Neural Networks for physics

Simone Rodini

April 9, 2024

Contents

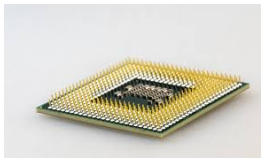
Outline

How does a computer think?

1. How does a digital computer actually think? A brief window on machine instructions, assembly and programming languages.
2. How can we make a higher-level model of intelligence (or logical reasoning) on a machine? Neural Networks demystified.
3. A specific application: how to use a NN to model an unknown physics phenomena.

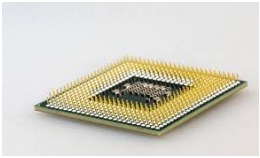
Computer basics

A general idea

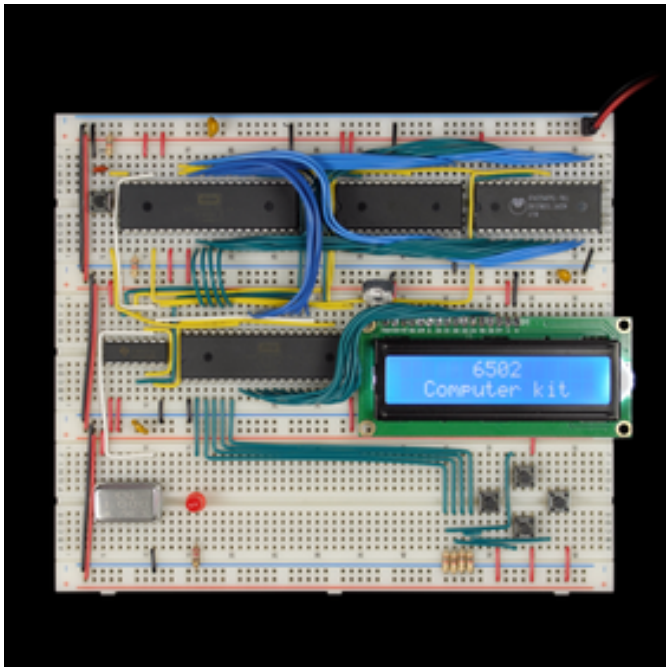


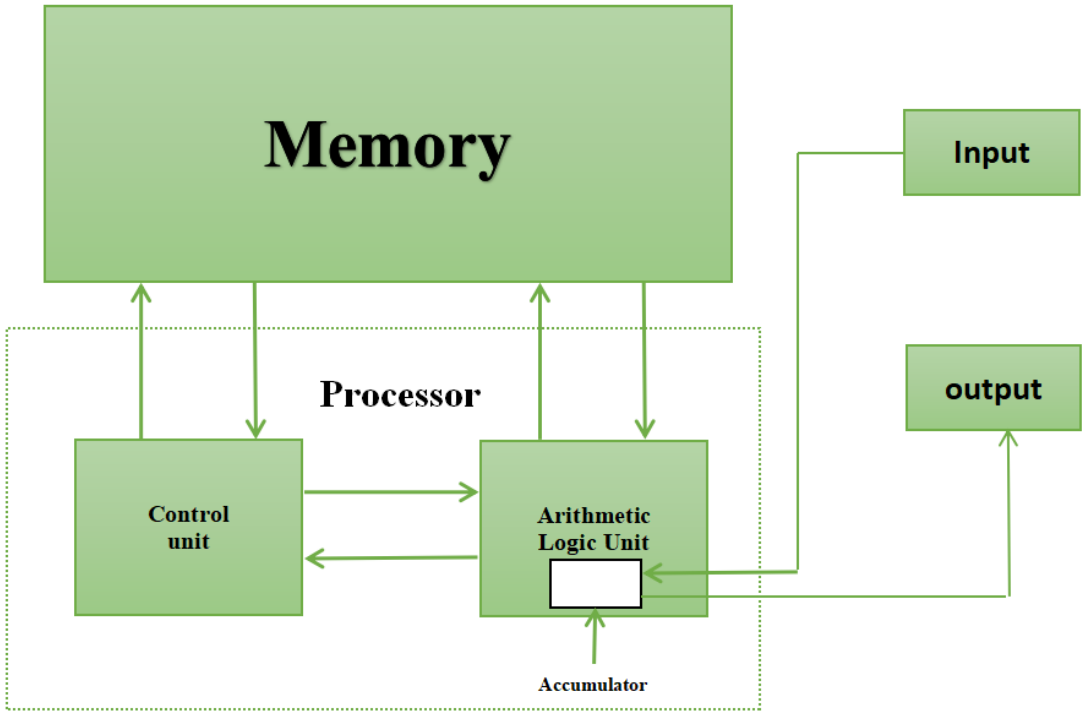
Transistors: the foundation

A general idea



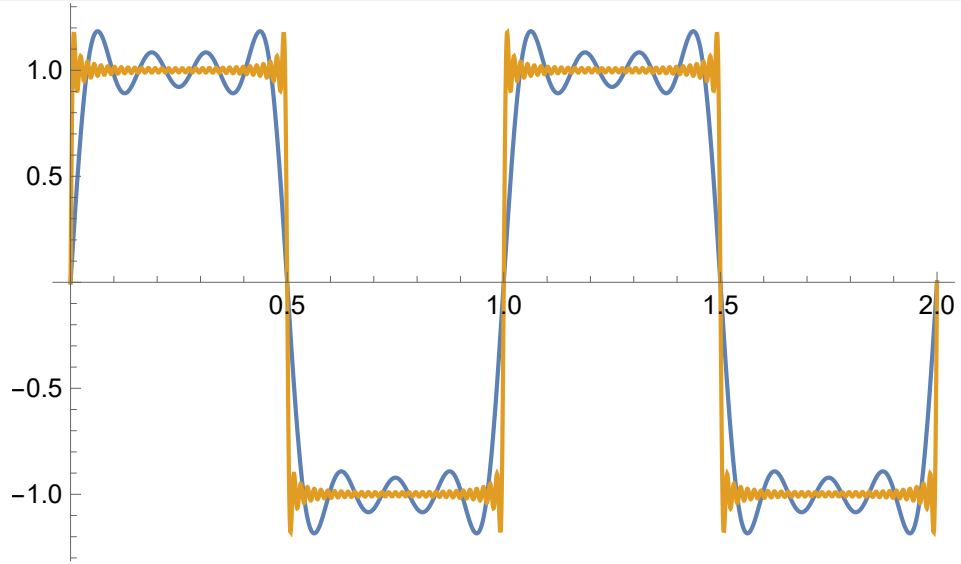
Transistors: the foundation





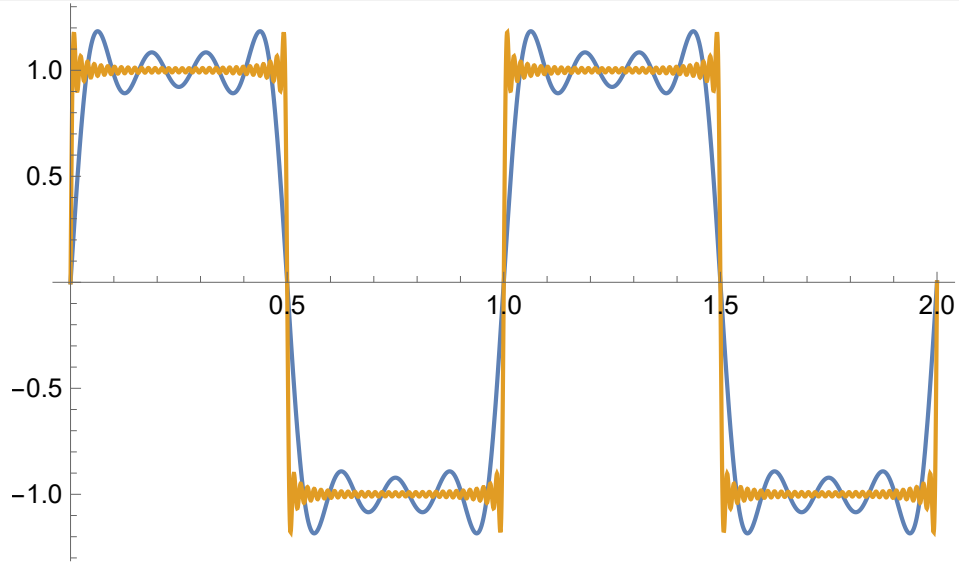
Number representation

All digital computer works storing information as voltage level.
We can store 2 values, conventionally referred to as 0, 1.



