Artificial Intelligence Lecture 2
Symbolic and Connectionist Machines

Dr. Manuel Pita

Quick Summary of Lecture 1

In the previous theory lecture we spoke about the history of AI – focusing on the different ways people have tried to define it since its inception in the 1950s. We started with the approach based on (1) *human-like action,* embodied as the *Turing test*. We saw that the Turing test does not provide an instrumental working definition of AI. However, it brought a lot of philosophical discussion that is still active today. You will find amongst the class materials a PDF document written by Katrina LaCurts from MIT that goes deeper into the criticisms of the Turing test for deciding whether a machine is intelligent or not. We then moved onto the ancient Greek school of Logic, with the (2) *rational thinkers,* which provided AI with two essential elements for – at least – some kind of machine intelligence: symbols and algorithms. However, we know that a lot of what intelligence is does not rely only on manipulating symbols, or using Logic. Think for example about how we learn to ride a bicycle. In this kind of situation our intelligence is capable of using the body to learn on its own, rather than by relying on a recipe given in the form of symbols. In fact, we (humans) learn both ways. We followed this by looking at the history of psychology and cognitive science referring to this as the (3) *thinking humanly* corner, the key aspect of which is the cognitive revolution of the 1960s. This revolution replaces the old paradigm that human intelligence is based on stimuli/responses. A new idea emerged positing that the brain is not just a stimulus-response connection machine, but that using neural connections it is capable of *processing information*. We ended in the (4) *rational action* perspective, defining AI in terms of *rational agents* that have sensors to get information from the environment, effectors to act on the outside world and between sensors and effectors, some architecture to process incoming information and determine the actions it will execute to achieve a predefined goal. This pragmatic definition is the one we will use for the rest of this course. This was followed by (1) an introduction to the Turing Machine as one of the root forms of artificial intelligence that implements symbol-manipulating machines; and (2) the neural networks of McCulloch and Pitts as the root form of AI based on non-symbolic interconnected artificial networks of neurones that can send electrical pulses – similar to those we find in real brains. In this lecture we further elaborate on these kinds of machine.

Turing Machines

As we discussed in the first lecture of this course, the Turing Machine embodies the mechanism most computing devices use to work. Mobile phones, computers, washing machines, and many other machines implement a kind of Turing Machine that manipulates symbols. Symbols are important here in the sense that (1) their meaning comes from negotiations and conventions made between the users of those symbols, (2) there are ways to combine and recombine symbols taken to be things we know, to obtain other symbols that correspond to things we do not know yet. (3) Symbol manipulation can also be used to control dynamic processes, such as a washing machine cycle, or the alarm clock in your phone.

The Turing Machine has a *hardware* component defined first by a *tape*, which is divided in individual 'cells' or spaces. Each of these cells or spaces can be either blank or contain a single symbol. The other physical component of a Turing machine is the head, which is always in one cell of the tape (in a given position). The head can read the symbol on the tape, it can write a symbol (effectively replacing a previous symbol), it can move one space to the left or one to the right in a given single machine instruction. The head also stores what we know as the *internal state* of the machine, which is very important for any kind of computation that does any real information processing. See Figure 1A.

