Computationalism

Introduction to Cognitive Science

Computationalism

- Cognition can be defined in terms of information-processing:
  - Perception is taking in information from the environment
  - Memory/Beliefs/Knowledge is storing information
  - Reasoning is inferring new information from existing information
  - Planning is using information to make decisions
  - Etc.
- Information-processing can (only?!) be done through computation
- Therefore, cognition can (only?!) be achieved (implemented/realized) through computation.

Computationalism and Predictions about Aliens

- Notice that the argument on the previous slide is a purely *conceptual* one in that it is not based on any empirical evidence.
- In fact (assuming we fill in the ‘only’ part on the previous slide), it *predicts* the existence of some kind of computer in (behind) any kind of cognitive being.

Computationalism and the Brain

- Our brain is a computer
  - We will see some arguments for this claim in the rest of the presentation
- So, the fact that we have a brain can be seen as empirical *confirmation* of (and thus *evidence for*) the view of computationalism.
  - Indeed, we know that the nature of the mind changes when the brain changes (neural dependency). Thus, maybe (although I'll reject this analogy later!!!!):
    - brain = 'hardware'
    - mind = 'software'

The Brain is a Computer, Part I

- The brain is unlike any other organ; the heart, lungs, liver, etc. all do something very much physical: they collect, filter, pump, etc.
- The brain, however, is quite different: Its function seems to be to take in signals, and send out signals, in communication with the nervous system.
- But, as such, the brain seems to be an information-processor: a computer.

A COMPUTER WANTED.

WASHINGTON, May 1.—A civil service examination will be held May 18 in Washington, and, if necessary, in other cities, to secure eligibles for the position of computer in the Nautical Almanac Office, where two vacancies exist—one at $1,000, the other at $1,400..

The examination will include the subjects of algebra, geometry, trigonometry, and astronomy. Application blanks may be obtained of the United States Civil Service Commission.

The New York Times
Published: May 2, 1892
Copyright © The New York Times

Algorithms

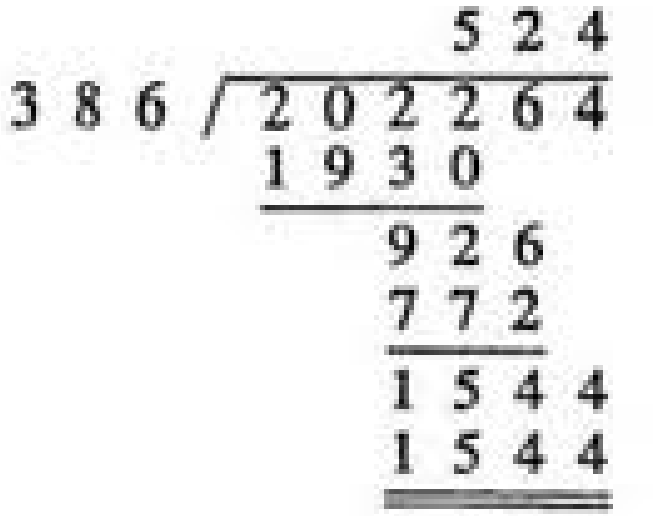
- An algorithm is an effective step-by-step procedure:
  - During each step, we perform some operation, and move on to some other step
  - *Effective*: A ‘normal’ human being can ‘follow’ the algorithm, i.e. at each point it is exactly clear what action to perform, this action is simple enough for us to perform, and after each step it is clear which step to take next
  - Usually, we require that there be some end to this process.

- Examples:
  - Cookbook recipe
  - Filling out tax forms
  - Furniture assembly
  - Long division

Computations

- A computation is a kind of algorithm: a computation is a symbol-manipulation algorithm.
  - The symbols represent something
  - Hence, the computation is about that something: we compute something
  - Example: long division.
- Not every algorithm is a computation
  - Example: furniture assembly instructions

Example: Long Division



Computers

- A ‘computer’ is something that computes, i.e. something that performs a computation.
- Between the 17th and 20th century, a ‘computer’ was understood to be a human being; humans who computed things!
- It was only by automating (mechanizing) this process, that we obtained ‘computers’ as we now think of them.

The Scope and Limits of Computation

- In 1936, Alan Turing wrote a paper in which he tried to give an answer to the decision problem in formal logic: is there a way to decide, for any arguments expressed in formal logic, whether it is valid or not?
- Turing thought the answer to the decision problem was negative: that there is no procedure to tell all valid arguments from all invalid arguments.
- To prove this, Turing needed to generalize over all possible computations.
- Turing tried to do this by finding basic elements to which we can reduce any such process.

States and Symbols

- Take the example of multiplication: we make marks on any place on the paper, depending on what other marks there already are, and on what ‘stage’ in the algorithm we are (we can be in the process of multiplying two digits, adding a bunch of digits, carrying over).
- So, when going through an algorithm we go through a series of stages or states that indicate what we should do next (we should multiply two digits, we should write a digit, we should carry over a digit, we should add digits, etc).