Number representation

All digital computer works storing information as voltage level.
We can store 2 values, conventionally referred to as 0, 1.



Why not more? Possible, but it has costs: energy, reliability, scalability etc...
How can we represent numbers using only $0, 1$? Binary!
How to read a base-10 number?

$$\underset{\hookleftarrow}{6710597111}_{10} \equiv 1 \cdot 10^0 + 1 \cdot 10^1 + 1 \cdot 10^2 + 7 \cdot 10^3 + ...$$

So, binary = base-2:

$$11011011_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + ...$$

Number representation

Nowadays hardware can manage 64 bit numbers: up to $2^{63} \lesssim 10^{19}$.
Inconvenient to think in binary $\Rightarrow$ hexadecimal! Use 0 and other 15 symbols to represent numbers: $0, 1, ..., 9$,A,B,C,D,E,F: $\underbrace{1101}_{\text{D}}\underbrace{1011}_{\text{B}} = \text{DB}_{16}$

1 bit: one 1 or 0, 1 byte: eight 1 and 0 like $01000010 \equiv$ two hex number 42.
64 bits $\equiv 8$ bytes.

Addition table:

	0	1
0	0	1
1	1	0*

and multiplication table:

	0	1
0	0	0
1	0	1

*: carrying the 1, like in $9 + 1 = 0$ plus the carried 1.

Beyond positive integers

From any base n to decimal is simple. From base-10 to any base: quotient-remainder algorithm: a tiny bit more annoying to carry out.
Positive integers ✓
Negative integers? Use the most significant bit (the leftmost one) to encode the sign: $0 \Rightarrow$ positive. In formulas

$$b_{N-1}b_{N-2}...b_0 = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i$$

Beyond positive integers

From any base n to decimal is simple. From base-10 to any base: quotient-remainder algorithm: a tiny bit more annoying to carry out.
Positive integers ✓
Negative integers? Use the most significant bit (the leftmost one) to encode the sign: $0 \Rightarrow$ positive. In formulas

$$b_{N-1}b_{N-2}...b_0 = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i$$

For example:

$$1001_2 = -1 \cdot 2^3 + 1 \cdot 2^0 = -8 + 1 = -7$$

Check: $7_{10} = 0111_2$ and

$$\begin{array}{r} 0^1\,1^1\,1^1\,1+ \\ 1\;\;0\;0\;\;1= \\ \hline 10\;\;0\;0\;\;0 \end{array}$$

Floating Point

How to represent numbers that are not integers?
In binary we can use inverse powers of 2, for instance $0.1_2 \equiv 1 \cdot 2^{-1}$.
A curiosity: whether a number as a periodical form depends on the base! For instance $1/5 = 0.2_{10}$ in binary is $0.001\ 1001\ 1001\ \ldots_2$.

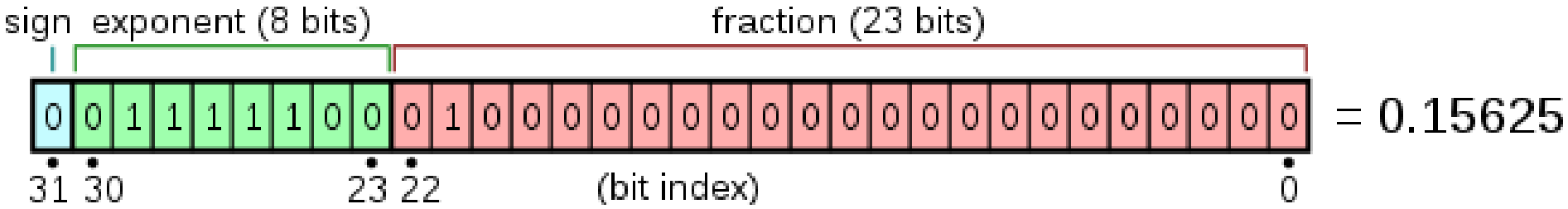
On a computer only so many bits: no real numbers!
Could we reserve some bits for the part after the period? Like

$$\underbrace{101\ldots1}_{\text{32-bit}} \cdot \underbrace{110\ldots1}_{\text{32-bit}}$$

Yes, but impractical: sometimes we want very large numbers, sometimes very small, this has no adaptation capability. This is called fixed-point representation. This representation is very fast, and in older machines that did not have any hardware for floating point it was extensively used (and if you dig in the gcc compiler, you will still find it).

