Part 2

Most intelligent tasks we perform in our lives, we learn those skills through examples rather than being told step-by-step how to do them. For example, no one told us how to recognize the numbers, but showed us many examples, and our minds figured out some subconscious rules to distinguish a “1” from a “2” and all the other digits. Scientists quickly realized that if machines must do complex tasks, they cannot be taught algorithmically, with step-by-step instructions, as we ourselves may not know these logical steps, but rather by showing many examples of the correct behavior. Although that was the holy grail of AI, machine learning was a hard task.

In part 1 of this two-part article, we explored how we can use step-by-step instructions or algorithms to play complex games like chess. In this approach, we used an algorithm to search through a huge game tree, and used rules of thumb to determine which branches of this enormous tree can be ignored and to determine the advantages of a specific board position. We saw that such techniques can yield powerful machines that can beat the chess world champion, but we also realized their limitations. There are games more complex than Chess where this method is inadequate. In the second part, we will see how machines can learn these rules of thumb or heuristics simply by watching others and, eventually, by playing against themselves and learning from that experience.

In the early days of AI, scientists could imagine machines that could learn a complex task through suitable examples or training, but we had yet to build such a machine. Warren McCulloch and Walter Pitts introduced the first idea of a machine capable of learning in 1943. Frank Rosenblatt implemented this idea on a

computer in 1957, and it was called a Perceptron.

The idea of a trainable machine can be best understood by a simple learning machine. In 1960, Donald Michie, a British computer scientist, came up with the idea of a machine made of matchboxes and colored beads that could learn how to play tic-tac-toe and eventually become an expert player.[1] In the game of tic-tac-toe, there are only 304 unique game states, where each game state is a possible arrangement of Xs and Os on the board. There are many more distinct game states, but many of them are logically equivalent and are collapsed together (see Figure 2). For example, one game state could be an O in the center, an X in one corner, and another O next to the corner X. Now, let's imagine we assign a unique color to each of the nine cells (Figure 1).

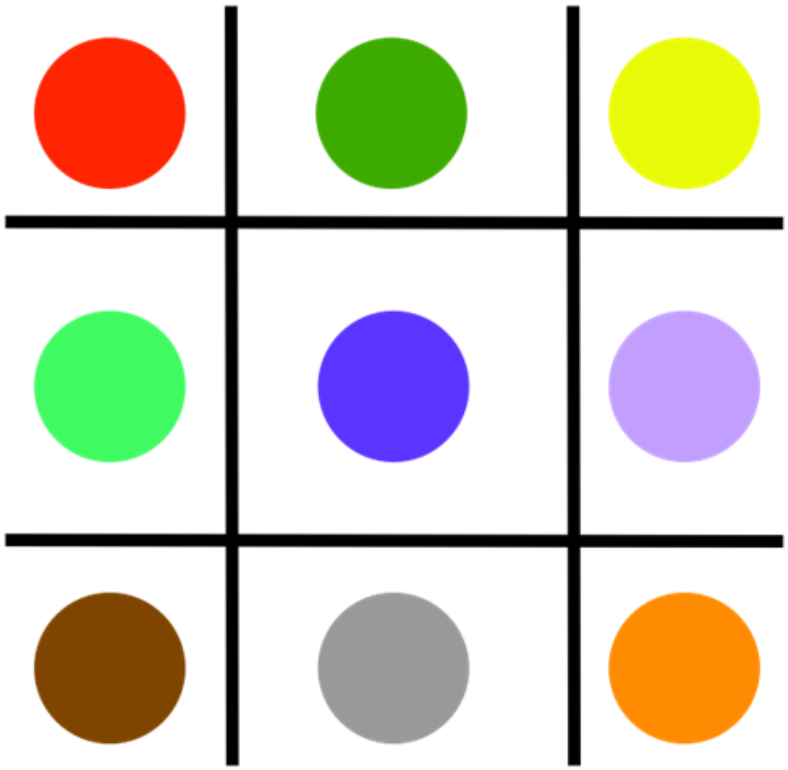


Figure 1: Color coding Tic-Tac-Toe board positions.

