CENTER FOR
Brains
Minds+
Machines

June 2nd, 2020

Universal Format Conversions

by

Qianli Liao[1] ,Matthew Leonard, Bryan Pyo, Cristian Castillo and Tomaso Poggio

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

Abstract:

Information is the fuel for intelligence. Any competitive intelligence system should be information hungry. "Formats" on the other hand, is the container for information. Accessing information without the ability to decipher its format is like drinking without a container.

From this perspective, current machine learning systems arguably have zero general information processing capability because it cannot really handle information that is presented differently from their standard input format, even if the changes are completely trivial and regular. Humans however have no trouble understanding reasonable changes at all. Thus, there is an unexplored research area to make machines understand formats in a flexible way like humans do. As the first step in this direction, we propose a task called Universal Format Conversions (UFC): a task that is designed to test a system's ability to understand formats and convert between any formats of data by just observing a few examples. This requires an intelligent system to extract useful information ("read") and convey knowledge ("write") with novel data structures and text with minimal training, leading to the ability to "communicate" flexibly in the format of structured data, artificial expressions and even natural language. Furthermore, we note that an ideal intelligent system should go beyond working with pairs of formats — it should discover interesting information by only looking at one format, namely possessing a zero-shot pattern discovery ability. Finally, solving UFC would directly lead to real-world breakthroughs in programming since enormous amount of time of all programmers is spent on converting all types of ad hoc data structures and formats.

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

[1]Q.L. started and designed the project and wrote the paper. The other authors participated in the research and also contributed to the dataset.

Contents

1 Introduction

Our world is full of information, often organized and stored in different ways that can be called "formats" or "structures". A capable intelligent system should be good at processing information, which implies that it should be able to understand different formats, either seen or unseen, in a flexible and robust way. However, current mainstream Machine Learning (ML) and Artificial Intelligence (AI) systems are extremely weak in this aspect: they are designed to understand a single designated input data format — either being a 4D matrix of images stacked in a certain way or a list of strings from a certain vocabulary etc.

Current ML/AI systems can be broken by slightly changing the input format: if the data matrix is transposed or the input is shuffled in a predictable way (at test time), the model has no way to realize the change and recover from it. For example, even with the least restrictive system, say a character-based language model, I can add a pound sign # every 3 characters in the input and the model will get confused. Such input format manipulation has endless possibilities while humans seem to be able to adapt to most of them without any training.

To address this shortcoming of AI systems, we propose a task that is designed to promote flexible and robust format understanding. There could be many approaches towards this goal, but one basic direction could be solving the task we call Universal Format Conversions (UFC): given two pieces of data in two formats, the system is required to figure out a way to convert from one to the other. There will be only a few examples of each data since humans usually do not require many examples to make sense of a piece of structured information.

The UFC tasks directly probe the problem of "flexible intelligence": a system needs to extract useful information ("read") and convey knowledge ("write") with endless possibilities of information formats without being trained on most of them beforehand.

One practical benefit of solving our task is that since format conversions are ubiquitous in the area of programming, successful automated UFC may constitute a major breakthrough in automated programming and could save people immeasurable amount of time.

Format Conversion (FC) is also a very general framework such that many tasks can be formulated as a problem of FC (often with memory, see Figure 3 in Discussion). A powerful UFC solver may solve many other problems off-the-shelf.

An extreme yet interesting form of FC is decryption and cracking code. Secret messages are sent in a format that is deliberately designed to be difficult to figure out. Yet if an agent is intelligent enough (like Alan Turing), he/she can find a way to understand it. From this perspective, UFC tasks are like simple code cracking tasks.

2 Flexible Intelligence

Humans are flexible with formats: one key difference between humans and current AI systems is that humans are much better at taking information from any arbitrary formats and make sense of it while current AI systems are easily broken by any small change in input format. Humans can quickly figure out unseen format as long as they can be reasoned with in some way. Humans are also extremely good at expressing their ideas in different formats, adapting to any particular need based on the context.

Current computer programs, including all ML and AI systems are so rigid in the sense that they cannot tolerate any simple changes in their input format. As ML algorithms outperform humans in different tasks in recent years, they start to saturate in the dimension of performance (Figure 1). We argue that it is the time to pursue an highly overlooked dimension of intelligence: flexibility.