Floating Point

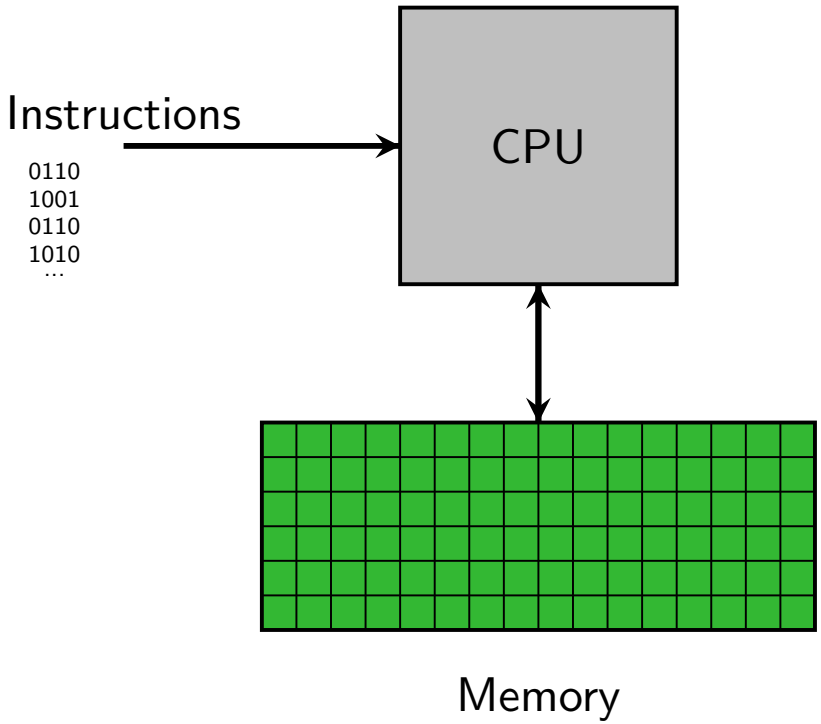
Modern solution: scientific notation! For a 32-bit number:



This is the IEEE 754 standard: everybody must agree how to parse the bits, otherwise it is a mess!

Exercise: you have a simple program that given a floating point input gives you the bit representation as in the figure above, plus the decomposition in powers of 2. What are your observations? How is the number formatted?

How does we instruct the computer



How does it work (in a very approximate way)?

Level 0: BIOS/UEFI = basic software built-in in the motherboard, manages the very low level hardware

Level 1: OS = manages all the software, it is loaded by the BIOS/UEFI by reading instructions off a particular region of the memory

Level 2: programs = the OS manages your programs, the memory for each of them, where they store information etc.

How does we instruct the computer

On a 64 bit machine each instruction can be up to 15 bytes long(!). The 64 bit refers to the maximum size of each register in the cpu.
Each instruction is literally a sequence of $0, 1$ that triggers some electronics to do 'stuff', like adding two numbers, moving memory, jump to different addresses etc.
It is not manageable to write instructions in binary (or hex) format:

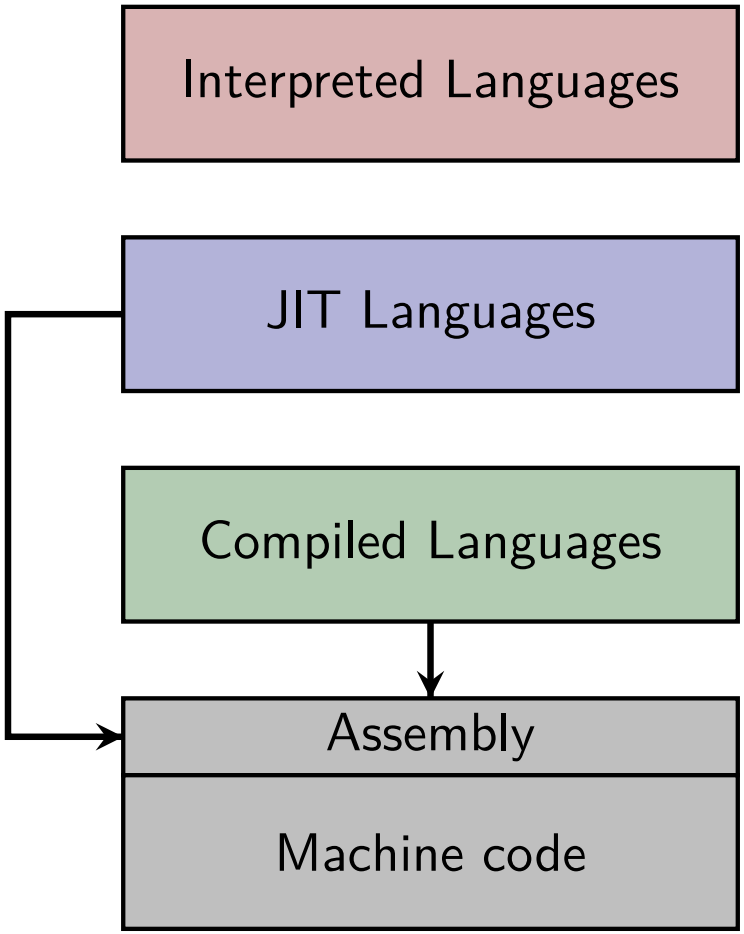
human-readable machine code $\sim$ assembly 


```
push rbp
mov  rbp, rsp
mov  DWORD PTR [rbp-4], 1
mov  DWORD PTR [rbp-8], 2
mov  edx, DWORD PTR [rbp-4]
mov  eax, DWORD PTR [rbp-8]
add  eax, edx
mov  DWORD PTR [rbp-12], eax
mov  eax, 0
pop  rbp
ret
```

More readable than a bunch of 1 and 0, but still a long way from its C counterpart:

```
int a = 1;
int b = 2;
int c = a + b;
```

Programming languages



Examples

What does it mean to 'compile' a language?

$$\text{Human-readable code} \Longrightarrow \text{Machine code}$$

It is a translation!
Python: interpreted language. What does it mean? It reads the commands, parse them, construct on the fly the logical tree and execute them via previously compiled functions.
Lua: JIT (just-in-time compiled) language. It reads pieces of code, compile them on-the-fly and execute the resulting binary.
C/C++: compiled languages. There is a dedicated piece of software (the compiler) that translates the code to machine instructions. This is done separately from the running of the program itself.

Programming languages (or syntaxes) come in all shape and form!
Not all of them are friendly: in the beginning select the language that feels most natural to you, then you should keep in mind to use the right tool for the job.
For instance, did you know that you can programm a roller coaster in Excel? Link
Or that it does exist a (Turing complete!) programming language with just 8 symbols? It is called Brainfuck (guess why...) and to print the string 'Hello World!' the code is

```
++++++++++[>++++[>++>+++>+++>+
>+<<<<-]>+>+>->>+[<]<-]
>>.>---.+++++++..+++.>>.<-.
<.+++.------.--------.>>+.>++.
```

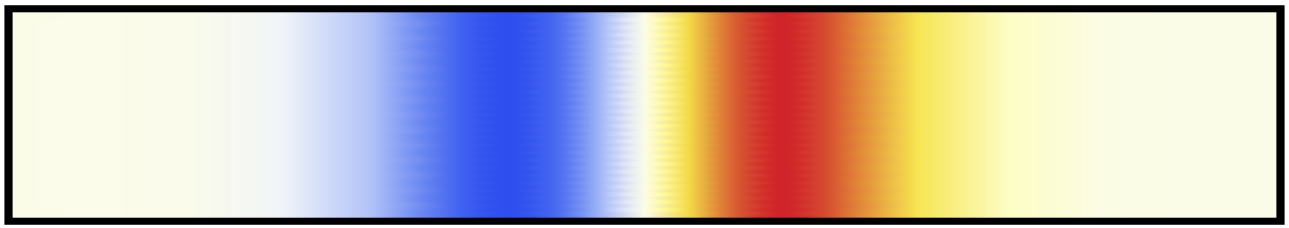
Crazy, ain't it?