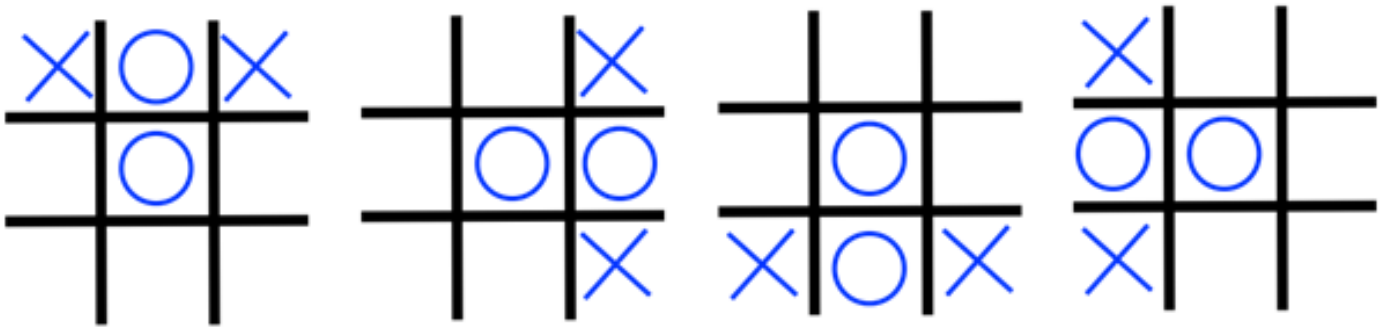


Figure 2: Equivalent game positions in Tic-Tac-Toe.

The machine is made of 304 empty matchboxes. In each matchbox, we attach a diagram of one of these unique game states (Figure 3). Given a position, the machine has only a few legal moves. For each cell where the machine can make its next mark, we pick colored beads of the corresponding color and put them in the matchbox.

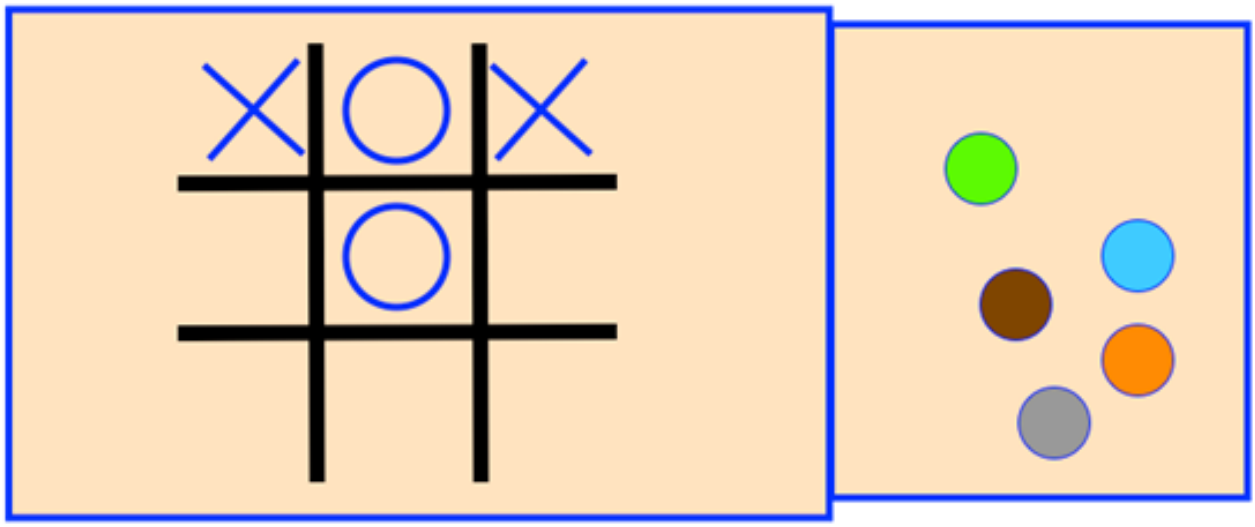


Figure 3: A labeled matchbox with five colored beads, indicating five legal moves

