linkedin.com

(9) Artificial Intelligence and Turing machines - Part 1: Algorithms

Edit article

9–11 minutes

You most likely heard about the Turing test in AI[1], which can roughly be summarised as: “In a written conversation, can a person tell whether they're talking to another human, or to a computer?” If we cannot tell the difference, then the computer has passed the Turing test.

The concept of “Turing machines” is a little less known. They are an abstract model of how computers work. This model is very useful in understanding things that computers can do, things that computers cannot do, and what things can be considered “easy” or “hard” for computers.

This two-part article is an overview of Turing machines and a discussion of why they matter for AI. Let me start by talking about Turing machines.

Turing machines, simplified

Put yourself into Alan Turing shoes, you live in a world where computers, smartphones and the internet do not exist yet, and you’re imagining what an intelligent machine could look like. You look at how people do calculations with pen and paper; they write something, they stop and think, they read something, they write again… and so on.

Alan Turing made a mathematical model of computation that today we call “Turing machine”. A simple machine that can read symbols on paper, has internal memory, and can write symbols on paper. It is a greatly simplified model of how we do calculation.

- Simplify the “thinking” part: model it as changing the internal memory from one possible “state” to another possible “state”.
- Simplify our reading and writing actions: make them limited to one symbol per page, with the assumption that the machine can use as many pieces of paper as it needs.

[a physical implementation of a turing machine with a paper tape]

If you built a turning machine, it would look like this. Image credits: Rocky Acosta [CC BY 3.0 (https://creativecommons.org/licenses/by/3.0)]

Algorithms, explained with Turing machines

What can a Turing machine do? The answer is: everything! Any computation we can imagine doing with pen and paper, can be done by a mechanical machine.

This is a fundamental idea that is called the “Church-Turing thesis” roughly stated as “Whenever there is an effective method for obtaining the values of a mathematical function, the function can be computed by a Turing machine.”[2]

Which also means that **any function that we can compute, can be computed by a machine**. Our computers today are (on a theoretical level) equivalent to Turing machines, because their operation could be simulated by a Turing machine, and today’s computers can simulate any Turing machine. Now that we know what a Turing machine is, we can see algorithms in a new light.

An algorithm is a very precise definition of what steps to take in

order to finish a computation. For instance, a recipe for cooking pizza is an algorithm, matrix multiplication is an algorithm, quicksort is an algorithm. Running the algorithm is equivalent to running a Turing machine that performs that computation. **On a theoretical level, algorithms, mathematical functions and Turing machines are all equivalent**.

One important practical difference with today's modern computers is speed, a Turing machine is an abstract idea, and the speed at which it can read/write and change state does not matter, while in practice the speed at which our computers can run matters a great deal.

Another important practical difference is that Turing machines are algorithms running in isolation: once it's set to run, it's assumed we don't touch it and we leave it running, while obviously for today's modern computers we do a lot more interaction, including clicking the power button on our phones to pause the computation to save battery life, or stopping an application that is taking too long for our taste.

Will this machine terminate?

We've seen how the mathematical model of **Turing machines is useful to add a mechanical interpretation to the notion of algorithm**. Another important theoretical question that Turing machines can help with, is finding out if our instructions to the machine will produce the output we want or not.

Unfortunately, the answer to this theoretical question is negative, there is no general way to make sure our algorithms will produce the correct output. Sure we can, and we do, verify specific algorithms when it's important (think CPU correctness) but there's a well-known result called the **"halting problem" that states it's impossible to build a machine that can check our algorithms. It's impossible to build a machine to check that our computation will even**

terminate, let alone produce the right answer.

The P=NP question

Another well-known question related to Turing machines, is the P=NP problem. There is a $1 million prize at the clay mathematics institute for whoever can settle this question[2]. But don't rush to solve it, many experts have tried.

I can't describe it as precisely and elegantly as Stephen Cook does for the millenium prize, so here is just a quick overview.

Suppose you know of a few problems that computers can solve really fast (e.g. do an online search), and you're aware of some problems that computers can solve, but they're really slow (e.g. discover the private encryption key given the public encryption key). You also know some problems cannot be solved at all, no matter how hard you try (e.g. solve all Diophantine equations[3]).

Easy problems are in the "P" set, we can roughly define all problems as "easy" if there is a Turing machine that can calculate a solution, and the machine does not require too many steps of computation[4]. **The "NP" set contains all the easy problems, plus all the hard problems that we can solve with a Turing machine, but one which takes way too long to finish**[5].

The first, obvious answer now would be: "P is smaller than NP: there are difficult problems that are not easy, that's the very definition of difficult!" But are we sure about it?

Suppose we have a computer that can solve a hard problem, but it's taking too long. You call a clever, very clever computer programmer, and with a few days (or a few years) of hard thinking they discover a new method, a new computer program that solves the same problem really fast. **That problem suddenly changed side, it was a hard problem in the past, but it's an easy problem now**. All thanks to

someone who found a very clever solution.

That's the core of the question, are we sure that some problems will be difficult forever, or can we always find a fast solution? What if there was a fast solution to all the problems we currently can't solve quickly? What a wonderful world it would be, efficient computers solving lots of complex problems quickly. But careful, you don't want people to easily crack encryption keys. **The question whether P=NP is still open, and it will probably stay that way. We might even find that the very question is a problem that cannot be solved**.

This concludes part 1 of this article, in Part 2 we will explore even more theoretical questions of what computers can and cannot do.

[1] This particular post is not about machine learning or deep learning, but computability and complexity theory. Remember, machine learning is a way to produce algorithms, not an algorithm on its own, see for instance this post that highlitghts the differences https://medium.com/@geomblog/when-an-algorithm-isn-t-2b9fe01b9bb5

[2] Stephen Cook millenium prize challenge can be found here http://www.claymath.org/millennium-problems/p-vs-np-problem

[3] An example of a problem for which no algorithm exist http://mathworld.wolfram.com/DiophantineEquation.html

[4] P stands for Polynomial time, the number of steps won't be more than a polynomial function of the input size

[5] NP stands for Non-deterministic Polynomial, it would take a polynomial number of steps for a non-deterministic Turing machine (which roughly means a Turing machine that is allowed to check all possible combinations in a single step).

Appendix: turing machines formal definition

Define a finite set of states, a finite set of symbols that can be read or

written, and a list of instructions, each composed of 4 elements: (initial state, symbol, action, new state) where the states have to be part of the set of possible states, the symbol has to be one of the possible symbols, and the action is one of three options: move left, move right, or a symbol to write. That is a Turing machine. The process of computation consists of reading and writing symbols on a paper tape of arbitrary length, the machine scans one page of the tape at a time, each page contains only one symbol, the machine can move the tape to the left or to the right to read or write another page on the tape.

A computation for a Turing machine starts with an initial state for the machine, and with the tape containing some symbols, the initial state of the tape is called the input of the computation. The Turing machine always tries to match an instruction with the current state and symbol. If it finds one, it executes the corresponding action and switches to the corresponding new state. If it doesn't, the machine terminates with an error. If the machine reaches a pre-defined end state, the computation is finished with success, and whatever is written on the tape is the output of the computation.