June 16th, 2020

Flexible Intelligence

by

Qianli Liao and Tomaso Poggio

Center for Brains, Minds, and Machines, McGovern Institute for Brain Research,
Massachusetts Institute of Technology, Cambridge, MA, 02139.

Abstract: We discuss the problem of flexibility in intelligence, a relatively little-studied topic in machine learning and AI. Flexibility can be understood as out-of-distribution generalization, and it can be achieved by converting novel distribution into known distributions. Such conversions may play the role of knowledge and is accumulated in the intelligent system, leading to human-like learning and generalizations.

This work was supported by the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF - 1231216.

Contents

1 Introduction **3**

2 Flexibility and Out-of-distribution Generalization **3**

3 Out-of-distribution Generalization via Reasoning: Reduction to Known Distributions **3**

3.1 An Algorithm to Reduce to Known Distributions 3

3.2 The Role of Knowledge Accumulation 4

4 Discussion: Traditional approaches to make AI general **4**

4.1 Train one model on many domains simultaneously 5

4.2 Modular Approaches 5

A Some Ideas of A General Problem Solving Algorithm **6**

B General Problem Solving as Information/Data Conversion **6**

C Step 1: Explain the Data **6**

C.1 Equivalent Conversions 8

C.2 Reduction Conversions (Unidirectional, Can be Lossy) 9

C.3 Algorithm: Explain the Problem Input and Output By Applying Conversions 9

D Step 2: Connecting Knowledge (Input and Memory) with Goals (Output) **10**

D.1 Action 1: Finding Explanation Tree by Recursively Checking Explainability 10

D.2 Action 2: Identify Correspondances and Merge Same Nodes 10

D.3 Action 3: Apply Higher-level Conversions 11

D.4 Repeat Action 1, 2 and 3 11

1 Introduction

Traditional machine learning tends to focus on maximizing the empirical and expected performance in learning one particular function with a certain distribution. The best guarantee one can make is that the learned model can work well on this input distribution. This makes traditional ML/AI models very rigid in the sense that it cannot tolerate any significant change in distribution in the input. They work well in one domain and often fail to exterpolate — they lack "out-of-distribution" generalization. This leads to the perception that traditional ML/AI models are "brittle" and they do not exhibit any "general intelligence".

While in general it may be unrealistic and even unfair to hope that the learned model can generalize to other distributions, we show in this report that there is an approach for it to generalize to a certain large class of distributions, leading to a "flexible intelligence".

This topic was also briefly discussed in our new dataset [Liao et al., 2020], which is designed specifically to tackle this problem.

2 Flexibility and Out-of-distribution Generalization

Most of the studies on generalization of ML models are concerned with "in-distribution" generalization — training and test examples come from roughly the same distribution.

Yet to achieve a any form of general intelligence, there should be a systematic way to handle the problem of "out-of-distribution" generalization.

A traditional way of obtaining "out-of-distribution" could be encoding prior knowledge (or very similarly the recent popular term "inductive bias") into the learning system. For example, in visual processing convolutional networks generalize to translations of the same object without training on all translations. Furthermore, one can incorporate prior knowledge about rotation and scaling and make the network invariant to those transformations [Liao and Poggio, 2017b].

In this report, we propose a different approach for out-of-distribution generalization.

3 Out-of-distribution Generalization via Reasoning: Reduction to Known Distributions

We propose a general and simple way of achieving out-of-distribution generalization: if one can find a way to reduce the test distribution to some distributions that are known to the system, then the problem becomes solvable.

One advantage of this approach is that it does not assume much about the test distribution: it just need be reducible to some known distributions by the system in finite steps.

3.1 An Algorithm to Reduce to Known Distributions

Let $\mathbb{D} \sim \mathbb{P}$ be the space of data following some distributions and $P_{known} \subseteq \mathbb{P}$ be the set of distributions known by the system. The question is how exactly a novel piece of data that follows a novel distribution $d_{new} \sim p_{new}$

can be reduced to a set of data following known distributions $\{d \sim p | p \in P_{known}\}$.

First the system needs to maintain a collection of reduction functions R that each function $r_k \in R$ will convert a piece of data following one distribution to a few pieces of data following other (often known) distributions $r_k : \mathbb{D} \sim \mathbb{P} \rightarrow \mathcal{P}(\mathbb{D} \sim \mathbb{P})$, where $\mathcal{P}(.)$ is the power set.