When the match starts, the machine makes its first move by selecting the matchbox representing an empty board. Since, at this point, all nine cells are legal places to start, we will find nine dis-

tinctly colored beads in that box. As the machine's operator, our job will be to shake the matchbox, open it, and then, without looking, pick a random bead. Depending on the color of the bead, we will put an O on the cell indicated by the bead. The human will then make their first move by placing an X. This O and X pair now defines the next state of the game, and the operator will pick the matchbox that matches this state. As before, the operator must shake the matchbox and pick a bead randomly, determining the next move. Since all the moves are purely random, the machine will play a legal game, but will most likely lose to the human.

When the game ends, assuming it will be a loss to the human, the operator will "punish" the machine by removing the beads corresponding to each of the moves it made during the game. In other words, the machine will not make the same fatal mistakes again. This process will go on game after game. Let's say, eventually, the machine wins a game. When that happens, the operator will "reward" the machine by adding three beads of the same color to each of the moves it made that led to a victory. Doing so, the probability of making these same moves in the same situations will be enhanced. If a game ends in a draw, which is also a positive outcome, the operator will "reward" the machine with one extra bead of the same color that led to the draw.

With each such game, the machine learns not to make the same mistakes but also to make better moves. After about 150 games, the machine becomes virtually unbeatable, performing as well as any human player. This is an astounding demonstration of a simple machine learning to do a task only through practice. No one explicitly instructed the machine about how to play the game well, yet it learns to improve until it reaches perfection.

If you are intrigued by this and want to try it yourself and you are reluctant to collect hundreds of matchboxes and beads, you can find a beautiful simulation of the machine here [2]. In this

simulation, you will start with an untrained matchbox computer and play against it. With each game, you will notice that the machine is improving, and by looking at all the beads in each matchbox, you can see how that is happening. It will also show you the improvement of the machine over time.

While this is one of the most tactile ways to understand how machines can learn, it also raises some fundamental questions that are still actively debated among scientists and philosophers today. While this computer can learn to play a near-perfect game of tic-tac-toe, can it explain any of its moves? Does it understand? A child may learn the game through a similar process, but once mastered, if we ask the child why they made a particular move, they may say something like, "if I didn't put a cross in this spot, the other player would win the game in their next turn since they already have two in a row". So, the question is, what makes us offer explanations when our well-performing machines fail to do so? Clearly, an individual matchbox, or an isolated neuron in our brain, does not understand tic-tac-toe. Still, somehow, the whole system possesses the ability that we loosely call "understanding."

The same principle can be applied to much more complex games like chess. However, to create a matchbox computer that can learn to play chess, we will need roughly 1044 matchboxes. That is, we will need a good part of the solar system to stack all of our matchboxes. What worked for a small game like tic-tac-toe would not work for a complex game like chess.

While classic search-based algorithms dominated the chess world, others tried a completely new method based on neural networks. This approach to AI is often referred to as the Connectionist approach, where the strength comes from the connection patterns of a complex network of simple components. It is an alternative to the Symbolic approach, where knowledge is explicitly encapsulated in well-defined rules, and the system provides solutions

