Machine Learning

IMPORTANT

> Many of these concepts are explained in the best way possible using visual examples in this website https://mlu-explain.github.io/. Go here please.

Explicit instructions are at the core of nearly every algorithm that we use, if we want a loop that generates the Fibonacci numbers, we know exactly how to do it. We give the computer the instructions in the form of python code and it gives out exactly what we asked.

This is not the case in Machine Learning, the instructions that these algorithms use are not explicit, the algorithm themselves learn them.

These algorithms improve their performance after observations about the world, that is, they improve by using data. For example, they can evaluate their performance at a given time and change their internal parameters accordingly.

There are several reasons for using Machine Learning (ML from now on):

- Unexpected future situations; capacity of generalization
- Manual knowledge coding: hard and time consuming Task
- Adaptation
- When there is no solving method

There are three main types of learning, they are usually called paradigms:

- Supervised

  > with a teacher (learning from pairs of input-output, the desired outcomes)

- Unsupervised

  > without a teacher (only learning from inputs)

- Reinforcement

  > agent acting, there are reward/penalties for actions executed

All of these will be explained. We are also going to look at Neural Learning, which it is not a learning paradigm. It is a type of model architecture, it indicated how an AI model is build, not how it learns. This architecture can be used in the main tree learning paradigms.

The opposite concept of Neural Learning is symbolic learning, another model "architecture" which we are going to look also (e.g. decision trees).

Explanation-based Learning (EBL)

To describe ML models we also need to take into account the existence of prior knowledge. The main type of training that uses this concept to the maximum is Explanation-based Learning.

These models require a perfectly defined domain, we can take chess as an example, where there are perfectly define rules and game states, there is no ambiguity. We can this a perfect o complete domain, a domain that has all information needed to answer any question regarding the domain.

An EBL system can take this types of domain, and with just a single example of a state with a label (e.g. "forced loss of black queen in two moves"), an deduce all the features or characteristics that that particular state has.

We only need 4 types of information to create an EBL system:

1. the set of all possible conclusions (e.g. all game states)
2. axioms about the domain (e.g. the rules of chess)
3. training examples (e.g. examples of a *jaque mate*)
4. criteria for determining which features in the domain are efficiently recognizable, which features are directly detectable by sensors (e.g. if a white piece can "kill" a black one)

These models are notably used in NLP, to form a grammar tree (i.e. what is the Subject and the Predicate of a sentence). They can perform the syntax parsing of a language.

TLDR:

> With perfect information about a game/domain we can use an EBL system to find a way to deduce the features of each training example. With these features learned the model can deduce the best move.

Supervised Learning

Very often we can rely on datasets that are labeled. Imagine that we want to make a model that recognized images of cats, to build it (i.e. trained) we can use a dataset of cat images. By showing the models many many examples of cat images, it can be able to learn how to recognize them.

The learning is done with a teacher in the sense that the model has guidance, how if it has a correct output. Our cat recognizer, can have a binary output:

- Cat
- No cat

The dataset consists of pair of information, an image and a label for that image. During training an image is given as input to the model, that outputs a label (e.g. `cat`). By giving the model images from the dataset, we can compare the outputs to the label and adjust the parameters of the model accordingly. The specific methods to do this are going to be explained in the neural learning part. From now one keep in mind that there are two main types of model architecture Symbolic and Neural.

The key concept behind these types of models is that it learns from examples from which we have information. During the training the model gets positive and negative examples and learns from them.

In the case of our cat recognizer, we are going to give it a bunch of images of cats, and another bunch of no-cats. Keep in mind that is a very simple model that in practice does not make sense (see this 1min video to know what I mean).

By using this learning paradigm, the model is able to predict outputs of unseen inputs (e.g. recognize a cat from an image that has never seen during training).

To generalize, we can say that we give as input a vector of attributes. It can be text, images... Furthermore there can be two main types of output:

- Discrete (e.g. classification task `no cat`/`cat`), it is a boolean value
- Continuous (e.g. regression task for predicting temperature)

This can be tricky and I do not consider is a good way to explain things because the usual way to output is with probabilities. If with discrete outputs we can only use booleans, 95% of all models are going to be continuous because the outputs are expressed in probabilities (e.g. the model "says" *"I am 83% confident that in this image there is a cat"*).

Regression vs Classification

In a classification task, the goal is to assign inputs to predefined categories or classes, with the output being categorical. Examples include spam detection and image classification. If the output is the form of probabilities, it is a continuous value. For more complex tasks these is the case.

In a regression task, the goal is to predict a numerical value based on input data, with the output being continuous. Examples include predicting house prices and estimating temperature.

TLDR: Classification is discrete and regression is continuous, but there are cases with classification that this is not true.

Types of datasets

In a supervised learning case, a big dataset is divided into 3 smaller ones:

- Training set (to perform training)
- Validation set (to perform validation)

- Test set (to assess model performance)

The largest one of the 3 is the training set, often there is a 80/20 between the training and test ones. The validation one is used for what we call hyperparameter tuning (i.e. changing the architecture of the model)

The main takeaway is the the train and test datasets cannot have duplicates of the data. This is done in order to ensure generalization of the model:
Imagine that the training set consists of an unusual high number of black cats, the model is going to get particularly good at recognizing black cat, but when using one that is orange it may struggle a bit. The test set is used to check that this is not the case. We used to ensure accuracy in the "wild", with never before seen example of the data.

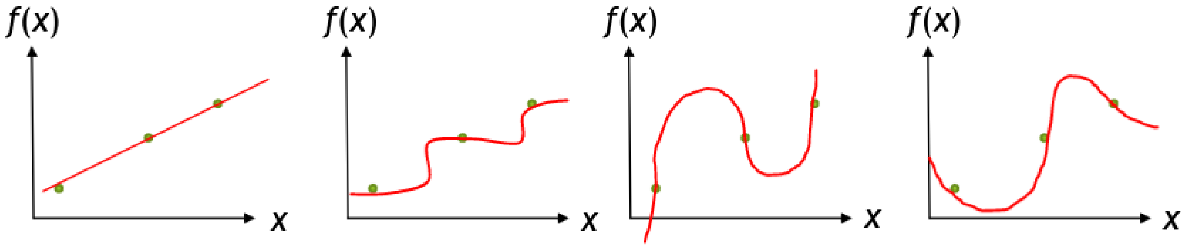
If you want to learn a bit more or there is something that is not quite clear, please check out this website it has wonderful explanation of this 3 types of datasets.

Example: Hypothesis function

- Training set consistent of paint of inputs x_i and outputs y_i called datapoints:

$$(x_{1,}, y_{1,}), (x_2, y_2), \cdots, (x_n, y_n)$$

- Unknown function $f(x) = y$ that describes the datapoints
- We look for a function h called hypothesis, such that h approximates f:



- To validate the h function we use the test set, a sample of datapoints (x_i, y_i) different from the training. With this dataset we assess if h is valid for our application.

Inductive learning

The key idea behind inductive learning is to figure out a general rule from a sample of specific examples.

This is closely related with Rule-based machine learning, where rules presented in the form of `{IF:THEN} expression` are often used. For example: `{IF 'red' AND 'octagon' THEN 'stop-sign'.

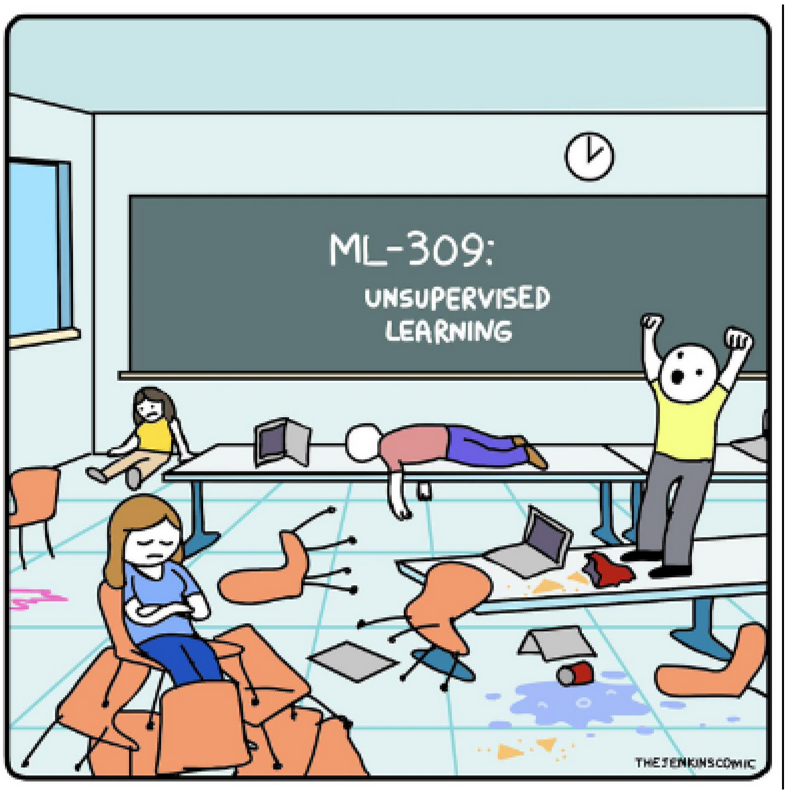
As you can imagine, decision trees play an important part in this types of algorithm because we can work with a clear set of rules where there is no ambiguity.