Your turn now, let us play a bit!

You have two codes:
1) 'lecture_1/HeatEquation.ipynb', which is a Jupyter notebook (runs on python)
2) an executable 'Code/lecture_1/a.exe', which is generated from the 'Code/lecture_1/play_w_bits.c'

Heat Equation

It describes how the heat defuse along a (mono-dimensional) bar as time goes on. The surrounding environment is assumed at constant temperature and with infinite heat capacity (to start with).



Besides how it is defined, you can play around with the initial condition (i.e. how the heat is distributed at the initial time) and how the surrounding temperature changes

What you can do

1) Play around with the bit executable, try to feed different numbers. What are your observations? Do you understand the procedure? Check some of the powers of 2 decompositions to convince yourself that the sums that are given actually produces back your original number.
2) In the Heat equation notebook you have two lines enclosed in the comment:

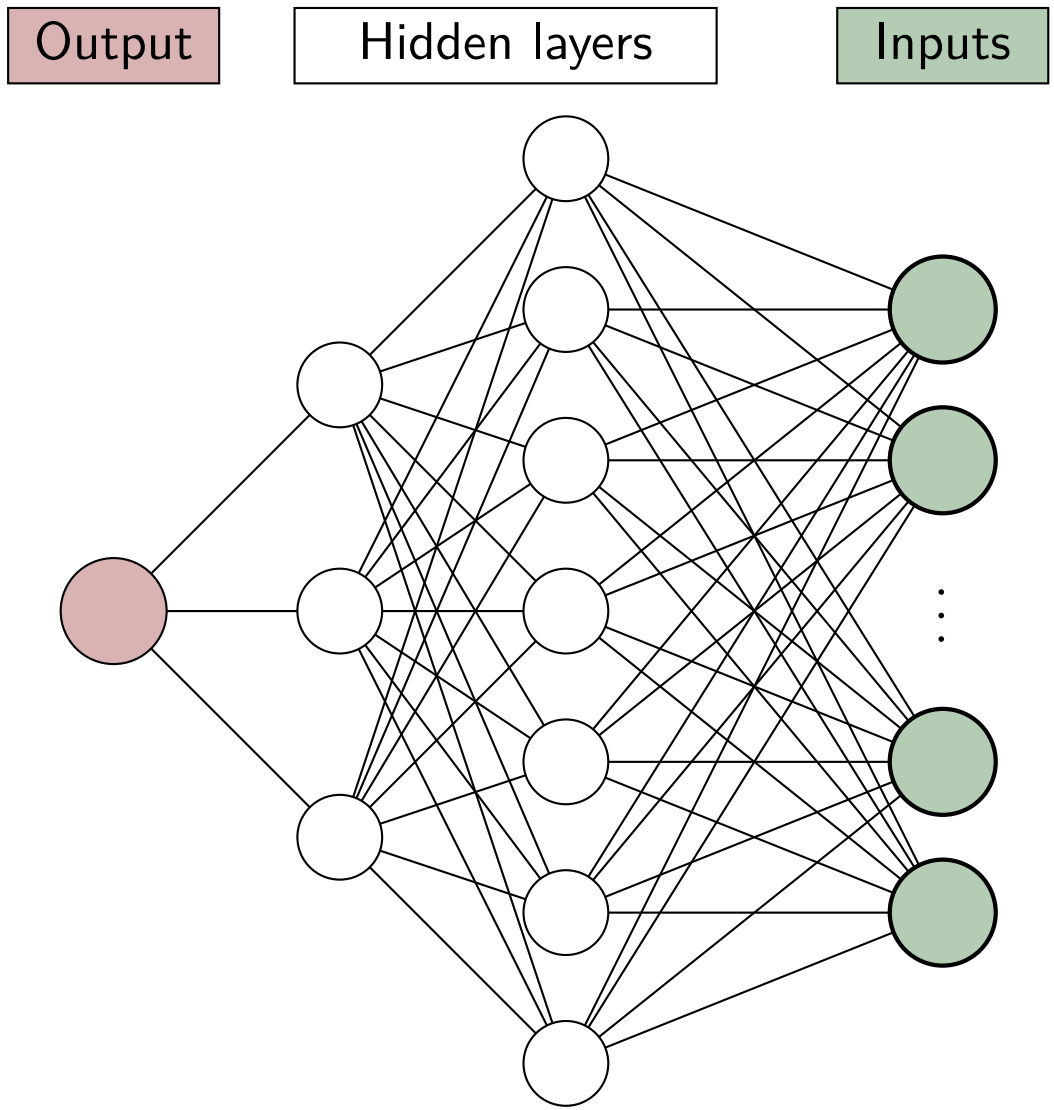
```
# ----- Change me -------:
u[i] = np.exp(-x[i]**2) * np.sin( np.pi * x[i] / 3)
v[i] = 0
```

