Bottom-Up Artificial General Intelligence

Gleb Shevchuk

Abstract

Artificial general intelligence can be achieved by training a model to solve increasingly difficult problems expressed in a Turing-complete language. To make this claim, we describe general intelligence from the perspective of complexity theory, present a generalized Turing Test for determining intelligence, and discuss the limitations of training a model to progressively learn the set of all computable problems.

1 Ideal AGI

To teach human-level intelligence, we might train a model on the set of all human problems (language, mathematics, etc.). To teach general intelligence (AGI), we must begin with a more generic, Turing-complete, language that allows us to express both human and super-human problems. In this paper, we will argue that the only way to reach general intelligence is to define all problems in such a language and train a model on problems of increasing difficulty.

Imagine, then, that an agent achieves general intelligence by learning **in this bottom up manner, on increasingly difficult problems until it is able to solve all computable problems**. In this setting, what does generalization look like?

1) if the agent learns a problem in a simple setting, it perfects that same problem in a more complex setting.

2) if the agent learns a set of simple problems, it uses that knowledge to solve much more complex problems.

Importantly, this implies that if there is a program that uses finite resources and produces outputs for a set of valid inputs, our model produces the correct output for all inputs to that program. We stress that this ought to be true for the set of all possible programs. Having set the stage, let us formalize this problem using the language of Turing Machines (TMs), talk about problem difficulty and similarity, and define a generalized Turing Test for measuring intelligence.

2 Notation

To begin, we define a problem τ as a Turing Machine f and a set of corresponding input/output examples $[(x_i, y_i), \ldots, (x_n, y_n)]$. Computability theory tells us that there are problems that are unsolvable, so we limit ourselves to the subset of computable problems.

$$\tau =< f, [(x_i, y_i), \ldots, (x_n, y_n)] > \tag{1}$$

Turing Machines are the standard language for discussing computation and consist of three core components: an input tape, a head, and a set of instructions. The tape stores a finite, one dimensional sequence of symbols and starts in a special configuration x. The head starts on a particular symbol x_0 of the tape and is allowed to move left and right and either read or write to the symbol that it is currently on. Finally, the set of instructions specifies what action the head should perform at each position on the tape and when it should terminate. Though this is only one of many variants of a TM, it is general enough that we can define any computer program using it. In doing so, it allows us to define a program as a sequence of steps that change elements of a tape, one-by-one, and produces an output after a fixed number of steps [1].

Notably, though the formal description of TM is often described as a 7-tuple, we will stick with this simplified description that only depends on a fixed length input x, a set of instructions p, and a head that operates discretely on the tape. For the sake of clarity, we do not need to assume that the head has its own separate set of states or that there is an additional working tape. As a last piece, we can assume that TMs are compositional, meaning that a TM F can be described as a sequence or composition of smaller subroutine TM's $[f_1, \ldots, f_n]$.

In turn, whenever we train a model, we train it on a sequence of n problems, $[\tau_0, \ldots, \tau_n]$. This allows us to combine supervised learning and sequential learning problems, as we can treat sequential learning as analogous to learning several problems, one after another, with each problem having a corresponding TM where the previous state is encoded as a subroutine. In our ideal AGI case, we would be able to train on a fixed number of problems and be able to solve any subsequent problems shown during testing.

Specifically, to train a model, we produce a training dataset D_{train} which is composed of several problems $[\tau_0, \ldots, \tau_n]$, each with the minimal amount of input-output samples required to learn the problem. Then, during testing, we condition the model on input-output samples from an unseen TM, pass a larger set of inputs through the model, then measure its predicted outputs against the true corresponding outputs. For an agent to "solve" a problem, the predicted outputs should match the true outputs.

Finally, we can define a single reduction of a TM f as another TM g such that solving g also allows us to solve f. We can also extend this to the concept of learning

reduction by finding a sequence $[g_0, \ldots, g_n]$ that the sequence $[\tau_0, \ldots, \tau_n]$ reduces to.

2.1 Problem Difficulty

To determine a problem's difficulty, we normally use the length of the initial input tape x and the length of the TM. For any given TM, there are an unknown amount of TMs that reduce to it, meaning that we measure the length of the TM by measuring Kolmogorov complexity (KC), or the length of the shortest set of instructions $|p|$ that creates that required set of input-output examples. Formally, the KC of a Turing machine f with inputs x, outputs y, and instruction set p is expressed as

$$K_f(x, y) = min\{|p| : f_p(x) = y\} \tag{2}$$

KC also allows us to define the compressibility of a program. If a program is incompressible, there is no shorter program that encodes all the information needed to create that required set of samples. If a program is compressible, it implies that it can be composed out of smaller programs. Therefore, compressibility implies composability. Finally, we can define similarity between problems by measuring the similarity between their input tapes and the similarity between their two shortest corresponding programs.