Do not give much though to inductive learning, it seems a terms more used in cognitive psychology than in ML. Just relate the term to decision trees.

Unsupervised Learning

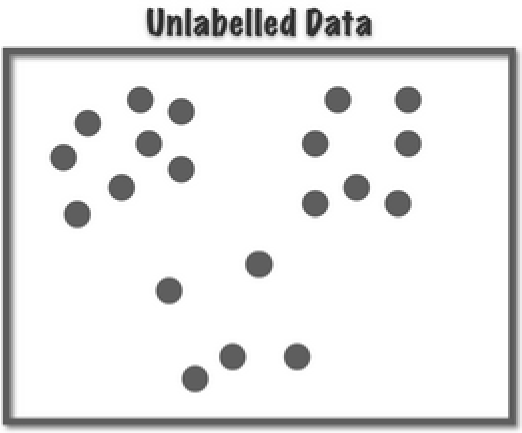
> *But what if we do not have labels in our data???* - You, probably right now

Then, we just use unsupervised learning:



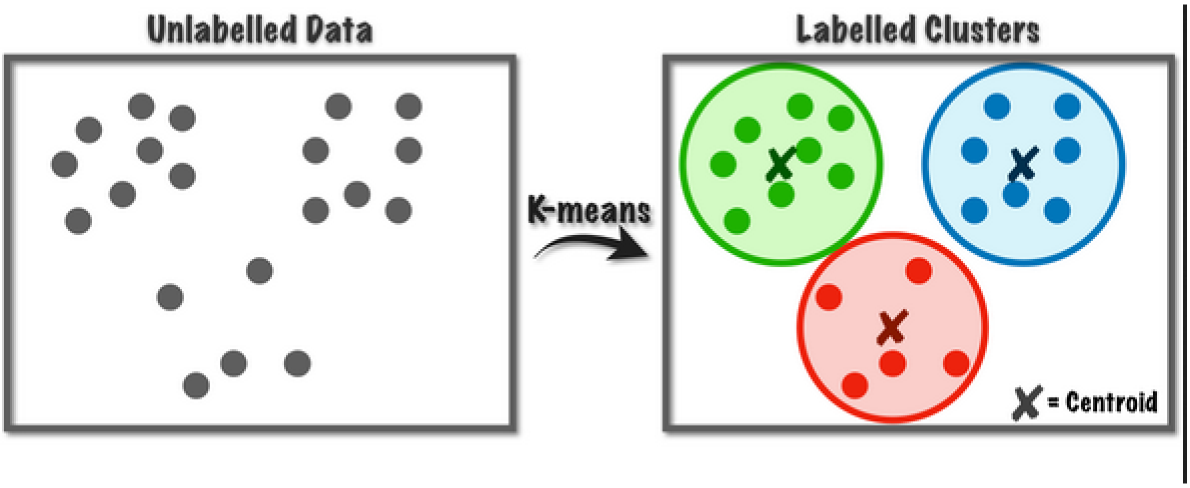
The training is performed without a teacher, that sees, the model does not evaluate their performance from labels in the training or test datasets. The models learn patterns exclusively from unlabeled data.

For example, I ask you to cluster (make groups) out of this data points:



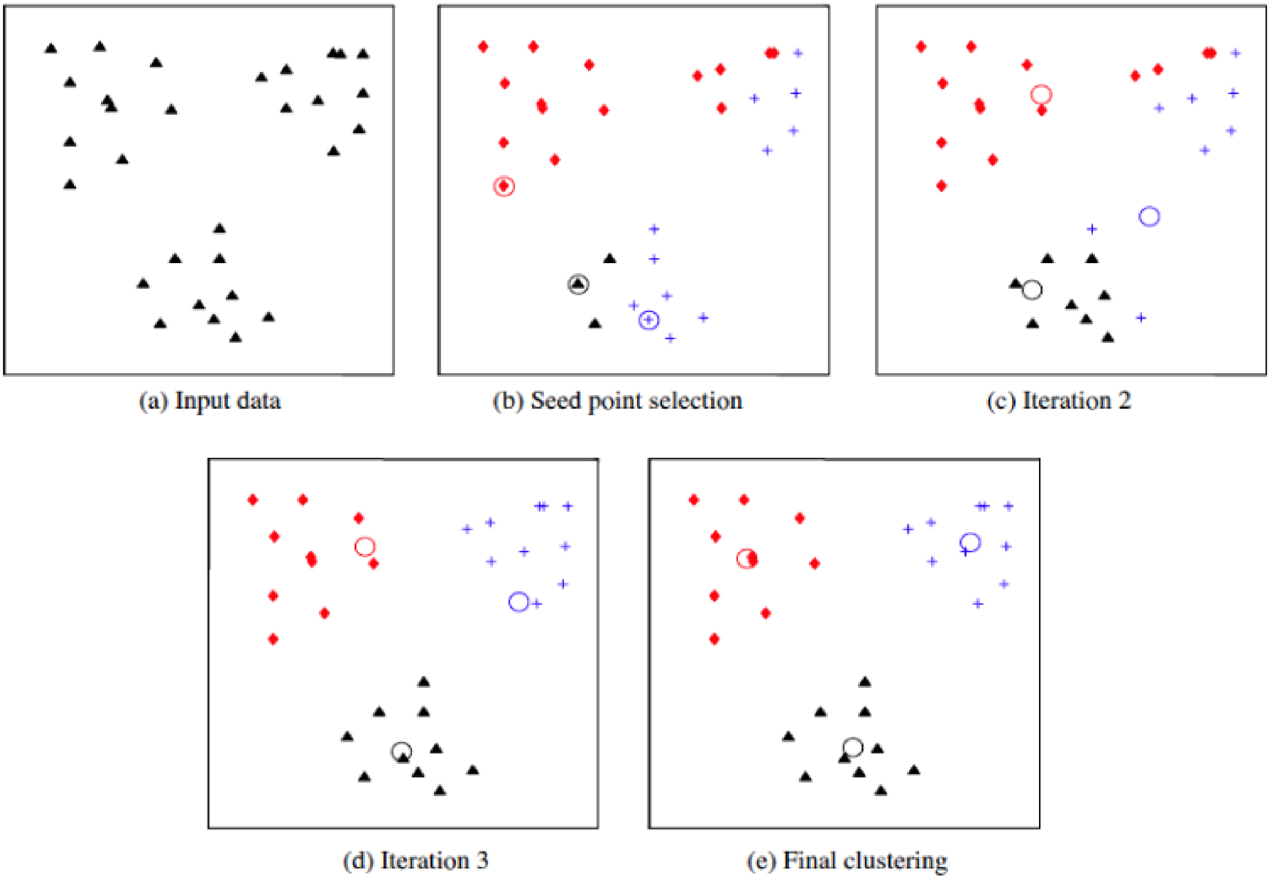
You are probably able to form the clusters in your head, without any additional knowledge of the data. It turns out that computers with the help of unsupervised learning are able to do so!

The main algorithm for these types of tasks (clustering) is k-means, which is able to cluster data points based on the clusters formed from centroids:



The centroid are just the centers of each cluster, from them we can define the cluster using a radius.