Then each reduction function r_k needs to have some way to be "triggered" by data d. So the algorithm starts with a single input $d_{new} \sim p_{new}$, some reduction functions are triggered by d_{new} and they convert d_{new} into a set of data D_2 following other distributions. Then a new set of reduction functions are triggered by each element in D_2. This will generate a new set of data D_3, each element following their own distributions.

This process is repeated many times to generate D_i for $i = 1...n$ until every element d in D_n follows some known distribution $p \in P_{known}$. To check if a data follows a known distribution, one just need a classifier or a set of standalone discriminators/detectors that can be either trained or hand-coded (only for simple distributions).

The whole algorithm is analogous to a reasoning process that converts a piece of novel/unknown data into a set of data that are already known by the system. Once that status is achieved, the system can invoke any existing functionalities/skills to process the known distributions to obtain the desirable information.

3.2 The Role of Knowledge Accumulation

In this algorithm, the system contains the following knowledge:

1. The collection of distribution reduction functions R and corresponding triggers
2. The set of known distributions P_{known} and corresponding detectors
3. The functionalities/skills associated with each known distribution in P_{known}

The first two collections of knowledge supports reasoning, handling and reduction of novel data distributions while the last collection of knowledge supports handling of known distributions.

With a clear definition of knowledge, this system has the potential of learning by expanding its knowledge library (e.g., adding functions to any of the three above collections).

With larger knowledge library, it should in principle perform better due to: 1. knowing how to convert between more distributions. 2. knowing more distributions 3. knowing more to do with a particular type of distribution.

Note that even a linear increase in any of the three knowledge categories can lead to the ability to deal with exponentially more number of new distributions due to the combinatorial effect of repeatly applying these knowledges.

This represents a little-studied yet important research direction: a systematic integration of knowledge, reasoning and traditional ML. The idea of learning with knowledge accumulation instead of tuning weights was also discussed in our 2017 paper [Liao and Poggio, 2017a].

4 Discussion: Traditional approaches to make AI general

In the last few years, we have witnessed superhuman AI models on individual domains. It is clear that if we restrict the task to a single domain, we can often achieve human-level performance given enough training data.

Researchers are not satisfied with a collection of independent AI models that each works well on one domain. They have been trying to develop a "general" intelligence such that a single model works well on a wide range of domains.

There are two approaches to this:

4.1 Train one model on many domains simultaneously

One obvious approach would be training one model on many domains simultaneously.

One issue is that by combining many tasks, one essentially constructs a single ultra-difficult problem for the system. In this setting, how to develop the right curriculum to train the system is also a difficult decision. And divide-and-conquer principle tells us that solving one difficult problem is not as efficient as solving its components.

Furthermore, when one combines tasks that are unrelated to each other to a single task, There is arguably a "curse of multitasking" effect analogous to "curse of dimensionality": Different domains can interfere with each other, leading to many problems like catastrophic forgetting, etc. Many domains are completely unrelated so that sharing the same model may lead to inefficiency and unnecessary difficulty in maintenance.

4.2 Modular Approaches

On the other hand, there is a "modular" approach that is more promising. People hope to develop a collection of modules that each works well on a single domain and then a controller that will decide which module to use for each scenario.

This avoids the "curse of multitasking" in each module. Yet there are a few significant open questions: 1. How to partition the world into a finite collection of tasks? 2. How to have a reliable controller that will make right decision on which module to use? A controller that decides everything suffers from the same "curse of multitasking" problem that a monolithic model suffers. 3. To what extent experiences are shared across modules and how?

References

[Liao et al., 2020] Liao, Q., Leonard, M., Pyo, B., Castillo, C., and Poggio, T. (2020). Universal format conversions.

[Liao and Poggio, 2017b] Liao, Q. and Poggio, T. (10/2017b). Object oriented deep learning.

[Liao and Poggio, 2017a] Liao, Q. and Poggio, T. (2017a). Human-like learning: A research proposal.

A Some Ideas of A General Problem Solving Algorithm

In the last decade, we have developed machines that can solve problems very well on many domains separately. They are often trained or engineered on each domain intensively to achieve this level of proficiency. However, they still do not exhibit any "domain-general" intelligence: that is, when facing a novel problem, they often cannot solve it. They lack out-of-distribution generalization. They do not have any "general problem solving" skills that humans apparently have.

