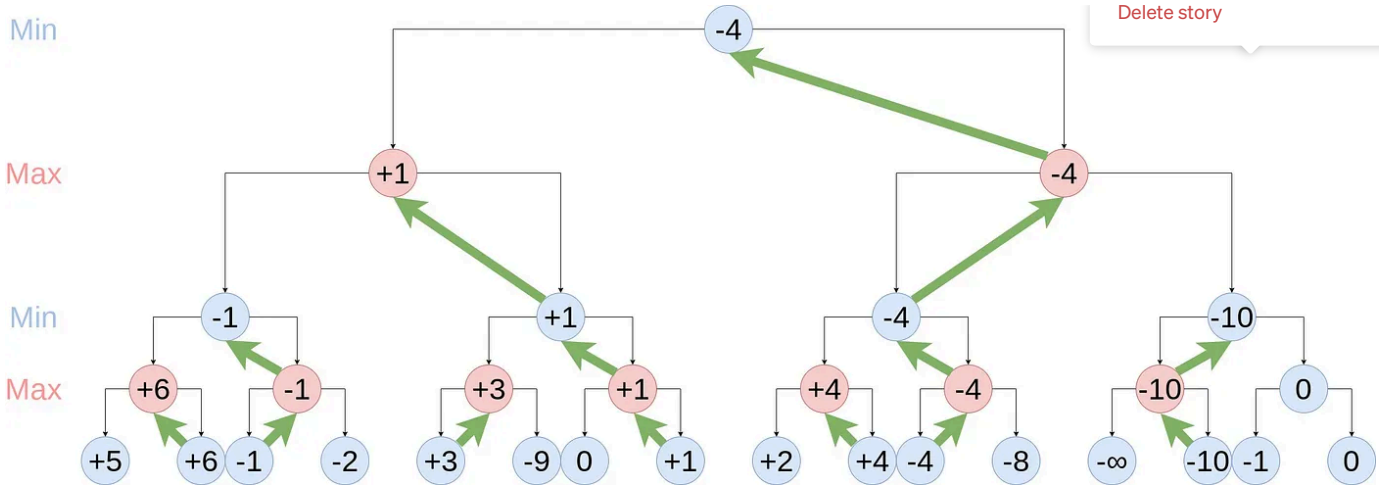
An animation of the DFS being performed on a short game-tree. Source: Wikimedia Commons.

Then, the terminal nodes of the tree — or the last move in each of the computed sequences — are assigned a numeric value representing, roughly speaking, how the good the position at that node is for White (the player with the white pieces); this value is known as the *static evaluation*. It is calculated using a number of heuristics formulated by a community of strong chess players, and is intended to represent the factors that a human considers when evaluating any given position. These heuristics include obvious metrics such as the material count (the raw total value of the pieces of both sides) and piece-square table readings (piece-square tables detail how good, generally speaking, each of the pieces is on each of the 64 squares), but also more subtle considerations such as king safety and pawn structure. The full list of heuristics used by Stockfish 14*, the leading chess engine as of 2022, can be found here.