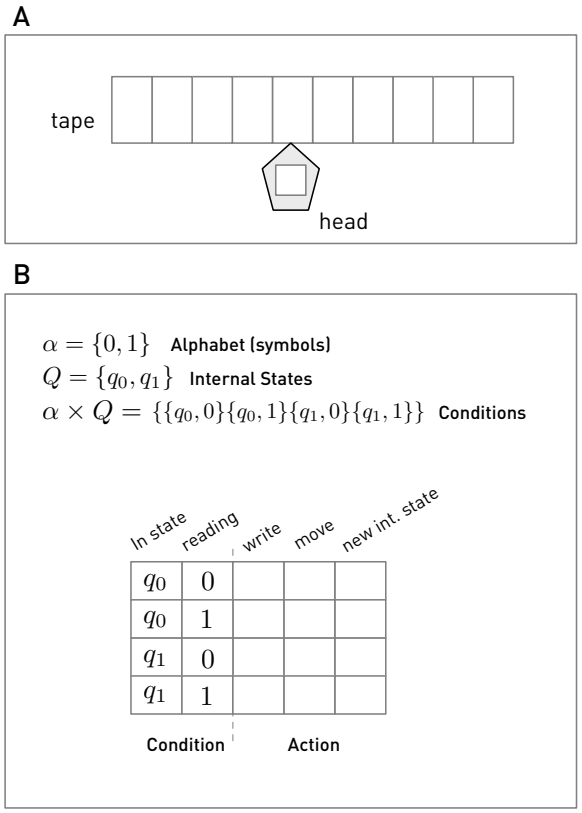

Figure 1: The elements of a Turing Machine.

With its hardware implementation, we can implement the *software* component as follows (see Figure 1B): First we need to define an alphabet α. That is nothing more than a collection of symbols we need to represent the inputs and outputs of the machine we want to implement. It is us (the programmer) who define the alphabet depending on what we want the machine to do. Then, taking into account that we need to know very well the procedure we are programming into the machine, we need to determine the *internal states* that the procedure requires. We will call our set of internal states Q (some books call this S). Recall that internal states are pieces of information your procedure needs to work but that are neither input nor output (input/output symbols are already in the alphabet α). And example of internal state is the carrying one when we do binary sum. See Figure 2.

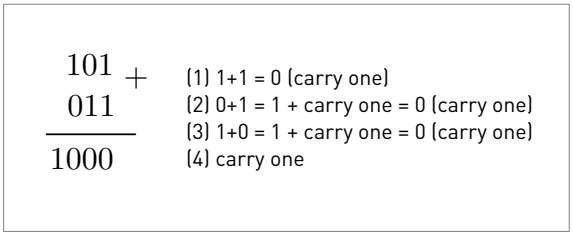

Figure 2: Example of Internal State. Notice that the "carry one" is an essential part of a procedure as simple as the sum of two binaries. Notice also that this carried one is not part of the input or the output we see on the tape, it is a property of the procedure itself.

The next step is to do what is known as the cross product $\alpha \times Q$. This is simply all the possible pairwise combinations of the elements of α and the elements of Q (see Figure 1B). Each of this combination pairs is what the Turing machine can find in a given moment t (a symbol on the tape where the machine is, and the machine's internal state). We refer to this as the *condition*. For each such condition we need to tell the machine what to do. We refer to each of these instructions as an *action*. In any Turing machine the action is given in three parts: (1) what the machine needs to write on the tape; (2) where the machine needs to move – one space to the left or one space to the right; and (3) what will its new internal state be. Every action is always executed in this order. Also, at least one of the instructions (for one combination pair) has to specify an internal state update to something we call *halt*. This tells the machine when to stop. Programming all the actions for the condition pairs is the hard part of implementing a Turing Machine, particularly when we want to implement complex procedures that have many symbols and many internal states. Sometimes it happens that for the finished program some condition pairs are never visited by the machine. This means that there is no sequence of instructions that can drive the machine to those 'unreachable' conditions. When this happens, the program includes only the conditions and actions for the valid situations, not including the unreachable conditions. The example you analysed in the first tutorial is an example of this. In other words, we often do not need to specify actions for all the conditions in $\alpha \times Q$ because a subset of these with the appropriate actions ensures an effective implementation of the procedure we want to program a machine to do.

McCulloch and Pitts Neural-Network Machines

Warren McCulloch and Walter Pitts thought about 'artificial intelligence' in a way that is very different from the symbol-manipulating approach of Alan Turing. Indeed McCulloch and Pitts come from the *thinking humanly* approach to AI. They were interested in understanding and creating machines that function in ways that are similar to real brains. For a little background, see Figure 3. Here the main parts of a real neurone are depicted. In a very simplified way, we can say that neurones have input sensors called dendrites, through which they can receive bio-electrical impulses. Neurones have a single output channel called *axon* that can emit a bio-electric signal. Axons can branch out into many axon terminals. When the axon terminal of a neurone meets a dendrite and the axon terminal is sending a pulse, we have what is known as a *synapse.* This process passes bio-electric pulses to the cell nucleus of the receiving neurone. The cell nucleus works like a capacitor with a saturation limit. When a neurone has incoming activity that is greater than its saturation, it 'burts' emiting a pulse that travels through the axon reaching all of its terminals. Some axon terminals can interact with the main axon output of a neurone causing an inhibition of outgoing pulses. This is essentially the basic machinery used by real brains. As you can see, there are no symbols here.