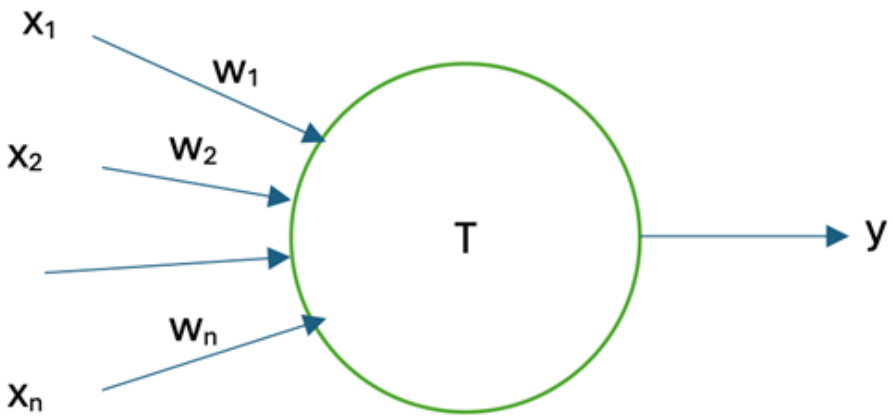
through logical and analytical processing of these symbolic facts.

Game	Board Size	State Space Complexity	Game Tree Complexity	Av. Game Length	Branching Factor
Tic-Tac-Toe	9	10^3	10^5	9	4
Chess	64	10^{44}	10^{123}	70	35
Go	361	10^{170}	10^{505}	211	250

In the early 1940s, scientists started modeling biological neurons using simple mathematical models. In 1958, Frank Rosenblatt developed an artificial neural network called a Perceptron. This network could classify simple patterns but failed at more complex tasks. In the 1980s, there was a revival of interest in neural networks, and multi-layer perceptrons could solve more complex pattern recognition tasks.

The basic unit of a neural net is a simple mathematical model of a neuron. In the earliest incarnation, these neurons were called perceptrons. A perceptron has an arbitrary number of binary inputs, accepting 0 or 1 as values. Each input has an associated weight, which is a real number. The output of a perceptron is also binary, and the value is calculated by calculating the weighted sum of the inputs and then checking whether this weighted sum is more or less than some threshold (Figure 4).

Let's consider a 3-input perceptron with weights of 0.5, 0.7, and 0.8. If the input to this perceptron is 1, 1, and 0, then the weighted sum would be (1 * 0.5) + (1 * 0.7) + (0 * 0.8) = 1.2. Now, if the threshold value of this perceptron is 1.3, then the output will be 0 since 1.2 is less than 1.3. However, if the input was 0, 1, 1, then the output would turn 1 since the weighted sum of 1.5 is greater than the threshold.



$$y = 1, \quad if \sum w_i * x_i \geq T$$

$$y = 0, \quad if \sum w_i * x_i < T$$

Figure 4: A perceptron

It eventually became clear that perceptrons were hard to train because sometimes a small change in the input could have a disproportionate effect on the output. As a result, during training, the consistency of the overall system became unstable. An alternative was to replace the output calculating function. A common alternative to a strictly binary output is a Sigmoid function. In this case, the input could be any number between 0 and 1, and so is the output. The mathematical function that calculates the output is also more sophisticated, resulting in more stable behavior during training. A Sigmoid neuron approximates the behavior of a perceptron when the inputs are binary (0 or 1), but has a continuous behavior when the inputs change slightly. There are many variations of this, but a common sigmoid function is the Logistic function – $F(x) = \frac{1}{1+e^{-x}}$, as shown in Figure 5.

At a very basic level, the first layer of neurons in a multi-layer neural net receives signals from the outer world (Figure 6). For ex-