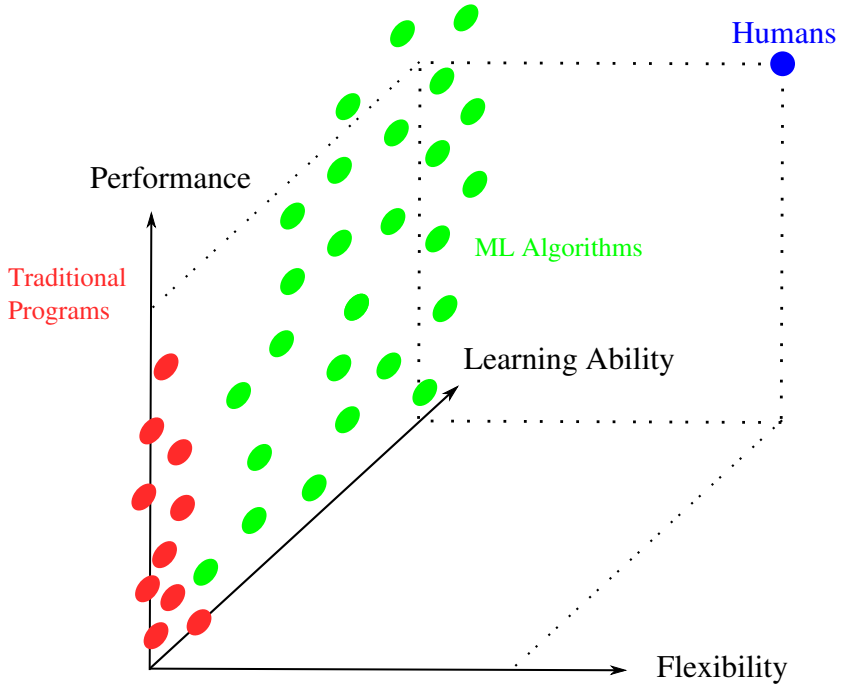

Figure 1: Humans are arguably much more flexible than traditional computer programs and ML algorithms. Note that the ability to learn should be orthogonal to flexibility.

3 Definition of Flexibility and Formats

One definition of a system being flexible is that it can understand a piece of information being the same no matter how it is presented. The way a piece of information is presented is basically the definition of a "format".

Thus, training the system to convert a piece of information from one format to another will ultimately force it to extract the relevant information into some internal representation and then express it in the desirable way.

If a system develops the ability to convert between a novel pair of formats by just observing a few examples of each, it arguably possesses human-like ability to "figure out" the structure of novel data, a hallmark of human intelligence.

This ability is essential in human's flexibility: humans never expect any certain data format when reading books and communicate in general — we can always "figure them out" upon encounter.

4 Dataset Organization

We currently have four types of conversions in our dataset:

4.1 Text to text

We show in Listing 1 and 2 an example of text to text format conversion. This task takes in a Python code with a docstring and automatically convert it into a HTML documentation file. We show the rendered HTML file in Figure 2.

```
def complicated_function(a, three, source, destination):
        """
        complicated_function: given a source list, add three and subtract
            a from each item and copy the item to the destination if the
           new value is odd.
        @parameters:
                a: a number to be subtracted from each value in the
                    source list
                three: a number to be added to each value. Should always
                    be set to the number 3
                source: the source of the values. Is not modified.
                destination: a reference to a list we will copy into. Is
                    modified.
        @returns: the number of values that were not copied into the
            destination list.
        """
    discarded = 0
    index = 0
    for s in source:
        newitem = s + three - a
        if (newitem % 2 == 1):
            destination[index] = newitem
            index += 1
        else:
            discarded += 1
    return discarded
```

Listing 1: Text to Text Example: Automatic Documentation Input

```
<h1>
complicated_function:
</h1>

<p> given a source list, add three and subtract a from each item and copy
    the item to the destination if the new value is odd. </p>

<h2>Parameters:</h2>
<table>
<tr>
        <th>Variable</th><th>Description</th>
</tr>
<tr>
                <td>a</td> <td>a number to be subtracted from each value
                    in the source list</td>
</tr>

<tr>
                <td>three</td> <td>a number to be added to each value.
                    Should always be set to the number 3</td>
</tr>

<tr>
                <td>source</td> <td>the source of the values. Is not
                    modified.</td>
</tr>
<tr>
                <td>destination</td><td>a reference to a list we will
                    copy into. Is modified.</td>
</tr>

</table>
<h2>
Returns:
</h2>
<p>the number of values that were not copied into the destination list.</
   p>
```

Listing 2: Text to Text Example: Automatic Documentation Output

4.2 Text to data structure

We show in Listing 3 and 4 a simple example of text to data structure format conversion. The rules should be simple to figure out.

complicated_function:

given a source list, add three and subtract a from each item and copy the item to the destination if the new value is odd.