In this appendix, we try to define what it means to solve a problem in general. We breakdown the process of solving a general problem into detailed, mathematically well-defined and machine executable steps. With this framework, theories and algorithms can be developed to solve general problems, especially the ones we proposed in [Liao et al., 2020].

B General Problem Solving as Information/Data Conversion

We can define general problem solving as an information or data conversion problem from input I to output O with memory M. For simplicity, we focus on training with only a few examples of input-output pairs $(I_1, O_1), (I_2, O_2), ..., (I_n, O_n)$.

The information conversion framework covers most of the tasks that an intelligent system can encounter: almost all tasks can in principle be solved by a process of:

1. **Read**: extract some information from the input
2. **Think**: convert the information in some way
3. **Write**: express the resultant information in some format

Note that all of these three processes may involve some memory M of the intelligent system.

In the following sections we show detailed steps of general problem solving that we believe are adopted unconciously by humans. We describe these steps in well-defined language so that they should be machine implementable.

C Step 1: Explain the Data

The first step of general problem solving is "making sense" of the data: it is a process of understanding the information and "explaining" the data. The data here include both the input and output.

Definition 1 *Explaining the data or "making sense" of the data is the process of conversion of any data* ***as much as possible into known distributions****.*

"Converting data into known distributions" means converting into a set of data each following a known distribution.

