Path to Efficient AGI

Xiyu Zhai

Contents

1 Introduction

(Warning: draft in progress. This is not meant for publication at a specific conference, but a central bus where different ideas are organised and described in a coherent way.)

This paper discusses the optimal path to sample and computation efficient AGI in the era of deep learning. We first confirm theoretically the capabilities of deep learning as general purpose end-to-end machine learning methods, then argue a school of domain specific modular machine learning methods powered by a new programming language will replace deep learning as the best path to AGI.

To give insights, we make some high level discussions.

General Purpose vs Domain Specific. In many fields, general purpose methods are worse than highly optimized domain specific ones. For example,

- Matlab, Mathematica are domain specific programming languages, more effective to use than C++/python for many specific circumstances;
- for specific computation tasks, specfically designed data structures are better than general purpose ones, like 'SmallVec' rather than 'Vec'
- in mathematics, PDE domain specific techniques are more powerful than general purpose functional analysis tools.

However, in the field of AI, general purpose methods seem to perform better. People no longer use domain specific feature engineering as a decade ago. Everything is some kind of neural network.

We believe the apparent worse performance of domain specific AI techniques is largely due to engineering challenges. On one hand, feature engineering and rule based system historically failed due to either shortage of engineering efforts or inability to exploit specific domain. On the other hand, deep learning can scale easily with more data and computation are advanced by the joint effort of countless researchers and engineers around the globe.

End-to-End vs Modular By definition, "modular" is a better property in terms of debuggability, robustness, flexibility. However, under the current deep-learning based AI frameworks, modularity is harder to achieve than end-to-end.

New Programming Language Design We propose a new programming language design that would make AI engineering much more efficient, allowing domain specific models that perform much better than general purpose end-to-end methods.

The language shall address challenges in the three major aspects:

- Inference. The domain specifically optimal inference could involve computation other than matrix algebra, the language shall be able to **implement quickly system level algorithms** that are the building blocks of the overall model; furthermore, queries could be repetitive with each other in some aspects, so the language shall be able to support **incremental computation and memoization**. Lastly, we desire modular rather than end-to-end models, so the language shall support **distribute features across modules and packages**.

- Training. Gradient descent cannot be used because the proper hypothesis class will not be smoothly parametrizable in general (say, each hypothesis is a rule database, or a Rust program). So the language shall allow us to **write sophisticated domain specific (symbolic) searching algorithms for training**. We may ease the optimization difficulty by labeling some intermediate features either manually or through pretrained deep learning models, so language need to **make it trivial to integrate with manual labeling or derived label functions**. It's also possible to adapt the training process, i.e. language need to support **higher order machine learning**.

- Development. In traditional programming, we largely have clear ideas about what we do beforehand and then write code to implement them; however, in domain specific AI engineering, we need to interact to know what to do. For example, we surmise that a certain feature based on a geometric function of curves will work for MNIST, only to find that it performs good overall except a few cases. We then need to narrow down to those cases and see what's happening. It might be our idea is wrong, or might be the code is buggy, or these cases are just not meant for this function. We need an **interactive, fully debuggable and visualizable IDE system that display inputs, labels, features for either the whole dataset, or a specific subset or just a single point**.

We shall give a more detailed explanation in section ??. But the full specification will be given in a separated paper.

If successful, we shall be able to invent domain specific AI techniques that

- Inference is more efficient, robust and explainable. Could have language models without illusion yet 1000 times faster, runnable on local device.
- Training.
- Development.

In the date of writing, the language is close to a new working version, including more than 90k lines of Rust code.

2 Related Works

3 Main Results

3.1 Neural Network Expressive Power

4 Modular Pattern Recognition Graph

> The greatest benefit to recognizing patterns is that it gives you a pathway to power, and a ladder out of chaos.
> *Tony Robbins*

Here we introduce a framework to describe the computation process of pattern recognition.

4.1 Pattern Recognition

Intuitively pattern recognition is a function that is (using classical computers)

- computable. For the very least, in polynomial time and space.
- learnable. Either theoretically or empirically.

The importance of computability is needless to say. Learnability is critical for scaling, because it's much cheaper to train on machines than hiring millions of programmers to hand write everything.