The parameters of the model (position of the centroid and their radius) are learned by iteration:


(a) Input data

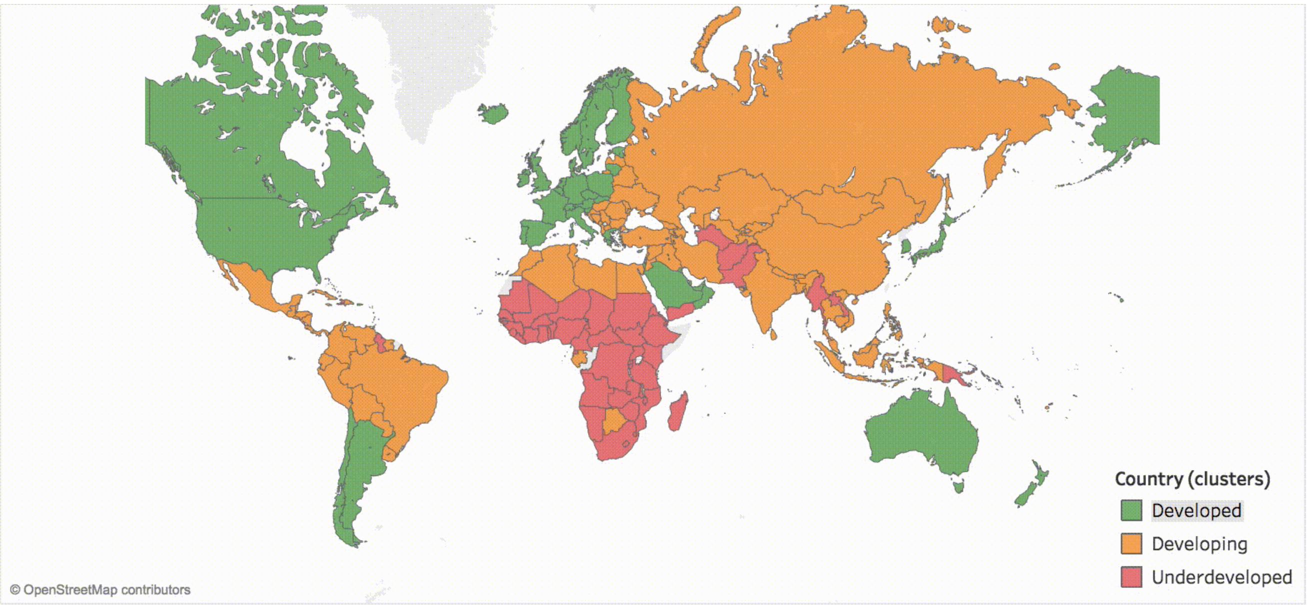
(b) Seed point selection

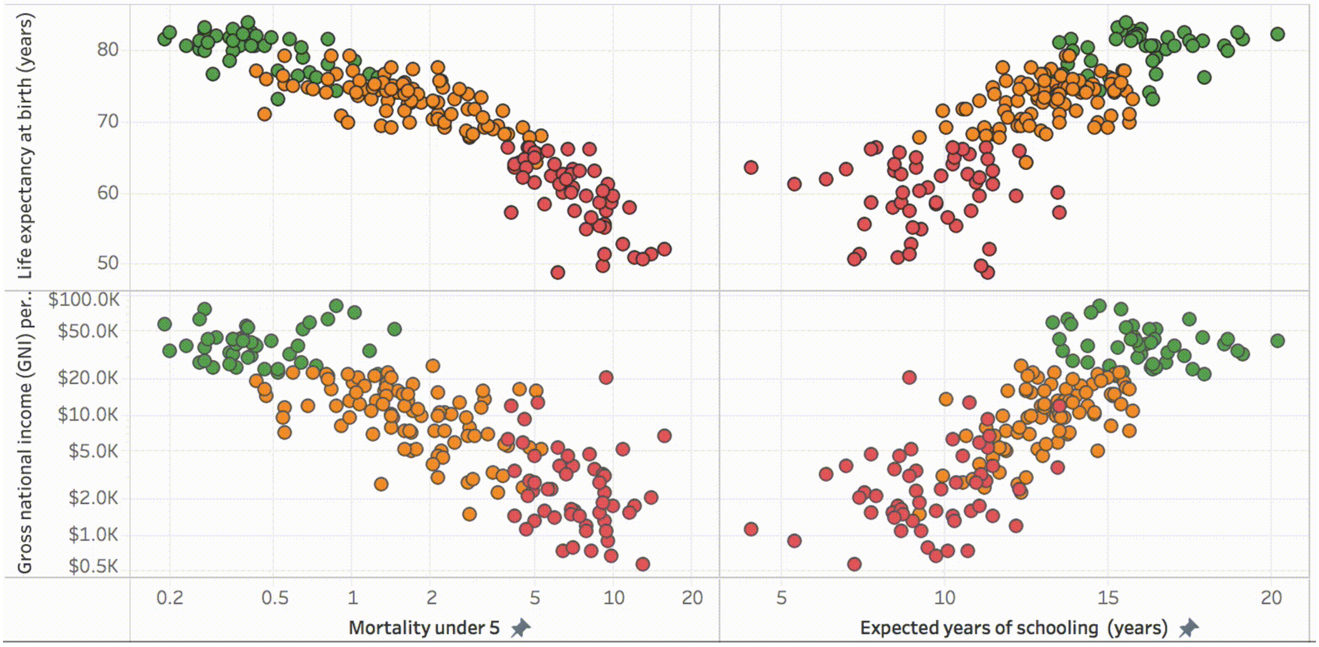
(c) Iteration 2

(d) Iteration 3

(e) Final clustering

Here there is a practical quick example of clustering done to classify countries based on their "development" status. Note how different charts (therefore parameters) play a role in the clustering.





Autoencoders

> Not important for the exam. This is extra info. But it also on the Neural Learning part of the lecture slides.

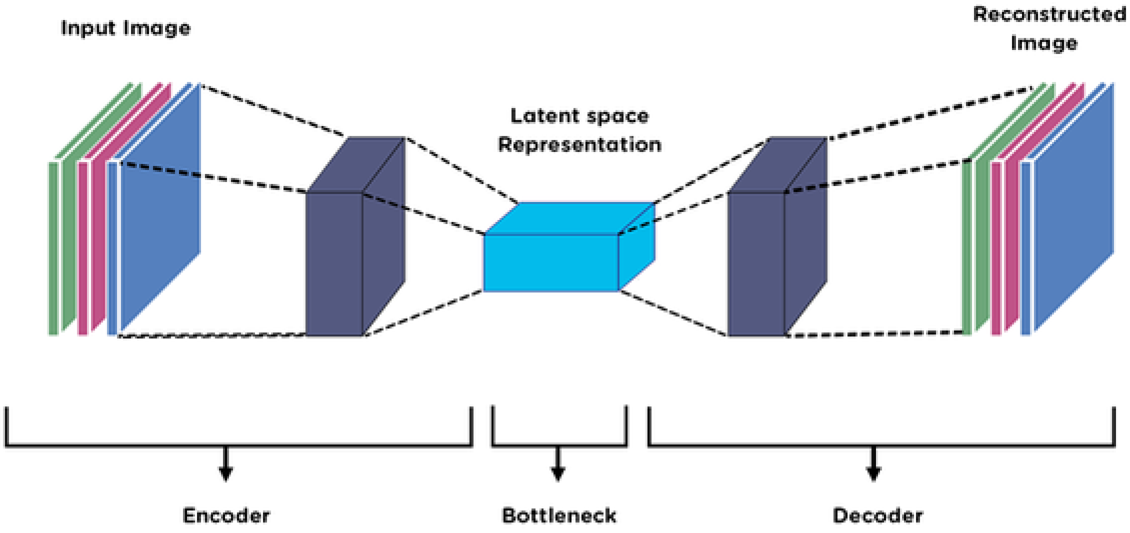
Maybe the most relevant use of autoencoders, a type of neural networks that are able to learn efficient encoding of unlabeled data.

The model architecture consists of an encoder and decoder:

- Encoder: Takes input data (e.g. song, image...) and compressed it into a set of different features. They are abstract and learned by the model itself usually.
- Decoder: Takes the compressed data and tries to turn it into its original form (i.e. the input given).
  You can see how the model is trained without the labels, the relevant work done here is

make as similar as possible the input data and the outputs. Reconstruct the compressed data into its original form.

The model architecture looks like funnel which gets narrower and then wider. The size of the input and output data match, of course, as we want them to be exactly the same:



The middle part, the compressed data, is called the latent space or the latent space representation of the data.