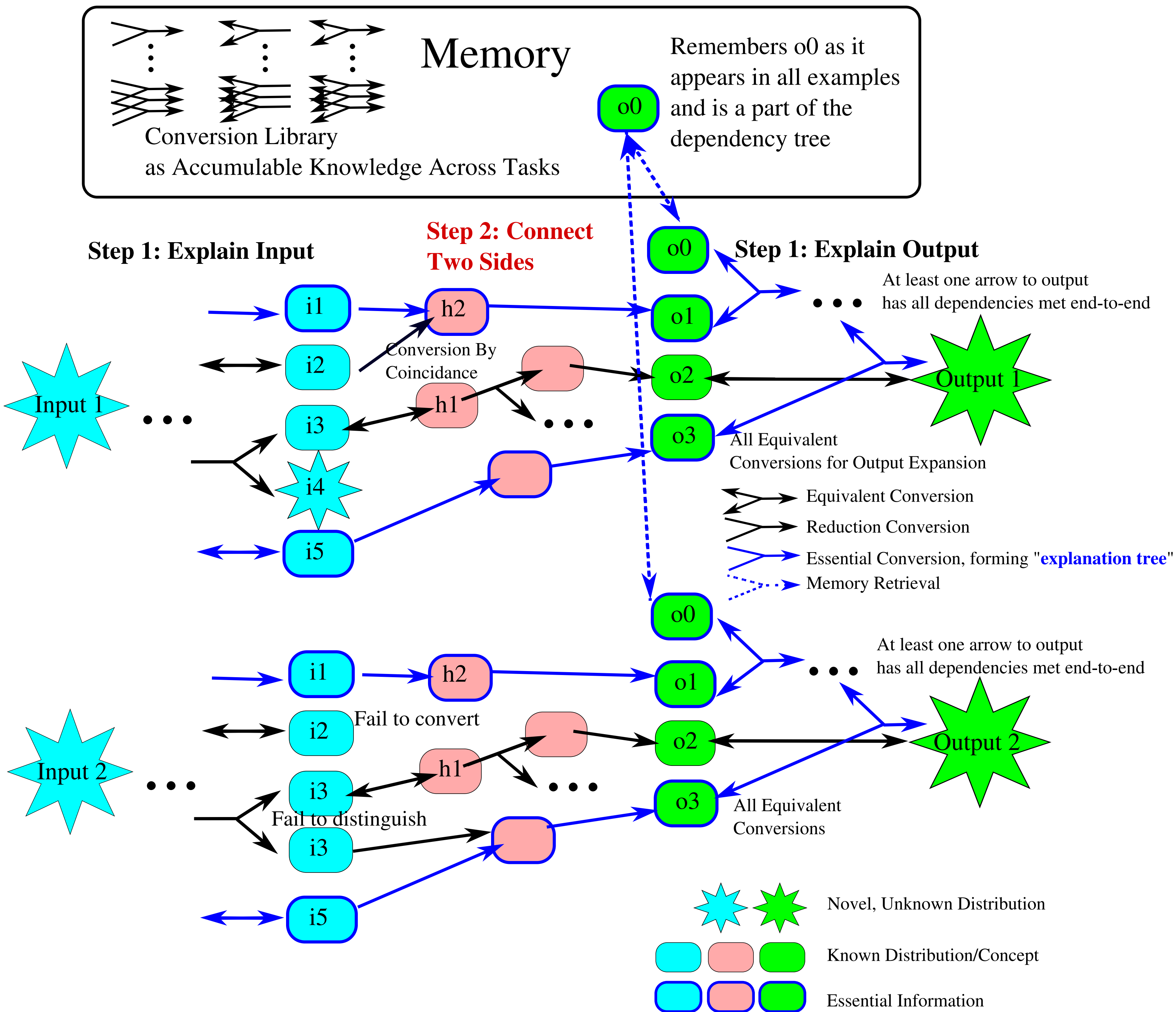

Figure 1: General Problem Solving as Information/Data Conversion. Given two input-output pairs. The goal of the algorithm is to figure out nodes (information) and arrows (conversions) to construct the output O from input I and memory M. During training, both input and output are expanded as much as possible into sets of data using the conversion library from memory. Then the algorithm finds at least one arrow to the output that is "fully explained" — all of its dependencies are satisfied end-to-end, traceable to either input or memory, forming a **explanation tree**. The same tree should be constructable from input and memory alone, generating the output. There could be multiple explanation trees, each corresponds to one candidate interpretation of the data. Explanation trees are verified across all training examples. As shown above, coincidental erroneous conversions and hard-to-distinguish nodes cannot exist in a explanation tree, thus eliminating less likely explanations.

Rule 1 *"As much as possible" indicates that this process can happen recursively until no more conversion can be done or some upper limit of repeatition is reached. If there are multiple conversions suitable for any intermediate data, all conversions are performed in parallel on that data: this is to get multiple "perspectives" on the same piece of data, which is useful in the case of ambiguity.*

All the conversions are stored in memory as a form of knowledge. It is not very efficient to loop over all conversions and try applying them. We should maintain some conversion "triggers" that are activated upon some tyeps/distributions of data.

There are two types of information conversions: **Reduction** and **Equivalence**.

C.1 Equivalent Conversions

One important type of conversion is equivalent conversion. In an equivalent conversion:

$$y_1, y_2, ... = E(I_1, x_2, x_3, ...) \tag{1}$$

The output set of data $y_1, y_2, ...$ has the same amount of information as $x_1, x_2, x_3, ...$. For every equivalent conversion, there is an inverse:

$$x_1, x_2, ... = E^{-1}(y_1, y_2, ...) \tag{2}$$

Example equivalent conversions:

- The conversion from a circle in 2D into a point in 2D and a radius
- The conversion between 56 and $8 * 7$
- The conversion between a 2D line segment and two 2D points
- The conversion between a object detection bounding box and the locations of the four boundaries (or equivalently the diagonal line segment)
- The conversion between a point in 2D and a pair of numbers (x and y).

Note that often there are multiple possible interpretations of a piece of data. For example a pair of values can not only correspond to a point in 2D, but also correspond to the height and weight of a person. Thus there exist multiple possible equivalent conversions for each piece of data. Which specific conversion is useful in solving a problem is context dependent and need to be determined on a case-by-case basis.

This also shows that information-wise the height and width of a person is "equivalent" to a point in 2D. With this type of conversions, the system can in principle figure out how to plot many peoples' physical information in a 2D plot for visualization.

In real life, there are "roughly equivalent" conversions. For example, an autoencoder can encode the picture of a cat into a latent code. This code can reconstruct the picture of cat through the decoder. But depending on how well the autoencoder is trained, there are noises in the encoded code and reconstructed picture. Whether this conversion can be considered conceptually as an equivalent conversion depends on whether the user/specification accepts this level of approximation.

There are also “contextually equivalent” conversions. In some context as far as the problem is concerned, a picture of a cat can be considered as equivalent to a word of “cat”. This may require temporary construction of some equivalences. We will study this more in future work.