Examples of pattern recognitions are

- Linear function. Computability is obviously. It's learnable both theoretically and empirically.
- Various tasks in computer vision. Empirically shown by neural networks.
- Various tasks in natural language processing. Empirically shown by neural networks.

Neural networks themselves are not viewed as "pattern recognition" because they can be hard to learn (the learning problem in neural networks is NP-complete, shown in Judd 1987, 1990).

4.2 Computation Graph

First, we use graph to represent a complicated function composed from simple ones. We use graph instead of DSL for the benefit of clarity.

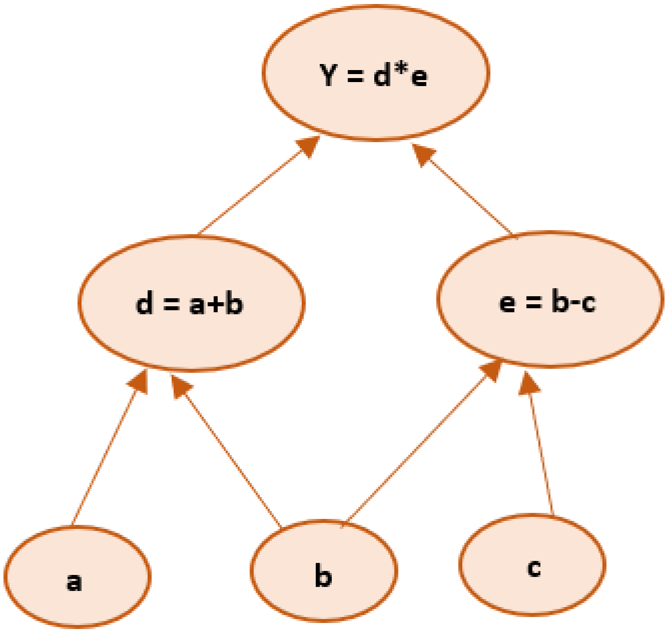
This is not much different from those graphs for neural networks.

Definition (Computation Graph)**.** A computation graph is a directed graph with nodes being types, and for a node T, with a nonempty list of incoming edges, the starting node type of which are $S_1, \cdots, S_n$, there is a function

$$f : (S_1, \cdots, S_n) \to T$$

.

All the nodes without incoming edges are seen as inputs, and all the nodes without outgoing edges are seen as outputs.



4.3 Pattern Recognition Graph

4.4 Modular Pattern Recognition Graph

Pattern recognition graph is a specialized computational graph such that it's computable, learnable, explainable and well-suited to a specific domain.

(Explainability is possible because we only allow nodes to be associated with explainably types, i.e. a type such that the value of that type has clear meaning)

To avoid unnecessary confusion from over abstraction, we discuss computer vision and natural language separately, but the similarity can still be easy to see.

Definition (Pattern Recognition Graph for Image Input)**.** The input is 2D rasterized image, i.e. a $M \times N$ matrix with each entry being RGB value.

4.5 Evolving Modular Pattern Recognition Graph

4.6 Relational Database Query

4.7 Type Checking

4.8 Formal Verification and Theorem Proving

4.9 Transpilation to Neural Networks

4.10 Transpilation to More Efficient and Robustness Programs

5 AGI Theories Based on Turing Machines

Pattern recognition graph is just about a single function, we will discuss how to compose them to form intelligence.

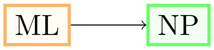
5.1 Conventions

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.

5.2 NP Problems Arising from ML Problems



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

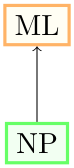
An ML problem is about finding a function $f\ \mathcal{X} \to \mathcal{Y}$ such that $\mathscr{L}(f) := \mathbb{E}_{(X,Y)\sim\mathcal{P}} l(f(X), Y)$ is small enough where $\mathcal{P}$ is an unknown distribution over $\mathcal{X} \times Y$ and $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a 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