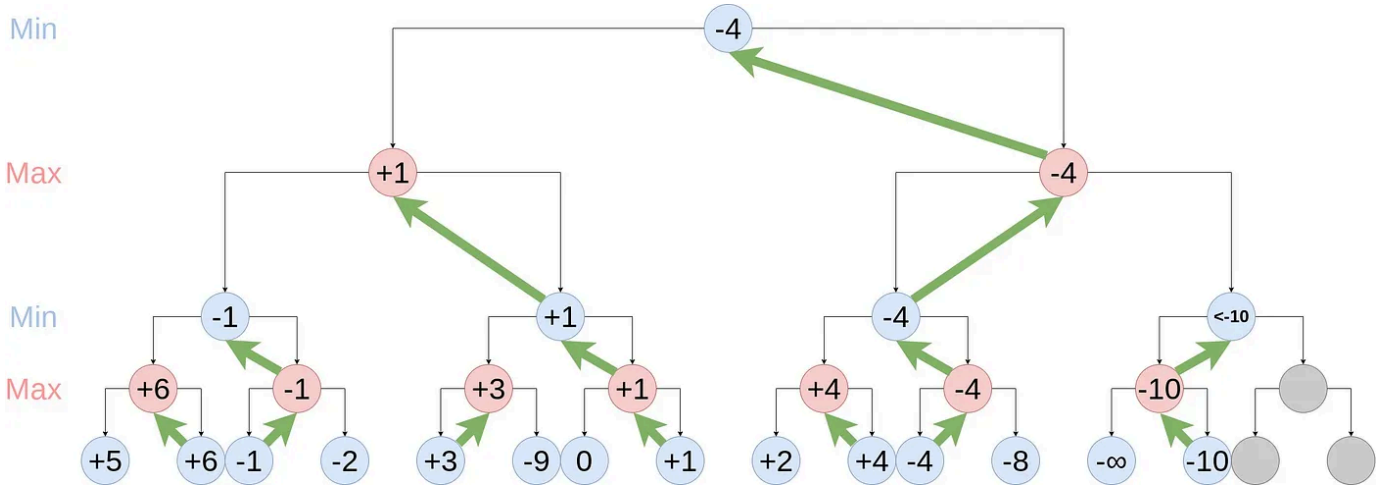
Static evaluations at terminal nodes of a miniature game-tree. Source :Author.

Consequently, something called the minimax algorithm is run from the terminal nodes up until th which represents the current position: for each node which has not been previously evaluated, the as the *maximum* of the set of evaluations of that node's children (or replies to the move represente White's turn to move at that node; and it is computed as the *minimum* of that set if it is Black's turn This ensures that the evaluation process entails optimal play, in the heuristics' estimation, from bo crude evaluation of the current position.



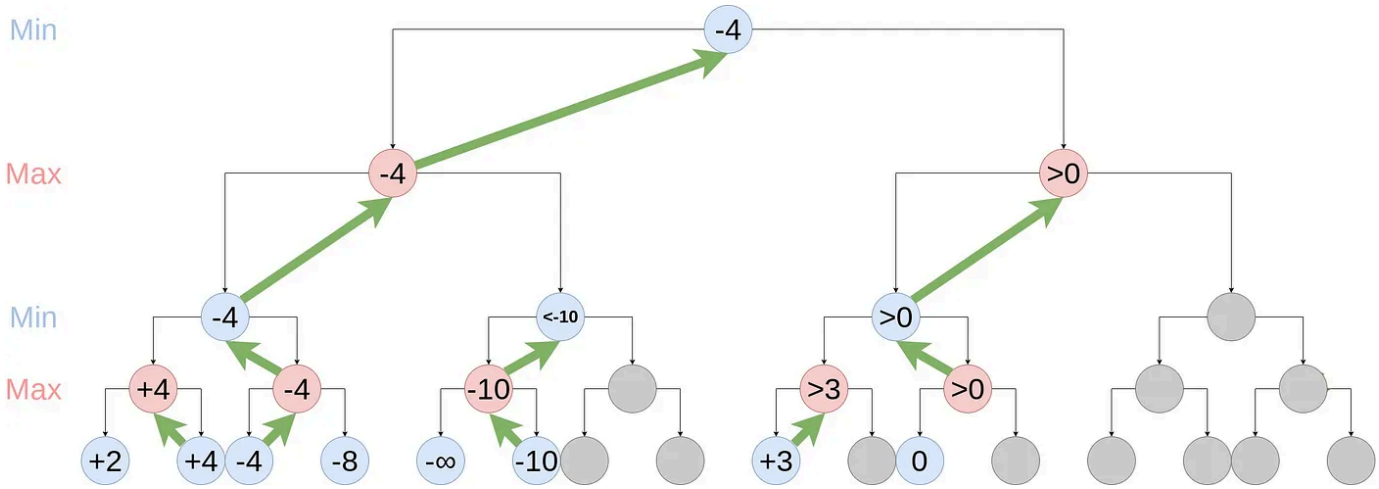
Minimax algorithm run on the game-tree from the previous image. In this case, the evaluation is minimised on the engine's turn as it is playing as Black. The evaluation of the position is -4. Source: Author.