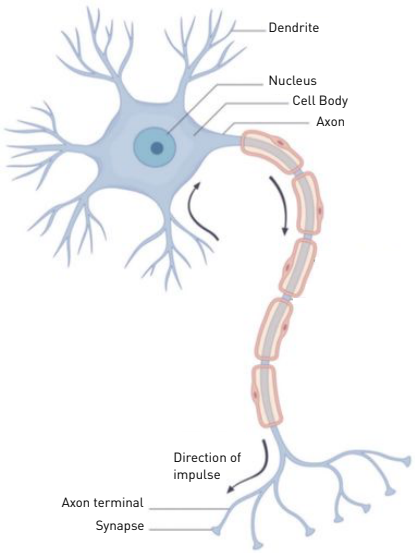

Figure 3: A simplified diagram depicting the main components of a real neurone.

McCulloch and Pitts were inspired by real neurones to build a machine idealisation that captures their essence and basic mechanisms, albeit in a very simplified way. The machines of McCulloch and Pitts are made of very simple artificial neurones. Each of these has a *threshold* denoted by l which is an integer number used to represent the saturation level of that neurone. The machine allows for incoming *fibres* that can be *excitatory* (terminations represented with a filled black circle) or *inhibitory* (terminations represented with a open white circle). Fibres always connect one neurone to another neurone or from one neurone to itself. Fibres can branch out, or join into single fibres. Excitatory fibres contribute to reaching the saturation threshold

of a neurone. However, a single active inhibitory connection will block the firing of a neurone. These details about the dynamics of these neural networks are explained next. See Figure 4 for a depiction of the basic neurone of McCulloch and Pitts, and Figure 5 for examples of correct and incorrect ways of wiring them into networks.

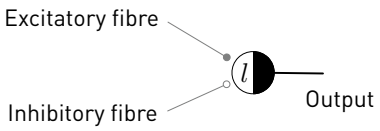

Figure 4: The basic neurone of McCulloch and Pitts.

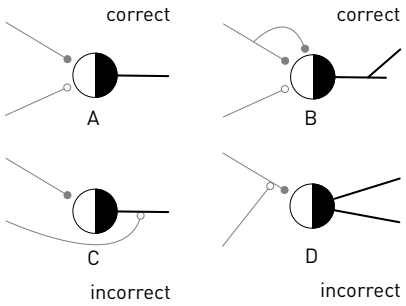

Figure 5: Correct and incorrect ways of wiring a McCulloch and Pitts neural network.

Once the network architecture is defined, the dynamics takes place in *discrete* time steps. This means that the network is not changing continuously. Instead, the way it works is as follows: at a given time t the network is frozen, and every neurone checks how many incoming excitatory fibres are firing (sending a pulse) simultaneously. If the sum of the firing fibres is equal or greater than the neurone's saturation threshold l, then that neurone will be firing (sending a pulse) in the next time step $t + 1$. However, and this is important, any neurone that has at least one inhibitory fibre firing at time t *will not fire* at $t + 1$ no matter how many excitatory incoming fibres were firing at time t. Inhibitory fibres that are firing into some neurone at time t will always block the output at $t + 1$. This extremely simple architecture allows the implementation of, for example, logical gates and delays as shown in Figure 6. This created a lot of excitement that gave rise to a scientific research field called *Cybernetics*. The evolution of this field has branched into what is known today as *Deep Learning*.

During class we discussed implementations of basic neural networks that implement two important mechanisms for information processing: control and memory. For control, have a look at Figure 7 and explain in the box how control is implemented by each of the examples (A, B and D). Concerning the main mechanism for implementing memory, we spoke about *feedback loops* (Figure 7C)in the practical session. Make sure you fully understand feedback loops, ask questions and add your notes here.