5.3 ML Problems Arising from NP Problems



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

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

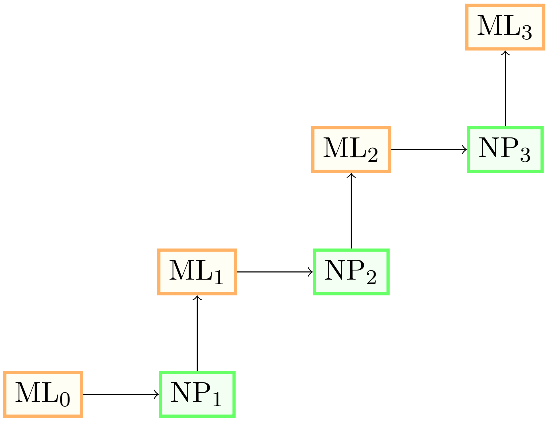
Now g is a function we can approximate with machine learning. Let the approximation we get be $\hat{g}$, then we can simplify the problem by searching within $\mathcal{I}$ of s

... RL is a special case.

5.4 the Ladder of NP-ML Ascension

"Good mathematicians see analogies. Great mathematicians see analogies between analogies."

– Stefan Banach, as in Banach Space



The typical machine learnings algorithms including deep learning, typically doesn't go up beyond NP_1, that's why they can't achieve AGI and needs a lot of amount of data to fake AGI in restricted scenarios.

Todo: give many concrete examples. Many will come from programming and mathematics.

Todo: explain why it doesn't take much data to ascend.

5.5 NP_0 Preceding ML_0: CV and NLP

6 AGI Theories Based on Realistic Computation Models

6.1 Locally Sensitive Hashing

7 Type System

7.1 Concept Level Types

7.2 System Level Types

7.3 Types for Machine Learning

8 Husky Programming Language

Here we give a brief description of the Husky programming language, invented to implement the above ideas.

8.1 Requirements for AGI-Ready Programming Language

8.2

8.3

Here we lay down the requirements for a programming language that can implement the efficient AGI as we described.

Basic Requirements We first discuss basic requirements, each of which is satisfied by at least one current programming language or is among the future design goals. However, there is no single language that satisfies all of them, a direct consequence of the fact that computer science has only been developed for 70 years.

(i) purity.

We don't want to use a programming language with which we can shoot ourselves in the feet.

Logic errors is fine, but it can be detected in the debugger and handled in a controlled manner, but memory bugs and other undefined behaviors will make a large system extremely hard to track. One needs to know much more to debug a memory bug, than a logic one.

(ii) modularity

(iii) term level coding

(iv) system level coding

(v) all level debugging support

(vi)

Advanced Requirements

(i) express evolving computation graph

9 Plans for Computer Vision

10 Plans for Natural Language Processing

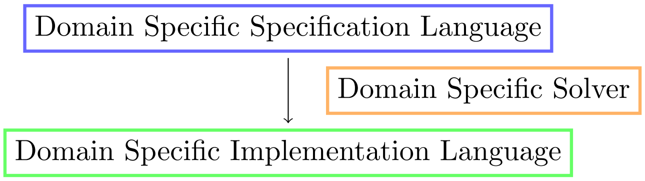
As claimed in previous sections, deep learning's success doesn't mean it's the optimal for natural language processing, but it will serve as a convenient tool for the development of a next generation far more superior tools.

Let's first conduct a worst scenario analysis. Suppose deep learning achieved AGI before us, and the AGI works amazingly well. Let's recall what makes symbolic AI methods like expert system fail: high development, maintenance, and debugging cost. But we can instruct deep learning AGI develop expert system and write out programs that parse English like parsing a programming language. It would take forever for humans, but for AGI, it would be trivial. The takeaway is, even if deep learning takes the holy grail, it won't keep it for very long as humans could then replace it a more rule-based AI with its help.

But this argument is not satisfying, due to the gloomy prospect of deep learning achieving AGI. Still there could be many practical approaches where we can leverage deep learning in its current form to build a more advanced form of AI, which I shall explain below.

10.1 Stage 0: Domain Specific Verifiable Natural Language Processing

We shall restrict ourselves to a very specific domain, relatively simple, so that we can easily build up a domain specific programming langauge to describe the tasks and also build efficient solvers for the task.