Reinforcement Learning

The key idea behind RL is that an AI can be an agent, in other words, that is a "player" in a environment that can makes actions depending on the current state of the environment. These are algorithms based on trial and error that learn to make decisions based on their environment.

To teach the agent what are the right actions, we use a reward function. This is the goal of the agent, maximize the function. We can also add penalties.

Imagine that we want an AI that plays Mario Bros, the reward is a functions that indicates the agent to go to the right of the screen (i.e. advance in the game). The additional penalty that we add is that their performance decreases if the agent does not move, thus, the agent will try to complete the game as fast as possible. Reference Video of AI playing Mario Bros

You can see how time plays a crucial role in RL, we do not want our agent to stay still. Moreover, keep in mind that the feedback that we give the agent is delayed, we cannot punish/reward if the consequences of their actions are not finish (e.g. in the middle of a jump).

Also it is easy to see how this are "online algorithms", they affect the environment, their actions determine the subsequent data it receives.

RL is the learning algorithm behind many evolutionary AIs which are mainly used to play videogames due to their fast adaptation. This YT channel is full of examples.

On the slides there is also this maze solver via RL (keep in mind that it is not evolutionary).

Neural Learning

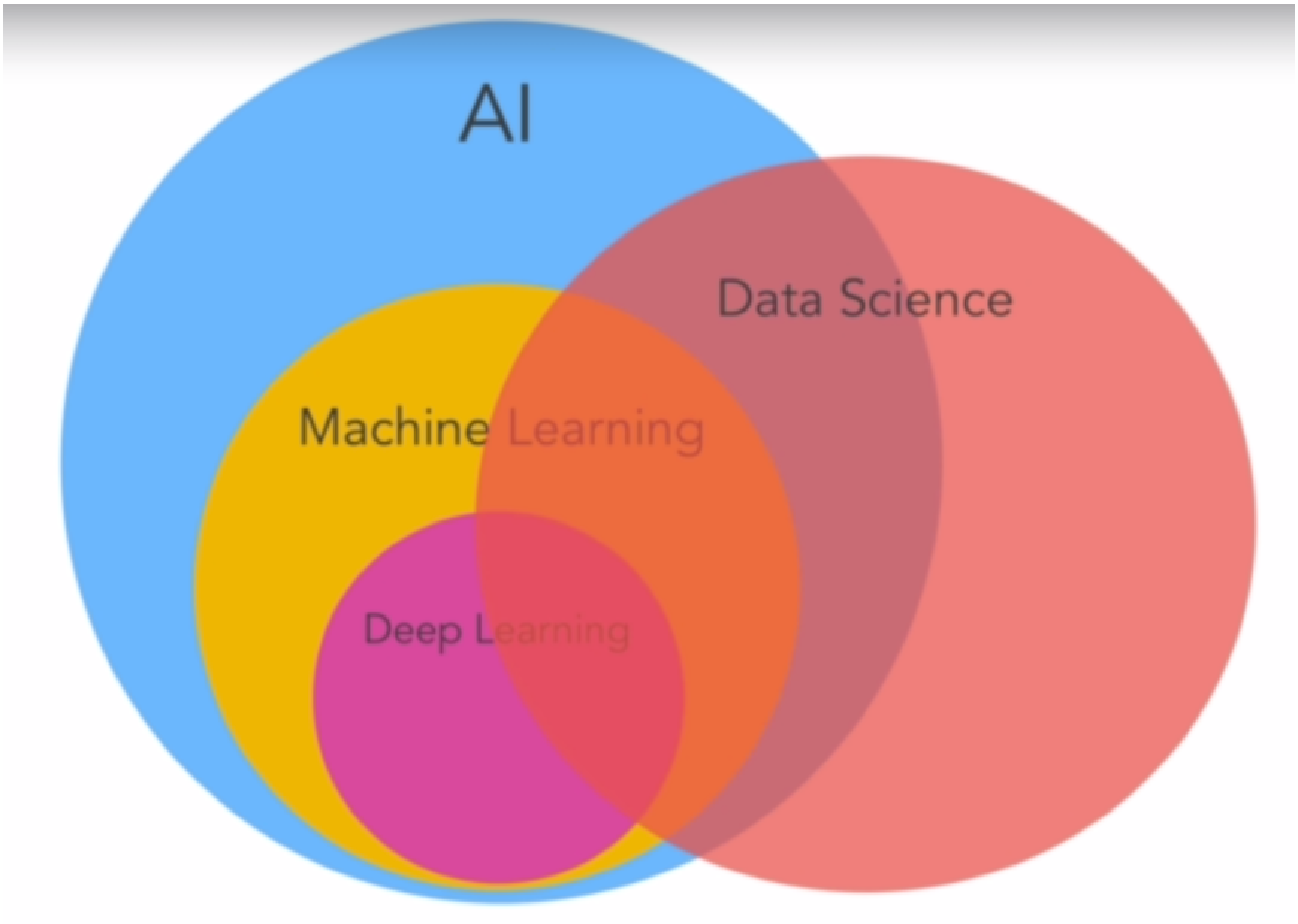
> You can skip this introduction but here I clarify some things about vocabulary and give some extra information about the current state of DL.

How the most important part, we are talking and talking about rewards, functions, actions, images, text, clustering and whatever. But how do this think work really?

The answer as always, is **Linear Algebra and Calculus**. However, we are going to call these things Neural Networks (NNs) and Backpropagation (Backprop) respectively.

When we talk about Neural Networks we are refereeing to the architecture of an AI model. How the data is transformed and treated. This type of architecture is just a bunch of matrices a vector. That is way I say that NNs are Linear Algebra

Keep in mind that NNs are ***just a type*** of AI models. Not all AI models are NNs and not all AI models use Machine Learning:



(NNs = Deep Learning, for the purpose of simplicity, but this is *not* entirely true). Read more here.

When we talk about optimization or machine learning with respect to this architecture we are talking about Backpropagation, the algorithms that makes the NNs learn. This is just a bunch of partial derivatives. That is way I say that Backprop is Calculus.

Deep Learning is just NNs taken to the extreme. All relevant AIs today, the ones that you hear in the news/you use are Deep Learning models. Here there are some examples:

- GPT-4 (OpenAI) `multimodal generation (text and images)`
- Gemeni (Google) `multimodal generation (text and images)`
- ChatGPT/GPT-3 (OpenAI) `text generation`
- **MIXTRAL (Mistral)** **`text generation`**
- Bard/LaMDA (Google) `text generation`
- **Stable Diffusion (Stability AI)** **`image generation`**
- DALL·E 3 (OpenAI) `image generation`
- **LLAMA (Meta/Facebook)** **`text generation`**
- Siri, Google Assistant, Alexa... `virtual assistants`
- Tesla Autonomous Driving Beta (Tesla) `coches que van solos`
- **Whisper (OpenAI)** **`speech to text`**
- GitHub Copilot (Microsoft/OpenAI) `coding companion`
- AlphaCode (Google) `code generation`
- **AlphaFold(Google)** **`protein structure`**
  I just list them so you now, the company that makes the models are in (parentheses). The ones in bold are open source.

You need to keep in mind how a company names their model and their product: ChatGPT **is a product**, the model behind it (the free version) is GPT-3.5.

Software used in Deep Learning:

- **PyTorch (Meta/Facebook)** **`python library for DL`**
- **TensorFlow (Google)** **`python library for DL`**
- **scikit-learn (open source)** **`python library for ML in general`**
- Hugging Face (Hugging Face) `community to share DL models and datasets`

PyTorch is the Python library that we are going to use in the future, scikit-learn is also **very much** important. Look into it to get a peek of what we are going to do in 2nd/3rd year.

sry for not including links, sometimes im lazy

Videos to understand a Neural Network

just watch them and you can skip the next section and just go to the Deep Learning one

Neural Networks

I am going to try to explain a bit more than necessary, hope is not too much.

The neuron or perceptron

What are the two simplest operations that you can do to data? Multiplication and Sum.

These are the two operations that an artificial network performs, combining them into what we call a *weighed sum*. It can be understood as a vector-vector multiplication, a scalar product:

$$\boldsymbol{x} \cdot \boldsymbol{w} = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

Where $\boldsymbol{x}$ and $\boldsymbol{w}$ are two real valued vector with the same size n.

$\boldsymbol{x}$ is a input vector (the data), and $\boldsymbol{w}$ has the weights that multiple the elements of $\boldsymbol{x}$. You can clearly see how each element has its own weights.