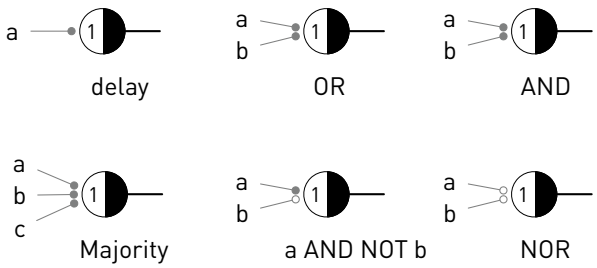

Figure 6: Examples of elementary machines that can be implemented with McCulloch and Pitts neural nets.

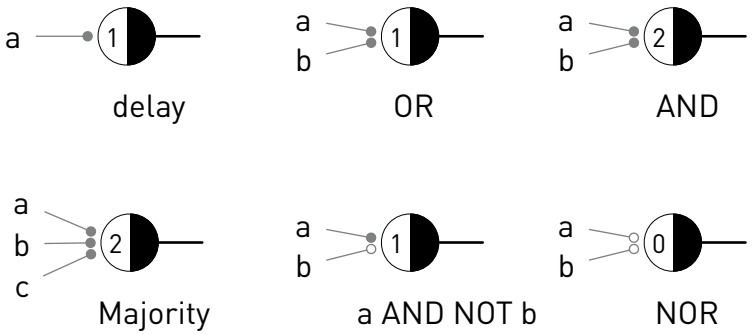

Figure 7: Control gates using neural networks.

State Transition Diagrams

Turing, McCulloch and Pitts defined very different machines. However, they are still machines and therefore they should have something in common. Indeed they do. While they are running, both have internal states. For the Turing machine this is represented by what the head stores in its memory after each instruction of the program (which is encoded by the programmer). For the neural networks of McCulloch and Pitts the internal state of the machine is the specific pattern of firing and non firing neurones at a given time that determine in what state it is. For example, if a neural network has three neurones, and at some time t they are all not firing (quiet) that will correspond to the internal state. This would be different to another internal state where the same neural network has neurones one and two firing, but neurone number three not firing. Turing machines have inputs: the symbols that are read on the tape at the start and during each time step of its execution. Neural networks also have inputs: the patterns of pulses that are going into the neurones of the network during each time t the machine is executed. Finally, Turing machines have outputs: the symbols that are progressively written on the tape until the final output, while neural networks have the patterns of pulses in the outgoing fibres of its neurones.

We can analyse the behaviour of any machine by being aware that they all have inputs, outputs and internal states that are updated during the machine's execution. Consider for example the simplest neural network possible in Figure 8A. If you try to simulate its execution, you will quickly realise that this machines outputs at $t+1$ whatever was the input one time step before at time t. This is called a single delay memory. Analyse the diagram presented in Figure 8B, and make sure in class that you fully understood it. Add your own notes about it here.

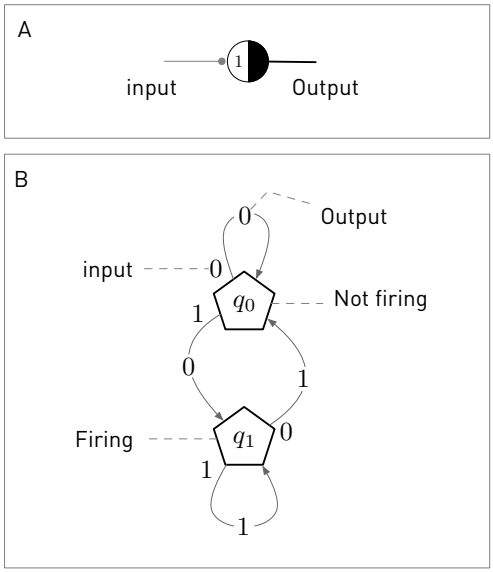

Figure 8: The simplest neural network, and its state transition diagram. To design this diagram one must identify the possible internal states of the machine, and represent what inputs can be seen from those states, represent the output, and with an arrow represent state transitions.