2.2 In and Out Generalization

Now, we can use these metrics to define generalization. We define in-generalization as a similarity metric between the longest input tape successfully learned in training and the longest input tape that the agent converts to a correct output in testing.

$$G_{in}(\theta; \tau) = L(x_{train}, x_{test}) \tag{3}$$

where θ is the model parameters, τ is the problem that x is an input for, L is a similarity metric, x_{train} is the longest input learned in training, and x_{test} is the longest input successfully converted in testing.

This gives a more concrete idea for measuring our first requirement for an AGI, which is that if it learns a problem in a simple setting (on a simple input string), it should be able to perform that same problem on a more complex setting (on a longer input string).

Then, we can define out-generalization as the difference between the most difficult problem seen in training and the most difficult problem learned in testing, as determined by their corresponding Kolmogorov complexities:

$$G_{out}(\theta) = K_{\tau:test} - K_{\tau:train} \tag{4}$$

where θ is the model parameters, $\tau : test$ is the most complex problem seen in testing and $\tau : train$ is the most complex problem seen in training. This captures the second requirement, which is that an agent should be able to solve more difficult problems than those learned in training through generalization.

2.3 Generalized Turing Test

With these definitions of in and out generalization, we can then measure the intelligence of an AGI agent. As we train this AGI, these two metrics should increase as it learns increasingly difficult problems.

We might refer to this as a generalized Turing Test because it takes the original idea of the Turing Test, which was to measure an agent's intelligence by testing it on the specific problem of human conversation, and generalizes it to the set of all computable problems.

In turn, we can define the AGI optimization problem as maximizing the total generalization power of a model:

$$max_\theta \sum_{\tau \in \tau_\infty} [G_{in}(\theta; \tau)] + G_{out}(\theta) \tag{5}$$

where τ_∞ is a subset of all computable problems, θ is the parameters of a model, τ is a specific computable problem, and G_{in} and G_{out} correspond to in-generalization and out-generalization.

The advantage of such a metric is that it gives a clear notion of intelligence. Ideally, this means that with enough compute resources and a well-formed optimization problem, we should be able to gradually evolve an AGI that can solve all possible problems, therefore becoming "generally intelligent". The notion of compressibility also gives a direction for how we might train the agent: by teaching it to first solve simple problems and incentivize it to re-use them for more complex ones.

3 Limitations with Bottom-Up AGI

In an ideal world, this optimization problem would be well-posed and would allow us a clear recipe for creating AGI: assuming that the model we choose is expressive enough, we can start it on simple problems and teach it to gradually solve harder ones, as defined by our measures of difficulty. By training in this progressive manner, we would not have to define an arbitrary upper bound on program size and would incentivize the model to learn composability. However, we run into several notable difficulties.

3.1 Problems with Complexity

First, although KC is a widely accepted metric for complexity, it is uncomputable [2]. Therefore, it is difficult to prove that a problem is incompressible or that two problems have similar complexities, meaning that we often have to use simpler measures in practice.

Second, although this recipe allows us to easily define new problems, it makes it difficult to convert known problems of interest to those of our complete language of choice. This is most visible when we discuss problems like language comprehension

– though we know that language complexity is roughly proportional to the number of computational units in the human brain, there is no easy way to create a TM for it.

Third, the number of problem reductions increases with the size of the instruction set $|p|$ for a program, making it difficult to accurately track out-generalization since it becomes more difficult to determine KC using brute-force.

Finally, and most interestingly, the frequency of incompressible programs increases to 1 as we increase the size of the instruction set $|p|$ [3]. Therefore, it becomes more difficult to construct meaningfully complex problem and most of the problems we generate will be analogous to random noise. Though this might be useful during training, where we want to see as many examples of compressible and incompressible problems as possible, it makes benchmarking during testing a challenge.

These considerations introduce a catch-22 to any such bottom-up approach: to train a model, we have to expose it to as many easily-constructible problems as possible. However, this makes it difficult to express interesting problems like language comprehension, to determine true complexity, and to test on interesting problems.

3.2 Modeling and Training Issues

This setup also creates several restrictions for any model that we may want to train in a bottom-up fashion. First, the presence of incompressible problems might have serious implications for how large any AGI model has to be.