We say that the weights are the parameters of the neuron/perceptron. Easy.

Usually the operation $\boldsymbol{x} \cdot \boldsymbol{w}$ is expressed as:

$$\boldsymbol{x} \cdot \boldsymbol{w} = \sum_{i=1}^{n} x_i w_i$$

But this is not all, in a neuron there is another parameter that we call bias, usually represented by b, it just sums the previous operation:

$$b + (\boldsymbol{x} \cdot \boldsymbol{w}) = b + \sum_{i=1}^{n} x_i w_i$$

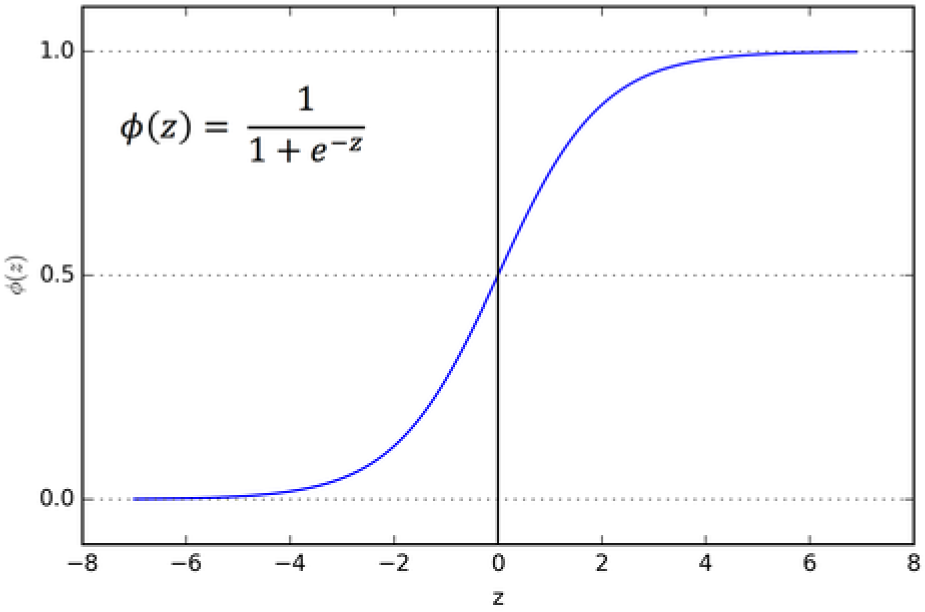
It is clear that b is just a real number, because we know by definition that the output of the scalar product/weighted sum is a also a real number.

The output of a neuron/perceptron is just a number. We give a vector as input, and returns a number. The weights and the bias are the parameters of the neuron.

However, there is a final thing, an activation function.

Do you see how the previous operation is a linear function (represented by two vectors and a number). This is no good, because with the neurons we want to form a very complex operation (the NN). If the neurons are just a vector-vector multiplication, no matter how much of them we have, we can just simplifying all the neurons to one. This is the same principle behind the matrix multiplication, if you have many matrices multiplying, they turn out to be just one matrix.

Thus, we need to add a non-linear function to the neuron to add complexity. We can for example use the sigmoid function:



There is something peculiar about this function also, the outputs are between 0 and 1, thus, they are normalized. If we use this activation function, the outputs of the each neuron are going to be a number between 0 and 1.

There are several activation function that are used. Here there is a list.

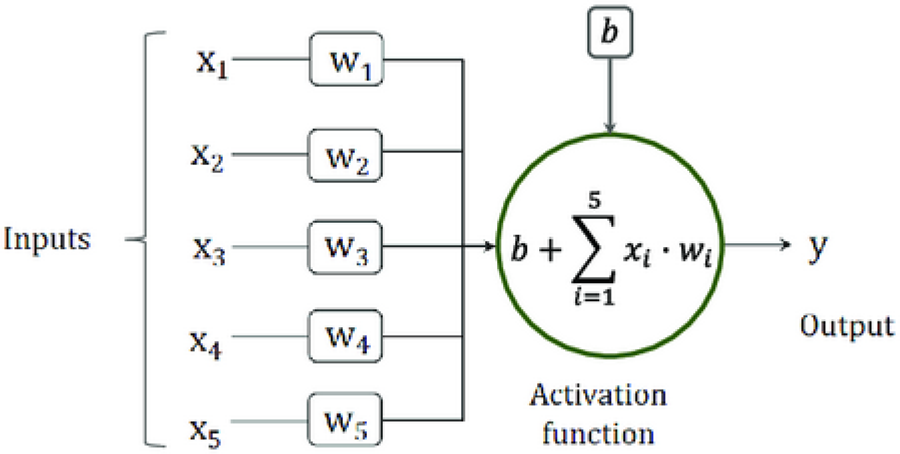
We can also use the binary step one, as Pedro points out in the slides:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

The normalization of the output depends on the activation function used.

TLDR: We use a non-linear function on the neuron to add complexity to the operation and normalize its output.

Example for neuron with a 5 element input vector:



The activation function is applied to the $b + (\boldsymbol{x} \cdot \boldsymbol{w})$. Easy.

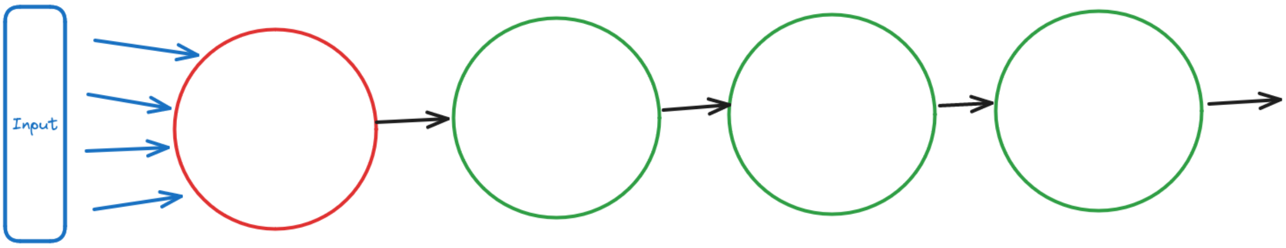
To summarize, what a neuron does is:

$$\sigma\Big(b + (\boldsymbol{x} \cdot \boldsymbol{w})\Big) = \sigma\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

Where σ is the activation function and n is the size of the input vector.

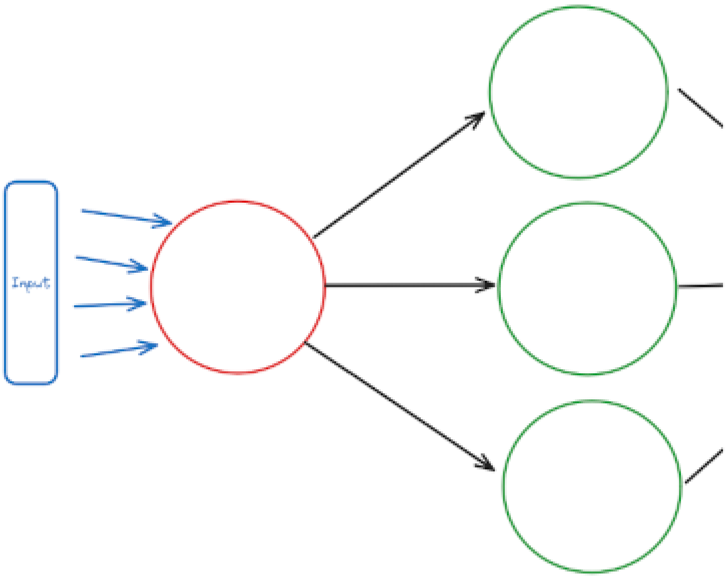
Layers

Before, I mentioned that we need to have many neurons together to form a more complex operation/function. The output of one neuron is input of the next one. Thus, we can make a chain of neurons concatenating them:



You can see that there is an input vector in blue, that goes into the red neuron. However, the neuron only outputs one number (black arrow), thus the following neurons only output one number also. This is not very efficient, we take a 4-element vector and turn it into a number. There is a huge loss of information.