C.2 Reduction Conversions (Unidirectional, Can be Lossy)

In addition to equivalent conversions, many types of data conversions are lossy in the sense that they selectively extract some information from the data and discard some other informations.

Example reduction conversions:

- an object detector that only detects target objects in the data and ignores the rest of the image.
- an image classifier or an attributes detector that only output corresponding labels of interest and do not retain other information in their outputs.

Almost all unidirectional conversions can be understood as reductions, because the amount of information cannot increase by going through a function. In the case of generative models, the conversion function takes in extra arguments from the memory (e.g., weights) of the system.

C.3 Algorithm: Explain the Problem Input and Output By Applying Conversions

In the case of general problem solving, it is necessary to understand the **output** of a problem using only **equivalent** conversions, since the goal is to be able to reconstruct the output.

Let O be the output of a particular training example. By applying a set of **equivalent** conversions C_e repeatedly k times until no more conversion can be done. We have a set of “explained” data $O_e = o_1, o_2, ..., o_n$ following some known distributions.

$$O_e = o_1, o_2, o_3, ..., o_n = C_e^k(O) \tag{3}$$

Since C_e is a set of equivalent conversions, one can reconstruct exactly O with O_e. In fact, usually a subset of O_e should suffice due to the redundancies in O_e introduced by Rule 1.

The **input** of a problem can be understood with either **reduction** or **equivalent** conversions or both, since we only need to extract relevant information that is enough to construct the output.

Let I be the input of a particular training example. By applying a set of **equivalent** and **reduction** conversions C_{er} repeatedly j times until no more conversion can be done. We have a set of “explained” data $I_e = i_1, i_2, ..., i_m$ following some known distributions.

$$I_e = i_1, i_2, i_3, ..., i_m = C_{er}^j(I) \tag{4}$$

D Step 2: Connecting Knowledge (Input and Memory) with Goals (Output)

In step 1, we applied conversions to the input and output to get "explained" input and output data I_e and O_e.

Note that O_e is not yet fully explained in the sense that they are not supported by input I_e and memory M.

In some sense we have just finished the "parsing" stage. The step 2 corresponds to the "thinking" stage described in Section B.

There are a few actions the system needs to perform in order to connect I_e and O_e.

D.1 Action 1: Finding Explanation Tree by Recursively Checking Explainability

We define a function $EXPLAINED()$ that checks if a node is "explained" and record possible explanations if **true**. Otherwise it returns **false**. It looks at all the conversions that leads to this node and for each conversion it recursively apply $EXPLAINED()$ on all of its inputs. If all inputs to this conversion are "explained", the conversion is explained. If any of the conversions to this node is "explained", this function returns true. Otherwise, it return false.

```
EXPLAINED(node)
 1 if Is Input or Memory Node then
 2     return True
 3 else
 4     conversions = node.conversions ;          // conversions that lead to this node
 5     if conversions is EMPTY then
           /* it is a leaf but not input or a node in memory                    */
 6         return False
 7     hasExplanation = False
 8     node.explanations = list()
 9     for C in conversions do
           /* check all conversions that leads to this node                     */
10         conversionExplained = True
11         for inp in C.inputs do
               /* recursively checking if the input node is explained           */
12             if EXPLAINED(inp) is False then
13                 conversionExplained = False
14         if conversionExplained is True then
               /* if all inputs of a conversion are explained                   */
15             node.explanations.append( C)
16             hasExplanation =True
17     if hasExplanation is True then
18         return True
19     else
20         return False
```

D.2 Action 2: Identify Correspondances and Merge Same Nodes

At this point, many pieces of data may directly correspond to each other in I_e and O_e. For every element in O_e, we can apply a search in I_e for essentially the same information. If we find a match, we can merge these nodes

into one.

D.3 Action 3: Apply Higher-level Conversions

In principle I_e and O_e are already sufficiently expanded due to previous repeated application of conversions. Yet there is a possibility that some conversions are only useful for high level reasoning and are reserved for Step 2. For example, when solving a math task, all the theorems should be applied only in Step 2 since they do not really affect the understanding of the input and output.

For efficiency reasons, we should make most of the conversions triggerable by data instead of randomly applying them to all data.

D.4 Repeat Action 1, 2 and 3

Finally, the system performs above actions in alternation until all explanation trees are found.