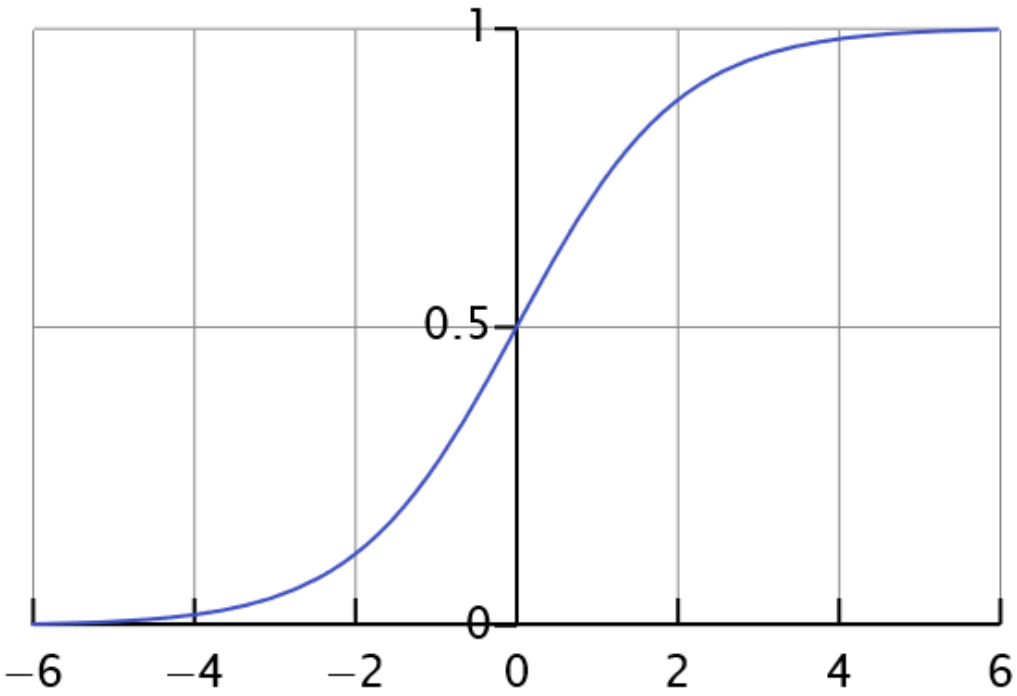


Figure 5: Logistic function

ample, a neural net designed to recognize hand-written numerals would get its input signal from an array of photoreceptors. Each cell in the receptor array would output some number between 0 and 1 depending on the pattern of the hand-written numbers. Let's say we are using an array of 8x8 photoreceptors. Therefore, there would be 64 wires carrying signals from 0 (dark) or 1 (light), and all values in between based on the grayscale of the input image. These 64 wires would act as input to the 64 neurons in the first layer.

Each of these neurons will have a single output wire. Each of these 64 outputs will act as the input to each of the neurons in the second layer. Let's say there are 20 neurons in the second layer of our neural net. Therefore, each of these twenty neurons will have 64 inputs, coming from each neuron in the input layer.

There may be additional inner layers. If the third layer exists, it can contain an arbitrary number of neurons as determined by the designers of the neural network. For instance, let's assume our