Parameters:

Variable	Description
a	a number to be subtracted from each value in the source list
three	a number to be added to each value. Should always be set to the number 3
source	the source of the values. Is not modified.
destination	a reference to a list we will copy into. Is modified.

Returns:

the number of values that were not copied into the destination list.

Figure 2: Rendered HTML from the output of one text to text task (Listing 2).

```
bbbyybbbbwwwwwjooooossddddduuufffffrrrrrebbbbhhhhhlllfff
```

Listing 3: Text to Data Structure Example: Count Repeats, Input

```
output = [('b', 3), ('y', 2), ('b', 4), ('w', 5), ('j', 1), ('o', 5), ('s
    ', 2), ('d', 5), ('u', 3), ('f', 5), ('r', 5), ('e', 1), ('b', 4), ('h
    ', 5), ('l', 3), ('f', 3)]
```

Listing 4: Text to Data Structure Example: Count Repeats, Output

4.3 Data structure to data structure

We show in Listing 5, 6, 7 and 8 a training and a test example of a data structure to data structure FC task. Note that in the test example, the matrix could be larger than that in any training example. This level of flexibility should be achieved by the model. Because if it figures out the correct rules, it would not rely on simple pattern matching and correctly exterpolate to larger inputs.

```
inp= [[ 0.9957,  0.7257, -1.6085,  1.0061],
      [-0.3867, -0.3575, -0.4872,  1.3073],
      [-0.1609, -0.4364,  0.8864,  0.9320],
      [-0.8357, -0.2334, -0.5085, -1.0113]]
```

Listing 5: Data Structure to Data Structure Example (Training): Input

```
class tensor:
    def __init__(self, value ):
        self.value = value
output =[(tensor([0, 1]), tensor(0.7257)),
         (tensor([0, 3]), tensor(1.0061)),
         (tensor([1, 3]), tensor(1.3073)),
         (tensor([2, 2]), tensor(0.8864)),
         (tensor([2, 3]), tensor(0.9320))]
```

Listing 6: Data Structure to Data Structure Example (Training): Output

```
inp=[[ 1.4090,  1.9947,  0.6410, -0.8073, -0.6047,  1.3098],
     [-0.5438,  0.1374, -0.0969,  0.4975,  0.4598,  0.6280],
     [-0.1921, -0.4329, -0.2485,  0.3116,  0.1823, -1.4685],
     [-2.8312, -1.0795, -1.0862,  1.0134, -1.1572,  0.3476],
     [ 0.7342,  0.5649,  0.0919, -0.2754,  0.6251, -0.4055],
     [ 1.6439, -1.8316,  0.1369, -1.7436,  1.8454,  0.7847]]
```

Listing 7: Data Structure to Data Structure Example (Testing): Input

```
class tensor:
    def __init__(self, value ):
        self.value = value
output=[(tensor([0, 1]), tensor(1.9947)),
        (tensor([0, 2]), tensor(0.6410)),
        (tensor([0, 5]), tensor(1.3098)),
        (tensor([1, 1]), tensor(0.1374)),
        (tensor([1, 3]), tensor(0.4975)),
        (tensor([1, 4]), tensor(0.4598)),
        (tensor([1, 5]), tensor(0.6280)),
        (tensor([2, 3]), tensor(0.3116)),
        (tensor([2, 4]), tensor(0.1823)),
        (tensor([3, 3]), tensor(1.0134)),
        (tensor([3, 5]), tensor(0.3476)),
        (tensor([4, 0]), tensor(0.7342)),
        (tensor([4, 1]), tensor(0.5649)),
        (tensor([4, 2]), tensor(0.0919)),
        (tensor([4, 4]), tensor(0.6251)),
        (tensor([5, 0]), tensor(1.6439)),
        (tensor([5, 2]), tensor(0.1369)),
        (tensor([5, 4]), tensor(1.8454)),
        (tensor([5, 5]), tensor(0.7847))]
```

