Path to Efficient AGI

Xiyu Zhai

Contents

1 Introduction

This paper is the prelude of a series of papers that explore a new school of AI methodologies, which naturally incorporates with established AI methods like deep learning, and shall lead us towards **efficient** artificial general intelligence.

Here **efficiency** is the key because an inefficient AGI will take too long to appear or too expensive to be of important usage, although they could be helpful for entry level tasks. The meaning of efficiency is twofold:

(i) Statistical efficiency. The ability to learn accurately from limited examples.

(ii) Computational efficiency. The ability to perform computation in time and within resource boundary.

Biological intelligence, especially that of human beings, border collies, huskies, etc, although being far more efficient than deep learning, is still far from perfection. We are already beaten by computer programs in terms of memorization or rule based computation. A prediction of this note is that, a near optimal AGI implementation over the architecture of CPUs and GPUs will be far superior than biological intelligence, let alone deep learning.

This paper is the blueprint of how to achieve efficient AGI, including

(i)

(ii)

(iii)

It's going to be followed by

(i) a paper on a new programming language called Husky, which satisfies the langauge requirments proposed;

(ii) a paper on image classification based on ideas in section XX;

(iii) a paper on image generation based on ideas in section XX;

2 Related Works

3 Theories Based on Turing Machines

3.1 Conventions

For the matter of succinctness, we shall in this section restrict ourselves to Turing machine level when thinking of computation. This is of course far from reality, but it helps with illuminating the high level ideas.

By an **NP** problem, we mean a decision problem together with the proof of that decision which can be verified in polynomial time.

Everything will be finite, i.e. representable in a Turing machine.

3.2 NP Problems Arising from ML Problems

$$\begin{tikzcd} \boxed{\text{ML}} \arrow[r] & \boxed{\text{NP}} \end{tikzcd}$$

We begin with the well-known fundamental (non-unique) conversion of an ML problem into an NP problem.

An ML problem is about finding a function $f \mathcal{X} \to \mathcal{Y}$ such that $\mathscr{L}(f) := \mathbb{E}_{(X,Y)\sim\mathcal{P}} l(f(X), Y)$ is small enough where $\mathcal{P}$ is an unknown distribution over $\mathcal{X} \times Y$ and $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a loss function, and we are given sampling from the unknown distribution $\mathcal{P}$ in a certain fashion (online or offline).

A model is a class $\mathcal{H}$ of functions from $\mathcal{X}$ to $\mathcal{Y}$ (called Hypothesis Space). We say the model is good if one of $f \in \mathcal{H}$ will make $\mathscr{L}(f)$ small enough.

Suppose that we're given enough data (the amount is still polynomial, which is possible if $\log \mathcal{H}$ is polynomial, which will be true if elements in $\mathcal{H}$ arer polynomially presentable), then $\widehat{\mathscr{L}}(f) := \textit{todo}$ becomes a good approximation of $\mathscr{L}(f)$. Then whether the model is a good one becomes an NP problem, with the proof being the specific $f \in \mathcal{H}$ making $\widehat{\mathscr{L}}(f)$ small enough.

More generally, we could think of meta learning. Suppose we have a family of machine learning problems indexed by S, todo

3.3 ML Problems Arising from NP Problems

$$\begin{tikzcd} \boxed{\text{ML}} \\ \boxed{\text{NP}} \arrow[u] \end{tikzcd}$$

Given an NP problem, which consists of an input space $\mathcal{X}$, a certificate space $\mathcal{C}$, and a polytime verifier $v : \mathcal{X} \to \text{Bool}$, the goal is then to find an efficient implementation or approximation of $x \mapsto \exists_{c\in\mathcal{C}} v(x, c)$.

Let's start with some brute force method for find certificates, denoted by $c_{\text{brute}}(x) : \mathcal{X} \to \mathcal{C}$.

Suppose that we are satisfied with the approximation $x \mapsto v(x, c_{\text{brute}}(x))$ and we only want to make it faster. We can collect a set of $x_i \in \mathcal{X}$, then build a dataset with $y_i = c_{brute}(x)$, then this becomes a machine learning problem.

(A more appropriate setup would be to make $c_{brute}(x)$ being a small finite set of certificates, and $v(x, c_{\text{brute}}(x))$ will become $\exists c \in c_{\text{brute}}(x), v(x, c)$)

Now in general we can't be satisfied with the simple approximation $x \mapsto v(x, c_{\text{brute}}(x))$, we can also use machine learning to make a better approximation.

One way is to divide and conquer. Factor $\mathcal{C}$ into a disjoint union $\sqcup_{i\in\mathcal{I}}\mathcal{C}_i$ indexed by a not necessarily small set $\mathcal{I}$. Then

$$f_*(x) = \exists i \in \mathcal{I}, \exists c \in \mathcal{C}_i, v(x, c). \tag{1}$$

Suppose that we have a brute force method $c_{\text{brute},i}$ for each $\mathcal{C}_i$. Define

$$g : \mathcal{X} \times \mathcal{I} \to \text{Bool}, g(x, i) = v(x, c_{\text{brute},i}(x)) \tag{2}$$

Now g is a function we can approximate with machine learning. Let the approximation we get be $\hat{g}$, then we can simplify the problem by searching within $\mathcal{I}$ of s

3.4 the Ladder of NP-ML Ascension

$$\begin{tikzcd}
 & & & \mathrm{ML}_3 \\
 & & \mathrm{ML}_2 \arrow[r] & \mathrm{NP}_3 \arrow[u] \\
 & \mathrm{ML}_1 \arrow[r] & \mathrm{NP}_2 \arrow[u] & \\
\mathrm{ML}_0 \arrow[r] & \mathrm{NP}_1 \arrow[u] & &
\end{tikzcd}$$

3.5 Reinforcement Learning

4 Theories Based on Realistic Computation Models

5 Type System

5.1 Concept Level Types

5.2 System Level Types

5.3 Types for Machine Learning

6 Language Requirements

7 System

7.1 Database

7.2 Debugger

7.3 todo