Finally, a pruning algorithm called alpha-beta pruning is applied. The algorithm works as follows: two values — alpha and beta — are stored at each node which represent the least favourable possible evaluation for White and Black, respectively, of any of the ancestor nodes. Intuitively, these values represent the thresholds beyond which the nodes can't have any effect on one of the ancestors due to being worse than some previously evaluated move(s). Hence, since it's worthless for White to consider moves which allow Black as much or more than moves that White had already calculated, and vice versa, nodes at which alpha is greater than or equal to beta are skipped, or *pruned*. This clever trick saves a lot computing power without compromising accuracy.



Alpha-Beta pruning algorithm applied to the game-tree from earlier. Source: Author.

Note that, in the image above, only 3 out of 31 nodes were pruned. That's because many moves which ended up being suboptimal, such as Black's very first calculated move (+1), were evaluated before the optimal moves, such as Black's alternative first move (-4). If we change up the order in which the moves are evaluated so that the more promising ones are evaluated first more often, we can drastically increase the number of nodes pruned:



Source: Author.

Therefore, the very last core component of traditional engines is another set of heuristics which determines the order in which the moves are calculated. A common heuristic of this class is the prioritisation of captures, for example, as captures are the only direct way of taking opponent's pieces off the board and thus weakening their forces.

There is a large number of additions and refinements that can be made to this simplistic model, such as a variety of additional pruning algorithms, an opening book, endgame tablebases, and much, much more. However, this is the backbone that saw computers defeat amateurs for the first time, best the most prolific of masters and grandmasters, and finally dethrone humans as the best chess-playing entity once and for all in a famous 1997 Deep Blue vs Kasparov match. It continues, with some modifications, to be used in the strongest chess engines of today.

Combining everything described in this section, we can produce some Python code for what a traditional engine might look like. Notice how the code merges all the outlined steps together in one streamlined process:

- *Stockfish 14 deviates from the traditional architecture by incorporating a simple neural network into its static evaluation function in a method called NNUE; however, heuristics are still in use to supply the neural network with training data consisting of a set of positions and their respective heuristic evaluations.*

AlphaZero

So, in the last section, we saw how chess engines had been, at their core, practically the same for decades, with improvements, significant though they were, being largely in degree — not kind. That all changed in 2018 when DeepMind unveiled the inner workings of AlphaZero, which had previously shocked the chess world with an impressive showing against the strongest engine at the time. Soon after the 2018 paper, an open-source AlphaZero clone project called LeelaChessZero was initiated, and the engine swiftly rose to the summit of computer chess, becoming the strongest engine in the world in 2019.

So what exactly is so different about AlphaZero and LeelaChessZero compared to traditional engines? Let's take a deep dive into AlphaZero's brain to find out.

At first, a convolutional neural network — the class of neural network that performs particularly well in visual classification tasks — is initialised to return 2 randomly generated outputs: v(s), which will later estimate the win probability of the player

from position s, and **p(s)**, a vector which will later estimate how promising each of the legal moves from position s is. The network will later be trained to take in position s and predict v(s) and **p(s)** from it. Call this initialised neural network *nnet0*.

While nnet0 isn't great, we can still use it to build a game-playing agent, with the intention of using data from the games played by this agent to train another neural network, *nnet1*, and use that network to construct a better game-playing agent. The ultimate strategy is to repeat this process enough times for our final neural network to have gathered enough data to render our final game-playing agent highly competent.

The game-playing agent design that AlphaZero uses is based on a version of Monte Carlo Tree Search, which, instead of exploring every possible branch as deeply as the hardware allows like DFS does, explores a small number of branches but until the very end of the game. The basic premise behind MCTS is that it does not require exploring as many branches as DFS, and the extra computational power can be redirected elsewhere.