A Finite Number of Abstract States

- The stages we are in vary between the different algorithms we use to solve different problems.
- However, no matter how we characterize these states, what they ultimately come down to is that they indicate what symbols to write based on what symbols there are.
- Hence, all we should be able to do is to be able to discriminate between different states
  - what we call them is completely irrelevant!
- Moreover, although an algorithm can have any number of stages defined, since we want an answer after a finite number of steps, there can only be a finite number of such states.

A Finite Set of Abstract Symbols

- Next, Turing pointed out that the symbols are abstract as well: whether we use ‘1’ to represent the number 1, or ‘☺’ to do so, doesn’t matter.
- All that matters is that different symbols can be used to represent different things.
  - What actual symbols we use is irrelevant!
- Also, while we can use any number of symbols, any finite computation will only deal with a finite number of symbols. So, all we need is a finite set of symbols.

A String of Symbols

- While we can write symbols at different places (e.g. in multiplication we use a 2-dimensional grid), symbols have a discrete location on the paper. These discrete locations can be numbered.
- Or, put another way: we should be able to do whatever we did before by writing the symbols in one big long (actually, of arbitrarily long size) string of symbols.

Reading, Writing, and Moving between Symbols

- During the computation, we write down symbols on the basis of the presence of other symbols. So, we need to be able to read and write symbols, but we also need to get to the right location to read and write those symbols.
- With one big long symbol string, however, we can get to any desired location simply by moving left or right along this symbol string, one symbol at a time.

Components for Computation

- Turing thus obtained the following basic components of effective computation:
  - A finite set of states
  - A finite set of symbols
  - One big symbol string of arbitrary size
  - An ability to move along this symbol string
  - An ability to read a symbol
  - An ability to write a symbol
- We call this: a Turing-machine

Turing Machines Demo

Computable Functions

- We can use a Turing-machine to compute the sum, and product, of any two numbers.
- These functions are therefore Turing-computable
- Lots of other functions are Turing-computable
- E.g. all functions needed to run Microsoft Word are Turing-computable (i.e. you can run Microsoft Word on a Turing-machine)

The Church-Turing Thesis

- If a computer of type X can compute a function f, we say that f is X-computable
- The Church-Turing Thesis:
  - No matter what type of computer X you have: All functions that are X-computable are Turing-computable.
- In short: *Turing-machines can compute anything that is computable.*

0’s and 1’s

- Turing showed how all computation can be done using a limited number of simple processes manipulating a small number of symbols.
- Some years later, Claude Shannon showed that all effective computations can be performed through the manipulation of bitstrings (strings of 0’s and 1’s) alone.
- You do need *lots* of these 0’s and 1’s, and you do need to perform lots of these simple operations.
- But this is exactly how the modern ‘digital computer’ does things. That is, at the ‘machine level’, it’s all simple manipulations of 0’s and 1’s.

Physical Dichotomies

- The 0’s and 1’s are just abstractions though; they need to be physically implemented.
- Thus, you need some kind of physical dichotomy, e.g. hole in punch card or not, voltage high or low, quantum spin up or down, penny on piece of toilet paper or not, etc.

The Brain is a Computer, Part II

- These theoretic results in information-processing show that one can obtain powerful information-processing capacities using very simple resources … as long as you have lots of them. Well:
  - Our brain has ~10^{11} neurons
  - Our brain has ~10^{14} neural connections
  - Thus, a neuron firing or not could constitute 0's and 1's, and a neuron's firing as a function of connected neurons' firing, could implement the necessary operations needed for computation.

Causal Topology

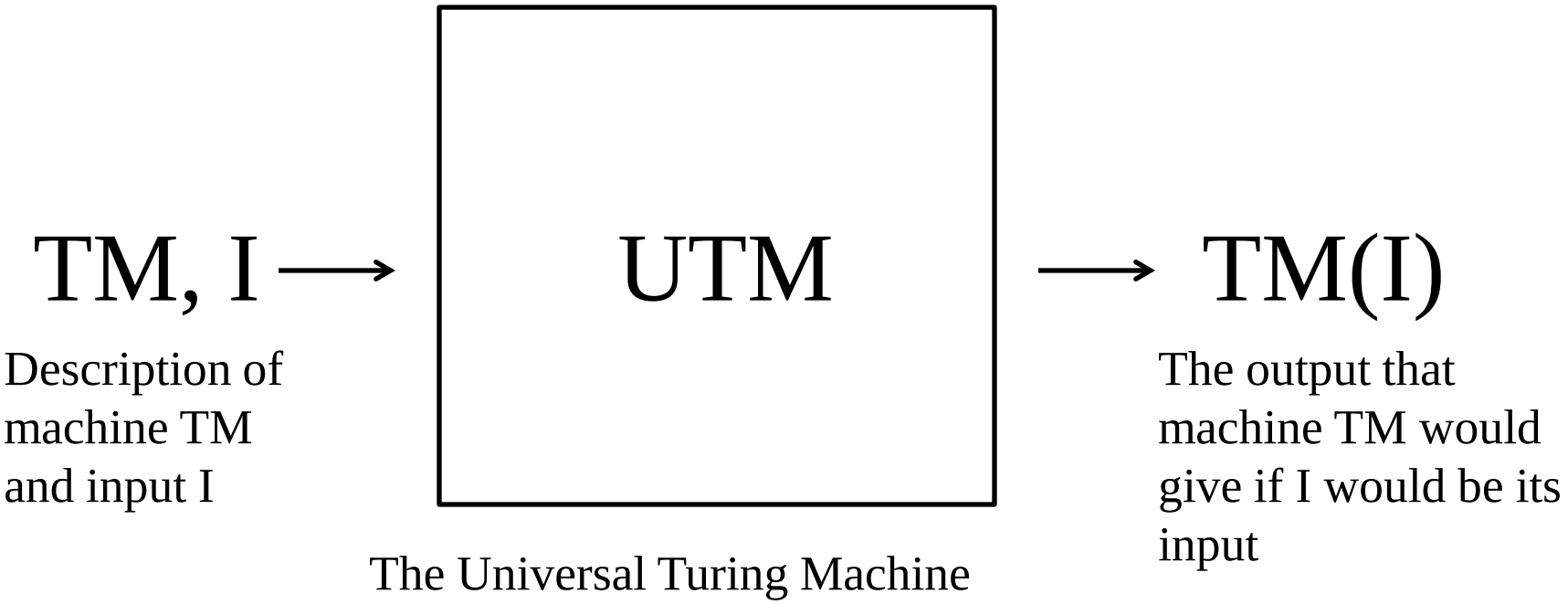
- A physical system implements a computer program if and only if that system implements a certain causal topology.
- This topology is highly abstract. As long as you retain the functionality of the parts, and the connections between the parts, you can:
  - Move parts
  - Stretch parts
  - Replace parts
- This is why there can be mechanical computers, electronic computers, DNA computers, optical computers, quantum computers, etc!