Domain Specific Specification Language
Domain Specific Solver
Domain Specific Implementation Language

Example (Mathematica)**.**

Example (Html, Css)**.** Html and Css are not specification languages, they are implementation languages.

Let's compare the pros and cons with ChatGPT

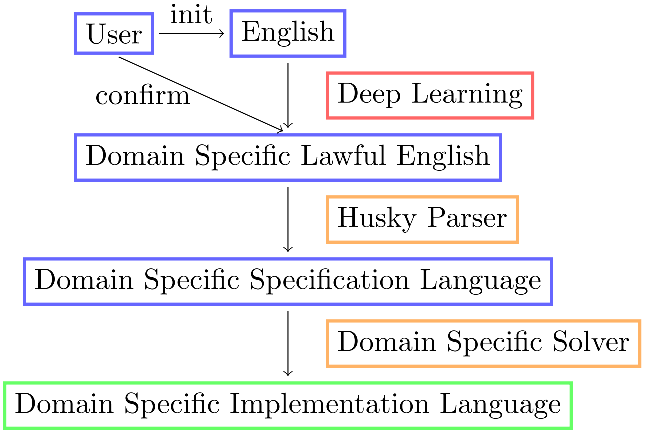
Here's a nice summary in table 10.1 written by the very ChatGPT itself.

I'm sorry that I have to copy because my hands are not in 100% health.

	ChatGPT	**DSLs**
Pros	Flexibility: ChatGPT can be used for a wide variety of NLP tasks, including text generation, question answering, summarization, and more.	Increased productivity: DSLs are designed to be easy to use and understand within a specific domain, which can increase developer productivity.
	Ease of use: ChatGPT can be accessed through simple API calls or integrated into other applications and platforms.	Enhanced control: A DSL provides more fine-grained control over a specific problem within a domain, allowing developers to tailor their solutions to specific requirements.
	Low development time: Implementing a ChatGPT-based solution typically requires less development time than creating a custom DSL.	Improved maintainability: DSLs can make code more readable and maintainable within a specific domain, as they typically use domain-specific terminology and concepts.
	Multilingual support: ChatGPT can generate text in multiple languages.	
Cons	Limited control: ChatGPT generates text based on its training data and can sometimes produce unexpected or incorrect output.	Higher development time: Developing a custom DSL can require a significant amount of time and effort.
	Lack of transparency: It can be difficult to understand why ChatGPT generates a particular piece of text, making it hard to debug or fine-tune.	Limited flexibility: A DSL is designed to solve a specific problem within a specific domain, so it may not be suitable for other tasks or domains.
	Dependence on data quality: The quality of ChatGPT's output is heavily dependent on the quality and diversity of its training data.	Steep learning curve: Developers may need to learn a new syntax and programming paradigm in order to use a DSL effectively.

Table 1: Comparison of ChatGPT and DSLs

We could have the best of both worlds by combining dsls with deep learning like this:


User
init
English
confirm
Deep Learning
Domain Specific Lawful English
Husky Parser
Domain Specific Specification Language
Domain Specific Solver
Domain Specific Implementation Language

First, we create a subset of English called Domain Specific Lawful English such as

(i) it's grammarly and semantically rigorous, so that parsing in Husky code is feasible (although it might be too difficult for traditional languages like C++/Rust/Python/Haskell)

(ii) it's expressive enough for the domain and can be faithfully translated into the Domain Specific Specification Language

(iii) easily understood by users

Second, we collect data and train a deep neural network that can translate arbitrary English into the subset.

When user initializes a dialogue using arbitrary English, it's translated into Domain Specific Lawful English, which is checked by the Husky Parser and also confirmed by the user that the translation is correct.

Then the Domain Specific Lawful English is translated by the Husky Parser into Domain Specific Specification Language so that it's acceptable by the solver.

Obviously, this setup will have the advantages of both worlds.

The details will be covered in the followup papers.

10.2 Stage 1: Universal Lawful English

Todo

10.3 Stage 2: Minimizing Usage of Deep Learning

Todo

10.4 Stage 3: Learning Augmented Verifiable Solver

Todo

10.5 Stage 4: Generating Human Readable Explanations

Todo