In AlphaZero's case, the exact version of MCTS that it uses is as follows:

- On every move *s*, a number of *simulations* are run. Think of simulations as mini-games the agent plays against itself to see which moves result in the most favourable outcomes.
- Each simulation entails executing the following algorithm until the first position that has not been visited by any of the previous simulations is encountered:
- Out of all the legal moves in the current position, select the *most productive one*. The most productive move is evaluated as the one that strikes the perfect balance between *exploration* and *exploitation*: that is, a move that has a good enough track record to be worth exploring (exploitation), but on the other hand has not already been explored too much to yield useful information (exploration). More rigorously, the most productive move *a* is the one that maximises the following expression, called the *upper confidence bound*:

$$Q(s,a) + c \cdot P(s,a) \cdot \frac{\sqrt{\sum_b [N(s,b)]}}{1 + N(s,a)}$$

Q(s,a) is the average evaluation (more details later) of move *a* across all the simulations which explored it, and represents the exploitation score, or how objectively good move *a* is estimated to be. *c* is a hyperparameter constant representing the degree of preference for exploration compared to exploitation. *P(s,a)* indicates how promising move *a* is, or the extent to which it is worth exploring (again, more details on this later). Finally, *N(s,a)* is the number of times that move *a* has been visited from position *s*. Note that the final term in the equation gets smaller as *N(s,a)* gets larger, so moves which have already been visited too many times will reduce the value of the term; one can hence think of it as representing the exploration score.

- Play the selected move and run the algorithm again until either the end of the game is reached — in which case the evaluation of move *a* for the simulation is recorded as 1 if the player wins the game and -1 if he doesn't, and the exploitation score *Q(s,a)* is recomputed — or, as mentioned above, a position *r* that has not been visited by any of the previous simulations is encountered. In that case, record the evaluation of *r* for the simulation as *v(r)*, and set *P(r)* to ***p(r)*** — that's where our *P(s,a)* comes from, as we only run the algorithm if the current position has been visited, so *P(s)* will have already been computed.
- Once the pre-selected number of simulations (AlphaZero uses 1600) have been completed, we now have a distribution, called *policy*, of how many times each of the legal moves in position *s* has been visited. Recall that only the most promising moves were visited in every simulation, so the policy is a direct reflection of how promising the agent has deemed each of the moves to be. Therefore, we can use it to select our next move in the game.

Phew, that was quite a handful. Well, luckily, from now on, things get a little bit easier. Before I get on to them, though, let's produce some Python code for the afore-described version game-playing agent:

Now that we've built our game-playing agent, we want to make it play itself a large number of times (AlphaZero uses 25,000) so that we can use data from the games and feed it into our new neural network. Quite simple, right? Not so fast. If we always play the most promising move, the agents will keep replaying either the same game or a very similar game over and over again, which won't provide us with much data. Therefore, a better solution is to randomly select legal moves weighted by the policy vector — the more promising the move is, the more likely it is to be played.

Okay, but what data do feed the new neural network? Well, recall that the neural network is intended to output a given position's win probability and the distribution of how promising each of the legal moves is. The former can be trained on the outcomes of the self-play games, and the latter on the policy vector, which is supposed to represent exactly how promising the moves are. Thus, each move in every game will produce a training example containing the position before the move as the input and the ultimate outcome of the game as well as the policy of the move as outputs.

Note that our first neural network will have completely random outputs, and our second neural network will output a completely random policy estimator, as it will have been trained on random policies. However, from our third neural network onward, all the outputs should be meaningful, and should only be getting more accurate with every new generation.

But what if they don't? To prevent that, we initiate arguably the most exciting part of the training process — we pit the old neural network and the new neural network against each other! We make them play a set number of games (AlphaZero uses 400), and if the new network's win ratio is higher than a pre-selected threshold (AlphaZero uses 55%), we use the new network to build our new game-playing agent and continue the training process; otherwise, we use the old network.

We repeat the entire process a set number of times (AlphaZero uses 200), and voila! Here we have a revolutionary chess-playing machine the likes of which had never been seen before. I mean, you'll still need incredible hardware and hours of training time to boot, but, unfortunately, I can't help you with that.

Implications

Okay, so now we know what the state of affairs was before AlphaZero came along, and how AlphaZero has changed it. In theory, at least. So what about in practice? Before I delve into some specific examples, let's first think about what we might expect.

We now know that AlphaZero selects the move to play based, at its core, on which move yields the highest estimated ultimate win probability, as opposed to traditional engines, whose move of choice is the one that leads to the most favourable-looking position a fixed amount down the line. Therefore, we might expect to see AlphaZero, unlike traditional engines, prefer moves that don't result in the most favourable short-term positions, but that increase the probability of a win come the end of the game.