The first set the initial condition to be $\begin{cases} e^{-x^2}\sin\left(\frac{\pi}{3}x\right) & \text{if } -3 \leq x \leq 3 \\ 0 & \text{otherwise} \end{cases}$

The second set the environment at constant zero temperature. Try to change these functions following the directions that we give you!

Neural Networks, the basics

Schematic



Ingredients

Each line is a 'weight' $w_{ij}^{(l)}$, where j index the node on the right and i index the node on the left and l gives the layer, numbered from right to left. To each node give a 'bias' b_i^l.
To compute the first node of the first hidden layer do:
1) for each input x_1, ... x_4 compute $t_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + w_{14}x_4$.
2) Add the bias: $t'_1 = t_1 + b_1^{(1)}$.
3) Apply some non-linear function* $y_1^1 = \sigma(t'_1)$.
Then repeat for all other nodes. For the second hidden layer use as input the $y^{(}1)_i$ etc.
* non-linear $\sim f(ax + by) \neq af(x) + bf(y)$.

Few more details

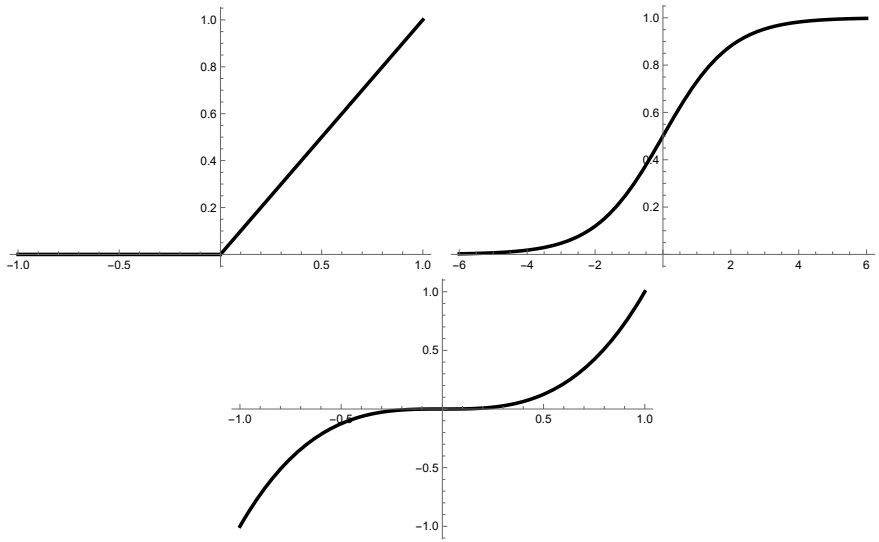
The choice of the function is important. In general different functions for each layer.

Classical example:

1) ReLU: $\begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

2) Sigmoid: $1/\left(1+e^{-x}\right)$

3) Cubic x^3



But how to get results?

If weights and biases are given at random the results is garbage!
We need a procedure to find a set of weights and biases such that the output is in line with what we want.