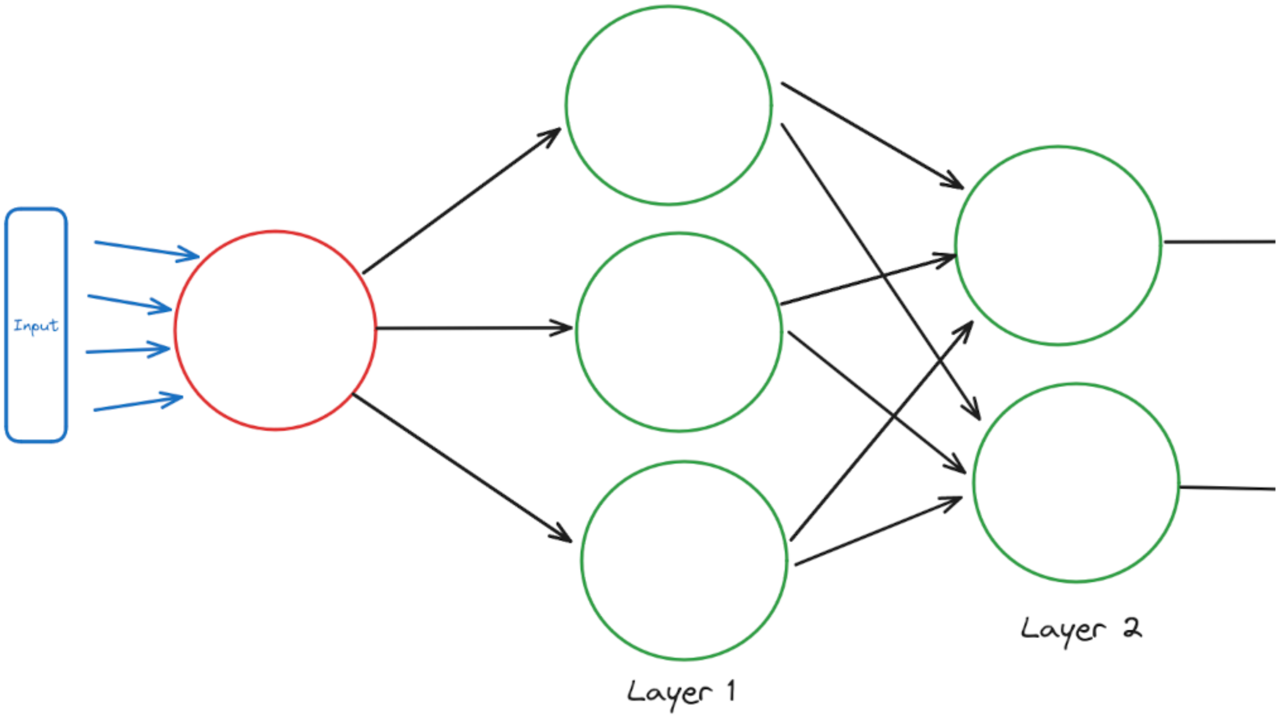
We can remediate this problem by doing a network of neuron, not a chain. The input of the red neurons can go to three of the green ones for example:



Note that is the same output in all tree, the same number goes into each one.

We call the trio of green neurons a layer.

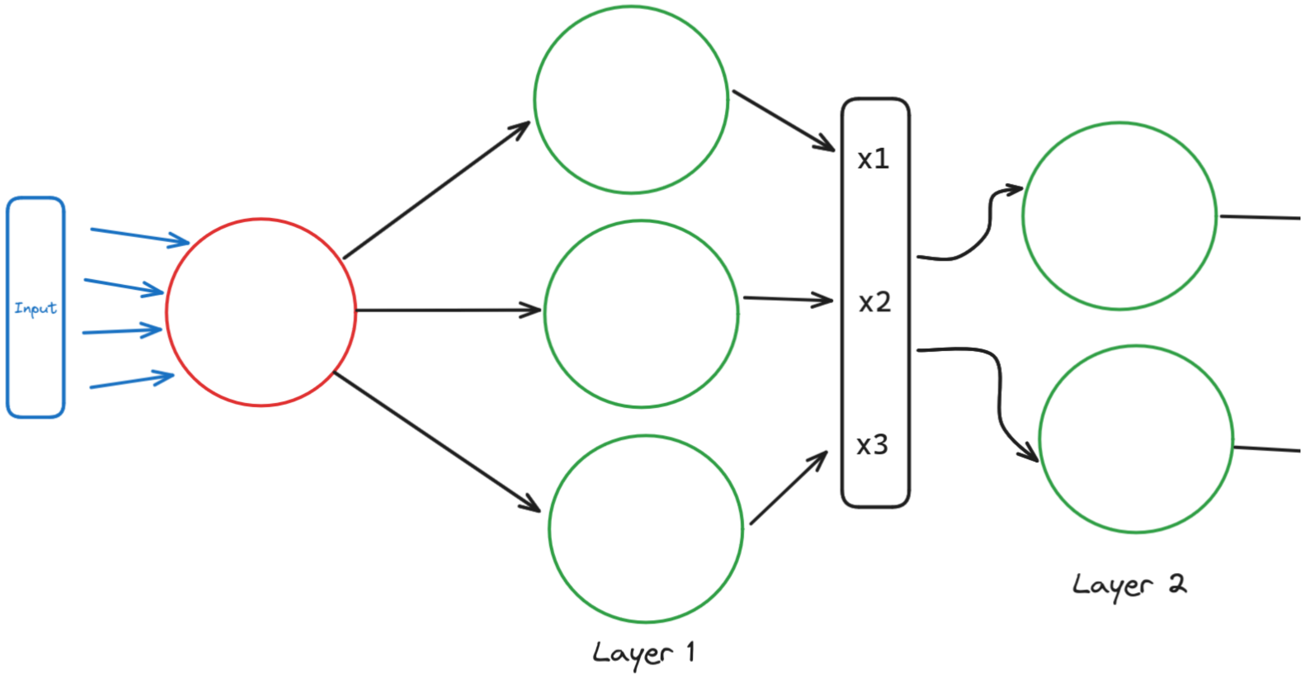
Now we can expand the Network further by adding an additional layer:



Now things get really interesting, the output of each neuron in layer 1 goes to each neuron in layer 2. Thus the latter ones have 3 inputs each.

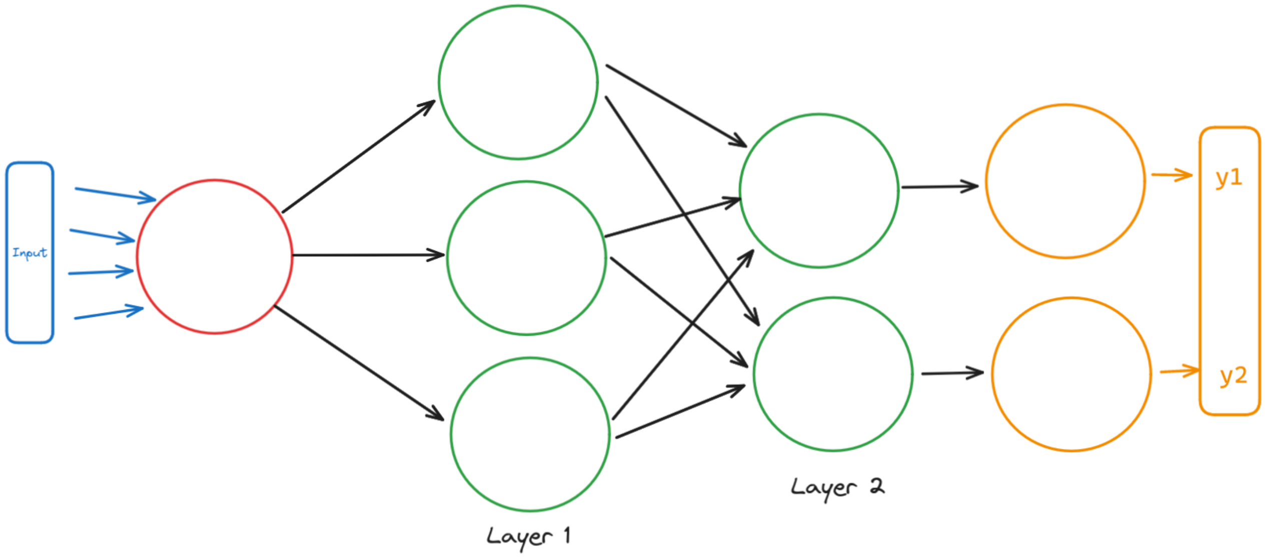
Before going further into the explanation, do you see how there are vectors everywhere?