Furthermore, we know that the evaluation function of traditional engines is based on rather simplistic heuristics, while AlphaZero's evaluation function is the product of a highly sophisticated neural network's experience accumulated over millions of games. Hence, one might suspect that AlphaZero would be more likely to recognise abstract concepts, such as those entailed by the solutions to moremovers.

Finally, arguably the most straightforward (yet also arguably the most exciting) prediction we can make is that, since AlphaZero does not make use of any human knowledge, unlike traditional engines (which use not only human-built

heuristics, but also opening books and sometimes endgame tablebases), we can expect it to come up with brand-new ideas previously unknown to mankind. One might have especially high hopes for the opening phase of the game, which is almost entirely circumvented by traditional engines that rely on a human-composed opening book.

So how do these predictions stack up? I'll answer this question demonstratively with an example from a real game played by AlphaZero.

Move 41 of a game between AlphaZero and Stockfish 8. Source: Author.

In this position, superficially, it appears that White's h-pawn serves to undermine the defence around Black's king by putting pressure on Black's g-pawn. However, AlphaZero decides to push its h-pawn up one square, surrendering the pressure on Black's defences and killing White's hopes of an attack.

Position after 42. h6. Source: Author.

Why did it do that? Its intentions got revealed 8 moves later: with a series of manoeuvres, it all but forced Black's queen to retreat all the way back to h8, after which it ingeniously locked it out of the game by shutting the airways with a rook.

Position after 49. Rf6. Source: Author.

And suddenly, the reason behind the h-pawn push seems clear: the black queen would love to get to g7 and then to f8, but the h-pawn prevents it from doing so.

Stockfish 8, a traditional engine, and the strongest one in the world at the time at that, underestimated this idea: from far away, it deemed this position to be favourable, as black has more and better pieces. On the other hand, AlphaZero was able to see further than that, understanding that the black queen is as good as nonexistent for the rest of the game.

Pushing flank pawns in front of one's king all the way up the board is an idea that had never been taken seriously by grandmasters; however, since AlphaZero demonstrated the proof-of-concept, this idea has routinely featured at the very top human level, notably being utilised by five-time world champion Magnus Carlsen.

While this is just one example, AlphaZero consistently demonstrated superior long-term understanding throughout its match against Stockfish 8, and has introduced many never-before-seen ideas in use by the top grandmasters to this day.

As to abstract concepts, the results have been a mixed bag. On the one hand, strategies such as the one described above clearly indicate a higher level of abstract understanding than what was previously seen: in the above scenario, for example, it would be impossible to correctly evaluate the position without recognising the abstract concept of a piece being permanently locked out of the game. On the other hand, AlphaZero and its clone LeelaChessZero still struggle with puzzles that have been solved by humans, and there are some positions that are trivial to even weak human players that they both misevaluate.

So, what can we conclude about artificial intelligence in general from this chess-centred study? We have seen that, via the use of neural networks, machines can be trained to master concepts that used to be exclusive to humans, such as very long-term planning. We have also seen that a neural-network-based architecture improves over competing architectures mostly where humans also improve over them: in our case, it was creativity and long-term thinking. From this, we might hypothesise that neural networks may yet break new ground in fields in which humans still have some advantages over machines, even if the machines are overall more competent than humans in said fields. Therefore, perhaps more machine-dominated areas like chess should be revisited by neural networks. However, with all of this said, we have also seen that, as impressive as neural networks are, they are still a far cry from humans in terms of some of the more general aspects of intelligence, such as abstract thinking.

My concluding statement will hence be: artificial intelligence is evidently advancing at a mindboggling pace; however, we still have a long way to go before it reaches its full potential.

Artificial Intelligence | Neural Networks | Alphazero | Chess | Deep Dives

Responses (2)

What are your thoughts?

Respond

Ben Bellerose
May 6, 2022

AlphaZero is amazing, and MuZero takes it that much further by playing even more games (Atari). Another cool model by then is Player of Games that does imperfect information games.

1 1 reply Reply

Herbert Braun
May 7, 2022

Great read, would have loved to see more game examples. Small correction: endgame tablebases have nothing to do with human expert knowledge.

1 reply Reply