Listing 8: Data Structure to Data Structure Example (Testing): Output

4.4 Data structure to text

We show in Listing 3 and 4 an example of data structure to text conversion. In this conversion, a data structure representing poker cards is converted to a text representation.

```
# Card.py
class Card:
    def __init__(self, suit, number):
        self.suit = suit
        self.number = number

# Input.py
import Card
input = [Card.Card("hearts",8), Card.Card("clubs",12), Card.Card("hearts"
    ,13), Card.Card("spades",8)]
```

Listing 9: Data Structure to Text Example: Generating Poker Cards, Input

```
/------------\
HHHHHHHH
\------------/
/------------\
CCCCCCCCCCCC
\------------/
/------------\
HHHHHHHHHHHH
\------------/
/------------\
SSSSSSSS
\------------/
```

Listing 10: Data Structure to Text Example: Generating Poker Cards, Output

4.5 Availability

We currently have created hundreds of format conversion tasks and the number is growing. The dataset is available at `https://github.com/liaoq/format_conversions`.

5 Discussion

5.1 Flexible and Robust with Formats Leads to Efficient Learning and Problem Solving

From a learning perspective, we argue that being flexible and robust with formats would probably be the first step towards a really powerful intelligent system. Because this system would be able to process arbitrarily structured data from any source, without requiring laborious manual annotations and data conversions, which will lead to aboundant amount of resources for learning.

5.2 UFC Represents Real-world Problems

One great feature of UFC is that it is not a toy task — it represents real-world problems that people encounter everyday:

1. When reading a new book, one unconciously figures out the 2D layout of the book and correctly convert it into a 1D text stream.
2. When checking emails, menus and bank statements, one understands the meanings of any novel tables and extract relevant information without being trained on them
3. When solving puzzles like a crossword on a newspaper, the meaning of character alignments manifests itself.
4. When looking up words in a new dictionary, many symbols and indexing numbers are understood without explaination.

and for programmers:

1. One can convert one type of machine learning data to a type that can be consumed by the model in mind
2. One can write a web crawler on a website's HTML source to get desired information without the website owner explaining the website to you.
3. One can read compilation output of C++ and extract the error messages and related information

5.3 Traditional ML Tasks Can also be Viewed as Format Conversions

Figure 3 shows that traditional discriminative and generative models can be seen as some types of format conversions. They are often continuous format conversions that can be trained with gradient descent. Most of the real-world tasks beyond sensory perception are discrete and combinatorial and we find them more challenging for current ML tools.

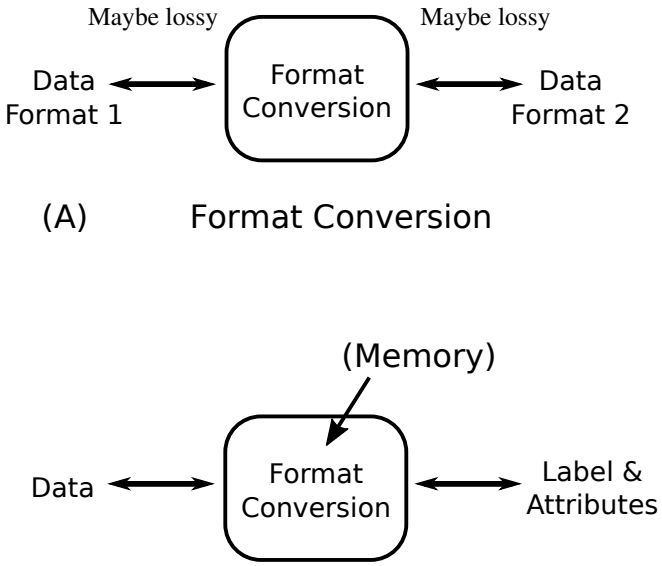

Figure 3: Traditional discriminative and generative models can also be understood as format conversions. Yet they are often continuous. Our tasks, including the most typical definition of format conversions (e.g., Word to Excel, .csv to .json, matrix to table, etc.), are more discrete and combinatorial.