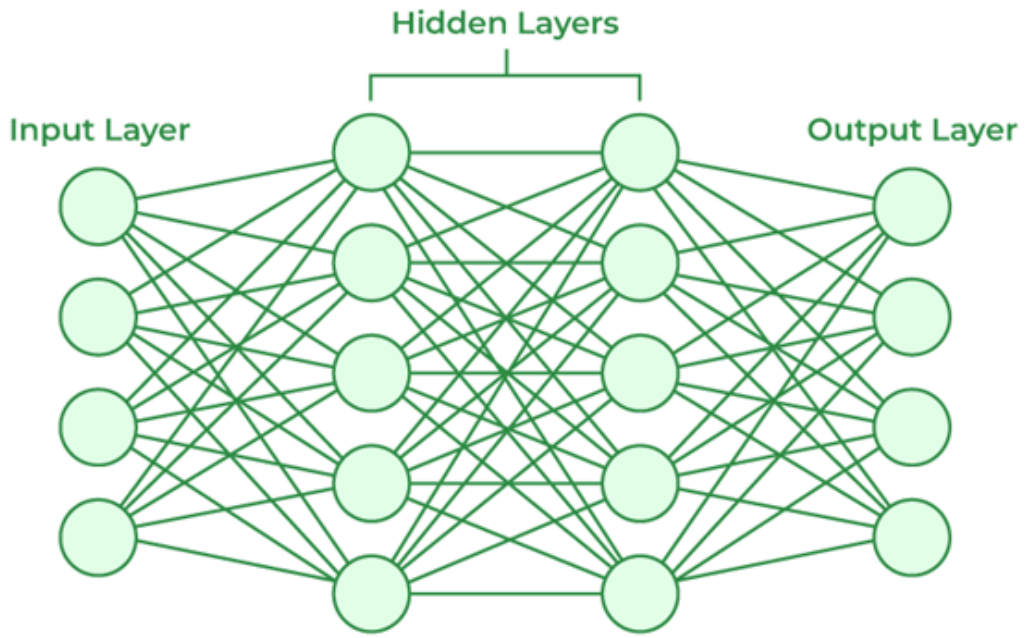


Figure 6: A simple neural net with 4 inputs, 4 outputs, and 2 hidden layers

third layer also has 20 neurons. In this scenario, the 20 outputs from the second layer will serve as the inputs to the third layer. Specifically, each neuron in the third layer will receive 20 inputs, each connected to the 20 neurons of the second layer.

The final layer of our neural net is called the output layer. It will have exactly as many neurons as our classification task demands. Since our task is to recognize handwritten numerals, we will have 10 output neurons representing the ten digits from 0 to 9. Each neuron in the fourth (output) layer will once again have 20 inputs, each connected to the outputs of the third layer.

The model is trained by showing a sequence of handwritten numerals. Each time, if the final output does not match the correct input number, this feedback is used to adjust the thousands of input parameters of each neuron and their thresholds. The technique used to apply these small corrections to the parameters is called backpropagation. Very simply, if a particular output is wrong, we nudge the parameters of all its input neurons slightly

in the direction that it is less likely to make the same mistake again. As more examples are shown and the parameters are adjusted each time, the model gradually improves. When fully trained, hidden patterns emerge. For example, we may discover that one of the hidden neurons in level 2 or 3 has specialized to detect a loop, as in the numbers 6, 8, and 9. Yet another neuron may specialize in detecting a vertical stroke as in 1, 4, or 7.

Over the last decade, basic neural net technology has undergone a series of enhancements. With more powerful computers, specialized parallel processors that can handle the computations needed for neural nets, and the availability of massive quantities of training data, the field experienced an unprecedented growth spurt. For the first time in its 75-year history, AI is proving commercially viable, and money is pouring in. This commercial turn started with a few very public events, and the center of it was again a board game.

In 2010, DeepMind was founded as a small British startup researching a more complex version of neural nets called deep neural nets. They stunned the world by creating a neural net that could learn how to play simple 1970s video games like Pong and Space Invaders without anyone telling the system anything about these games. All it had as input was the pixel output of a screen where the game was being played and the scores. Using a neural net technique called Reinforcement Learning, it taught itself the games and learned how to play them. The most remarkable quality of it was that, unlike Deep Blue, it was not specifically programmed for these tasks. Everything was learned from scratch, unsupervised, like a child would pick up a new game just by watching it and trying it out. In 2014, Google acquired the company.

While computers mastered the game of chess, it was generally accepted that such brute force search techniques would not work in

the case of Go. This ancient (500 BCE) Asian game is played on a 19x19 board where players take turns placing a pebble on any grid intersection (Figure 7). If pebbles belonging to one player surround another player's pieces, the encircled pieces are captured. Even though it is a simple game, it is far more strategic than chess, with a game space of 10360. In other words, the game space of Go is 10240 times larger than that of chess. DeepMind wanted to tackle this "impossible" task using a search algorithm augmented by a neural net-based system to pick the best moves. In 2015, AlphaGo surprised the world by defeating a human professional player. In March of 2016, it beat the world champion Lee Sedol in a five-game match, winning 4 out of 5 games. Lee Sedol later remarked that some of the pivotal moves made by AlphaGo were the most creative he had encountered.