Conjecture. *If we want to develop a parameterized model that contains the information to solve all computable problems, the number of parameters has to increase with the number of problems it is trained on.*

Proof Sketch 1. By contradiction, assume that there exists a model that only needs a fixed number of parameters to solve all problems it is trained on. We can find an incompressible program that is less than the most difficult program seen in training. Moreover, we can construct this sample such that the model will drastically fail to learn it. The only way to learn that incompressible program is to increase our number of parameters proportional to problem size or to re-train the model with this new problem.

Essentially, if we assume that a model cannot perfectly generalize to an unseen incompressible problem, then we have to most likely increase model capacity to learn that problem and ensure the same performance on all previous problems. However, often times the only problems we care about are ones that are highly composable – like those found in physical systems –. If we assume a degree of compressibility for all problems, we can establish how well our model might out-generalize to unseen incompressible programs and how it would grow in size.

Proof Sketch 2. We can accurately model an incompressible program to some degree by assembling a solution out of compressible programs. This approach, however, presents several unknowns: if we have some base units of intelligence – akin to sin waves in Fourier decomposition –, we would be able to establish convergence towards an incompressible problem. However, it is unclear whether such units exist or what they might look like. Depending on what model we use, this might also introduces a tradeoff between accuracy and size. Establishing these bounds ought to be of great importance to creating a more rigorous theory of intelligence.

Finally, we have to determine a minimum rate of improvement for any such bottom-up method, since it might be faster to find the subset of all problems that humans care about (language comprehension, speech synthesis, theorem proving, etc.) and tailor specific solutions to them than it would be to train in a bottom-up manner.

4 Finding Sequence Reductions

Having discussed some of the issues with our setup, we turn to a much larger problem: training a model on all possible problems is simply infeasible. So, how might we find a sequence reduction that allows our model to generalize?

One approach is to treat the problem of finding the next training task as an active learning problem. Intuitively, the best training problem is one that either maximizes or minimizes the conditional complexity of a problem τ_n with respect to all previously seen tasks, $K(\tau_n|[\tau_0, \ldots, \tau_{n-1}])$. By minimizing conditional complexity, the new task will be similar to previously seen tasks and will allow us to judge a model as a composition of old problems. By maximizing conditional complexity, we judge how good the model is at learning new types of problems.

Another approach is to accept that finding true Kolmogorov complexities is impossible. In a practical setting, we do not need to know the exact complexity of each program, but rather the relative complexities of programs. In turn, if we discover an algorithm for establishing linear order of problems ($A < B < C$), this would allow us to train in a bottom-up manner. However, it is unclear if such an ordering algorithm exists or if it is equally difficult to determining KC.

We can also search for a more intermediate language or problem representation where the complexity problem is simplified and where it is easier to convey real-world problems. One way to do this is to start with a set of human-world problems and automatically find a latent space where each problem representation is simple and accurately represented. Another approach is to treat problem representation as its own meta-problem and optimize over it in addition to agent performance.

Finally, we can ignore the hard problem of complexity altogether. Since the frequency of incompressible programs increases towards 1 as problem size increases, we can simply train our agent on random problems of increasing size. Though this

seems like the most structureless approach, it allows us to find problems much faster and move issues of problem solving from the training stage to the modeling stage.

5 Conclusion

We presented a bottom-up framework for discussing AGI and discussed the problems that arise in taking such an approach. Though our discussion was formalized to TMs, any such approach to AGI involving the generation of increasingly difficult problems will face similar challenges. Namely, that we will have to define a universal computation language and come up with metrics for generalization, similarity, and problem difficulty. However, in using a Turing-complete language to generate new types of problems, we run into inescapable issues of finding interesting problems, establishing complexity, and finding a sufficient subsequence of problems.

We hope that this spurs further discussion into AGI and provides a complementary paradigm to reinforcement-learning heavy theories [4] that have dominated the discussion of AGI sofar.

References

[1] Turing, Alan Mathison. "On computable numbers, with an application to the Entscheidungsproblem." J. of Math 58.345-363 (1936): 5.

[2] Li, Ming, and Paul Vitányi. An introduction to Kolmogorov complexity and its applications. Vol. 3. New York: Springer, 2008.

[3] Arnold, Holger. "Kolmogorov complexity and the incompressibility method." (2011).

[4] Hutter, Marcus. Universal artificial intelligence: Sequential decisions based on algorithmic probability. Springer Science Business Media, 2004.