The Brain is a Computer, Part III

- Again, it seems to fit:
  - The brain implements a complex causal topology where the only thing that seems to matter is how the neurons are connected, not where they are located, what physical material they are made of, how big they are, how they are shaped, etc.
  - E.g. In case of neural defects, other areas of the brain can take over their original task

Universal Turing Machines

Turing proved that there exists a Turing-machine that can emulate any other Turing-machine



Program as Data
Hardware and Software

- Universal Machines take in programs as data
  - Your laptop is a Universal Machine. It runs programs.
- Clear distinction between hardware and software:
  - The (universal) machine is hardware
  - The program is software
  - The machine will function differently depending on the software that is ‘fed into’ and ‘run on’ it
  - Without software running on the hardware, the hardware does nothing of interest
  - The pieces of software are portable: you can take the software and put it on a different (universal) machine

Mind :: Brain = Software :: Hardware?

- This analogy appeals to the idea of physical reductionism:
  - Minds (or aspects of the mind) exist as high-level abstractions of a working brain
  - (somewhat) Similarly, a program (a piece of software) provides an abstract description of the functionality of a machine running that program
- However, this is as far as the analogy goes.
  - As the next few slides show, there are some very important dis-analogies between ‘mind’ and ‘software’ (as well as between ‘brain’ and ‘hardware’)

Where the Analogy Breaks Down

- First of all, even if the program is written in a high-level programming language, it may still not provide the level of abstraction we use when talking about (and wanting to explain) the behavior of a machine
  - E.g. If the computer makes a mistake, there is usually no line in the program that says something equivalent to “make mistake now”. And even when things function just right, the behavior of the working computer as a whole is at a higher level of description than the program.
  - Similarly, most computationalists state that computations are what *underlie* the mind: we can *explain* mental properties by pointing to computational properties as physically realized (implemented) by the brain, but minds still exist at a higher level of abstraction than those computational descriptions.

Where the Analogy *Really* Breaks Down

- The brain is not a piece of hardware that ‘runs’ a mind
  - There is nothing similar in the brain to a ‘CPU’ that fetches and then executes one of an explicit set of instructions
  - The brain is not some ‘blank slate’ that’s just ‘sitting there’ when there is no ‘mind’ running ‘on’ it
- The mind is not a piece of software you ‘stick’ into a brain
  - It is not a piece of data; it is not symbol strings
  - It doesn’t describe the abstraction; it *is* the abstraction!
- There is no clear separation. Despite what movies depict:
  - Brains don’t ‘take in’ a mind ... or some other mind
  - Minds aren’t portable entities you can stick into any brain you want. In fact: no brain -> no mind!

Programming and Reconfiguring

- The brain does reconfigure
  - Neural connections get added or removed
  - More or less neural resources can be devoted to tasks
- These changes effect what happens at the higher-level of abstraction where we find ‘minds’
  - So the mind does not provide instructions to the brain saying how it ought to function, but rather is the abstraction that results from a working brain
- Outside stimuli (such as learning environments) may be the ultimate cause for these changes
  - Hence, if the brain is ‘programmed’ at all, we can say that it is programmed by its environment. But again, that is not at all what a mind is: Mind ≠ Software = Program = Instructions

Summary

- Two independent arguments for computationalism:
  - One conceptual: cognition is information-processing, and that's exactly what computers do
  - One empirical: the mind seems dependent on the brain, where the brain seems to be implementing a causal topology consisting of a large numbers of simple devices capable of supporting complex information-processing capacities