The output of layer 1 is a 3-element vector. And the neurons on the second one get as input the same 3-element vector:

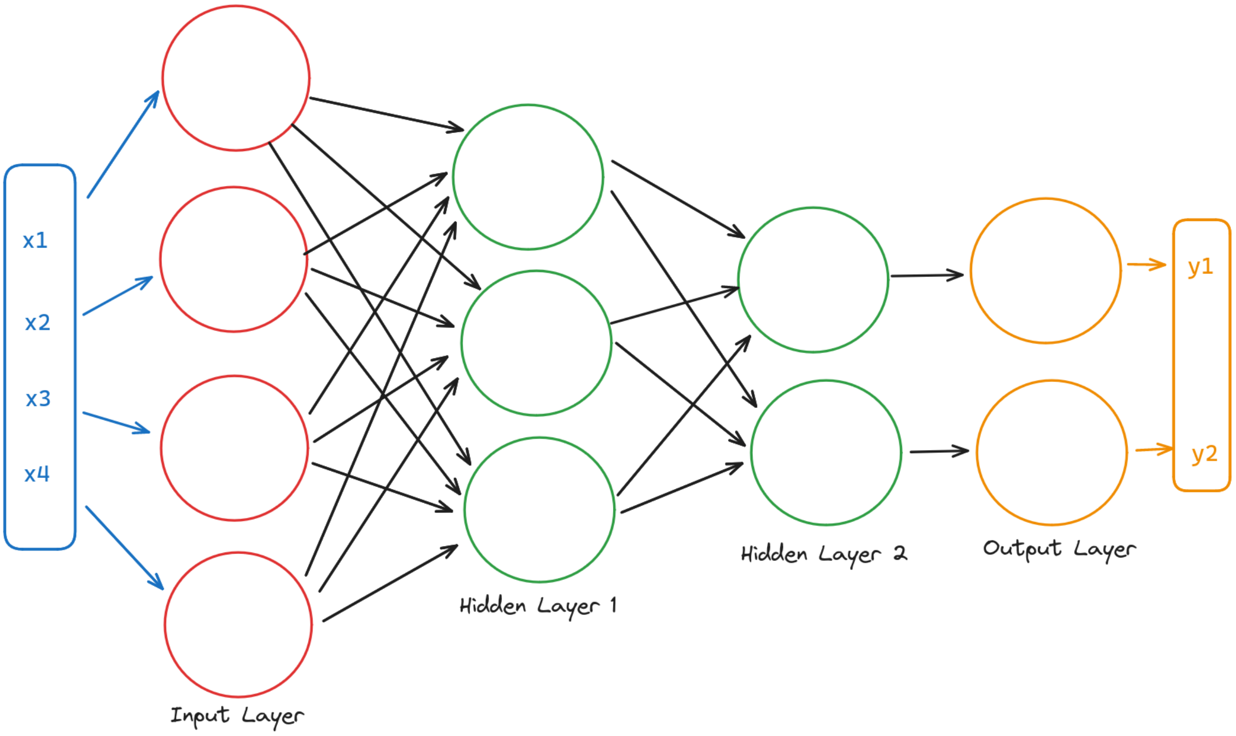


I told you was all linear algebra!

Finally, we put two more neurons that make the output vector (marked in yellow):



You can see how clearly the neurons of different color have different purposes. We have the greens that are in between the input and output, we call this the hidden layers. The red one is the input layer, and the yellow/orange the output layer:



I lied to you a bit before, usually, each element of the input vector has its own neuron, that why how the input layer has 4 neuron, one for each number in the vector.

TLDR:

> There are 3 types of layers in a NN:
>
> - input
> - hidden
> - output

> We give a vector as input and receive another one has output. The layer pass a vector of their outputs to the next layer. Each output comes from a neuron.

The neural networks are just many layers with many neurons each one connected between them:

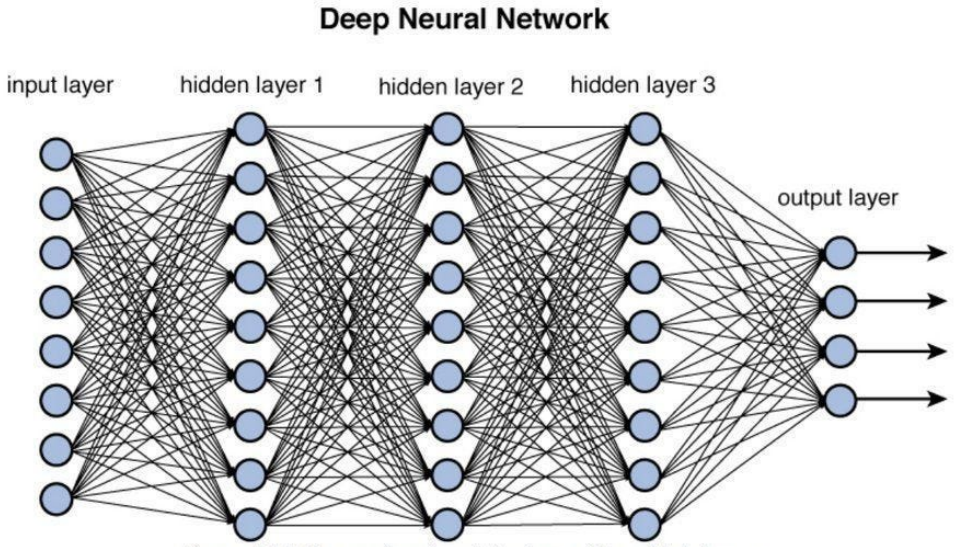


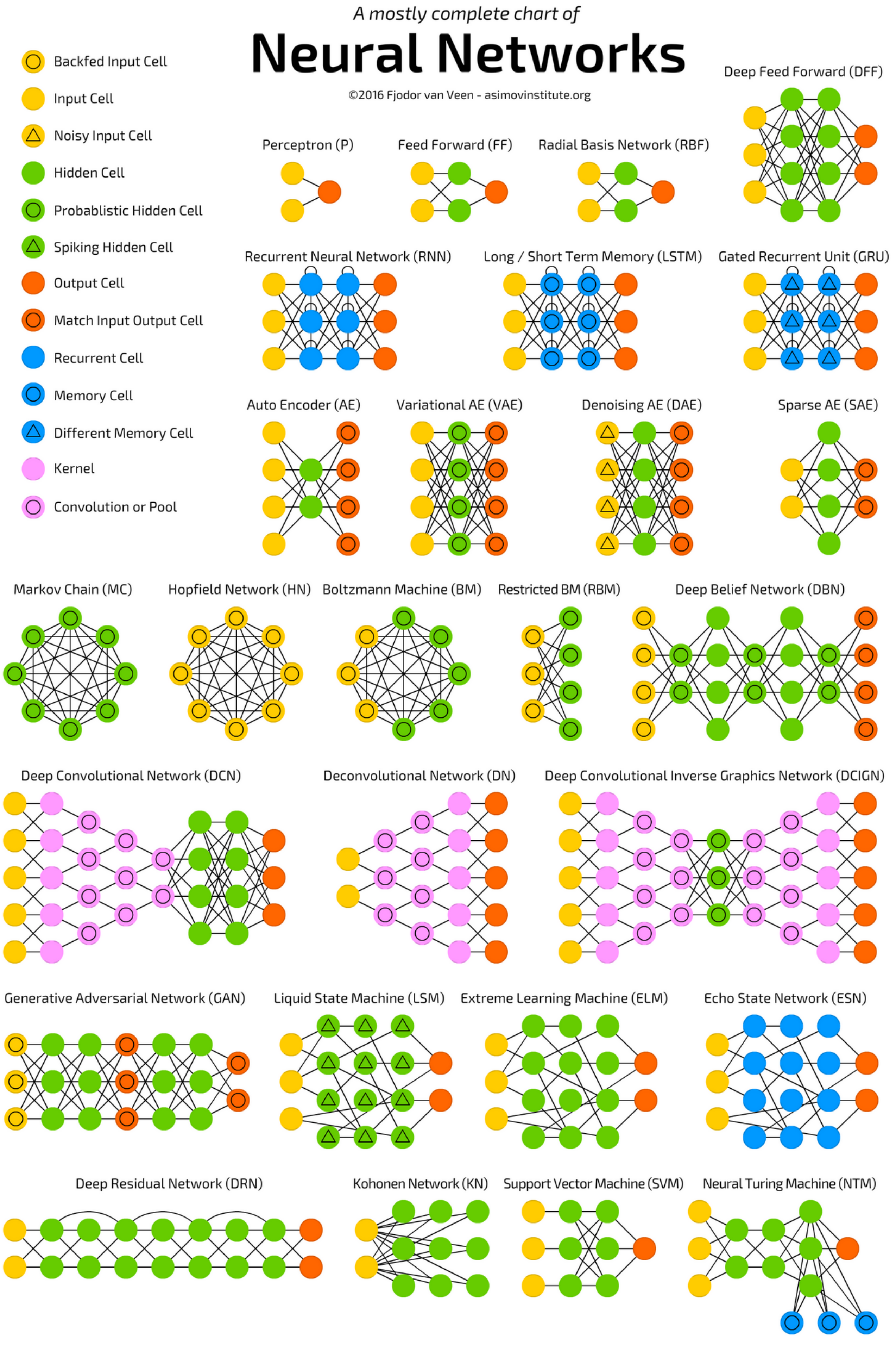
Figure 12.2 Deep network architecture with multiple layers.