Figure 7: A Go board [3]

AlphaGo was trained by exposing it to a vast array of human

games. Next, DeepMind attempted a different approach. In AlphaGo Zero the program was not shown a single human game. Instead, the program learned by playing against a copy of itself. As it started its training, the initial games were played very poorly. However, with each game, it learned more. In three days of training, by playing against itself, it reached a level to beat the original AlphaGo, winning 100 games to 0. In 21 days, it surpassed all the previous versions of AlphaGo.

In December of 2017, DeepMind released a program called AlphaZero. This was a generalized version of AlphaGo Zero that can learn any other similar board game. They then decided to teach chess to AlphaZero. As with Go, the training involved no human-played games and was entirely based on playing against itself. It started from scratch, having no idea how to play the game. However, after just 8 hours of training, it could beat the reigning champion Stockfish 8 in a 100-game tournament, winning 28, losing none, and 72 draws. It is also remarkable to note that while Stockfish evaluates 70 million positions per second, AlphaZero only evaluates 80 thousand positions. This 875-fold improvement is possible because AlphaZero can focus much better on the most promising variations using its deep neural network.

Watching games played by AlphaZero, a Danish grandmaster likened it to the play of a superior alien species. There were some initial criticisms that Google's AlphaZero had some hardware advantage over Stockfish. However, later, it competed against Stockfish 9 under tournament conditions and proved its superiority by winning most of the games.

In just 65 years, since we created the first chess-playing machine, we have come a long way. The initial programs could only beat a few novice players. After 40 years of improved computer power, they could beat the best human players, but they utilized brute-

force techniques and relied heavily on databases of human-played games. It took another 20 years to create self-learning machines that could learn from their mistakes and achieve superhuman capabilities without ever seeing a human-played game.

So, what comes next? One thing that is likely to happen is further generalizations of learning models. Just as AlphaZero generalized a model that can learn any strategic board game, we may see this generalization extended to other complex tasks. However, there is a distinct advantage of board games, where gameplay rules are entirely defined. Our real-life scenarios are not so. A self-driving car cannot anticipate that on Halloween night, a child can dress up as a mailbox and try to cross the street. Making learning machines deal with our messy world will be the next big challenge. We have seen great progress in recent years in the area of human language. Even though human language can also be messy with infinite variations, we have been able to tame it with Large Language Models (LLMs) such as ChatGPT because we could train our machines with billions of documents available to us. Similarly, a machine can recognize a cat or a dog inside a complex photograph better than we can, simply because we have millions of pictures of our pets on social media. However, the apparently simple task of loading a dishwasher by a robot remains a far greater challenge, as we don't have adequate data to train a machine.

There is a deeper hurdle. While neural nets are effective tools, they don't exactly represent what happens in our brains. Our neurons are far more complex, and we still lack adequate knowledge of their organizational structure. While spectacular in their behavior, LLMs need billions of documents to master a language. A human child, by contrast, listens to their parents and a few other adults for a relatively brief period and picks up a language. This happens on hardware, which is many orders of magnitude

slower than its electronic counterpart. This clearly indicates that we need a much better understanding of our brains and many engineering breakthroughs to achieve this efficiency. Simply scaling up the current technology may achieve a lot, but it will again be a brute-force approach. Given the current rate of progress and the economic pressures to develop better AI, these breakthroughs are almost inevitable and may occur relatively soon. It would be risky to predict beyond that.

References

[1] D. Michie, *Experiments on the mechanization of game-learning part i. characterization of the model and its parameters*, https://people.csail.mit.edu/brooks/idocs/matchbox.pdf.

[2] M. Scroggs, *MENACE*, https://www.mscroggs.co.uk/menace/, Accessed: 2025-11-4.

[3] Wikipedia contributors, *Go (game)*, https://en.wikipedia.org/w/index.php?title=Go_(game)&oldid=1337540495, Feb. 2026.