What we call Deep Learning is just a NN with a large number of layers. It is somewhat of an ambiguous term.

How my question is: Do you see how NNs are like Lego? There a bunch a different types of pieces that you can arrange in several ways to form vastly different NNs.

But wait.. "you said that there are several types of pieces", does that mean that there are several types of neurons? - You probably now.

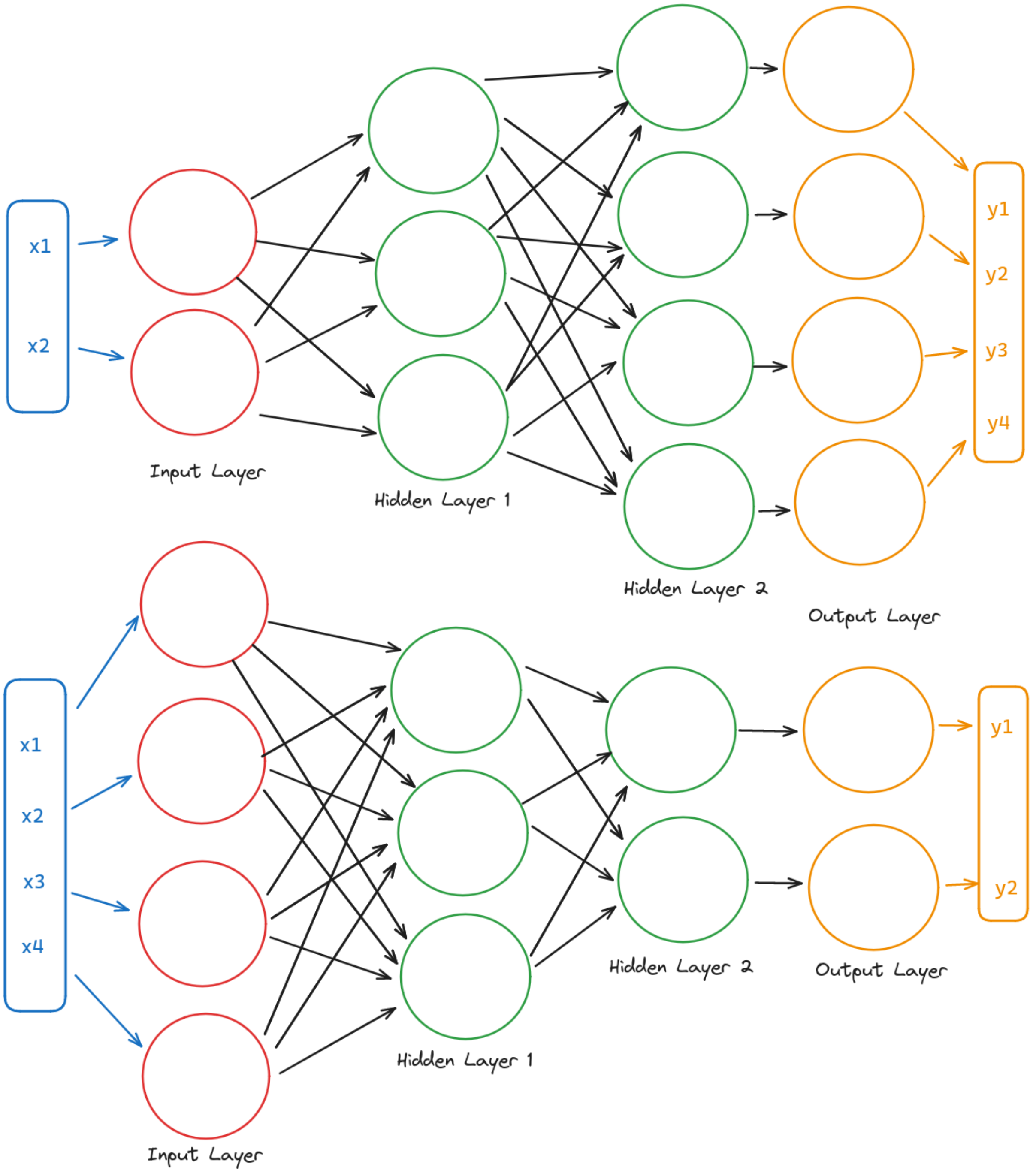
The answer is yes, there different types of layers and neurons. The type of neuron explained before is the most simple one. With this chart you can see how many types of neurons/layers/networks there are:


A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

Some of them are very important, other ones are outdated. The one that we have looked is the Deep Feed Forward network, also called fully connected network.

Take this chart as a reference for how to grasp the diversity of NNs that there is. If you want to know more go to the blogpost of the chart.

As final note, think about how the size of the input and outputs of a NN define its use. If the output is smaller than the input, that means that the NN is "compressing" the information in some way. If the output is larger, the NN is "adding complexity":



How you can think about how we can express these NNs in the form of matrix operations. Exercise left to the reader.

Backpropagation

> How are the NNs trained?/How do the NNs "learn"?

I have two answers to this question, a short one and a long one.

TLDR of Backpropagation (short one)

(TLDR means *too long didn't read*)

There is a function that we want to optimize with the NNs, it is called "loss" ($\mathcal{L}(\cdot)$). There are many types, the main one is Mean Squared Error (MSE):

$$\text{MSE}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - x_i)^2 \qquad \boldsymbol{y}, \boldsymbol{x} \in \mathbb{R}^m$$

$\boldsymbol{y}$ is the ideal output, and $\boldsymbol{x}$ the one that we get from the NN.

Each parameter (weight and bias) of the NN is outdated via the gradient descend method. Taking the derivative of the loss function $\mathcal{L}(\cdot)$ with respect to the parameter:

$$\theta_i^{t+1} = \theta_i^t - \mu \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \theta_i}$$

We just update the parameter via subtracting a small number that depends on the derivative ($\mu \approx 10^{-3}$). It is an iterative process.

To calculate each derivative we use backpropagation: The error directly depends on the output layer, thus we update the parameters of the output layer first.

The next layer to be optimize/updated is the directly connected to the output, thus, the last hidden layer. We use the error (derivatives) calculated before to calculate the ones of this layer. This is repeated until we reach the input layer.

We "backpropagate the errors" to update the parameters and optimize the loss function.

To do this we first need an output of the NN (to calculate the loss), thus each iteration step looks like:

1. Take an example of the dataset
2. Input it into the NN (forward pass). `Input -> Output`
3. Determine the error that the example "has" and update the parameters via backpropagation (backward pass).

The error is calculated by backprop process and the updating of the parameters are done by gradient descent. The backprop gives us the derivatives that gradient descent uses to optimize the NN.

These are repeated until we reach a minimum error or we exhaust our computing resources.

TLDR of the TLDR:

> - The error is `correct output - real output`.
> - Backpropagation is used to gradient descend in the parameter space
> - Gradient descent updates the parameters
> - The training is done by using many examples in an iterative procedure:
>   1. Take example
>   2. Execute `Input -> Output`
>   3. Determine error and update the parameters
>   4. Repeat until minimum error or exhausting resources.

This is all you need to know (forget the math for the exam btw).

It is a long story (long one)

Do not have much time to write and there are smarter people than me that explain things better. Watch these two videos:

- Gradient Descent Explained
- Backpropagation Explained

Deep Learning