Neural Networks and Deep Learning

PROF. *Alberto Testolin*
UNIVERSITY OF PADOVA

WRITTEN BY: *Francesco Manzali, Francesco Ferretto, Andrea Nicolai*

ACADEMIC YEAR 2020/21

Compiled on September 20, 2021

CONTENTS

Introduction

If you find any mistake/typo/missing thing, please write at *francesco.manzali@studenti.unipd.it* (or even just for feedback).

Francesco Manzali, 29/09/2019

This is my humble attempt to complete the notes previously written by Francesco and Francesco, I hope they are going to be useful for any student who is attracted in this new and interesting subject, or anyone else who needs to prepare for the exam. Please note that, due to the quickly evolving nature of the topic, some results might seem outdated for the next years, as well as Professor might change the course schedule.

Andrea Nicolai, 17/09/2021

Acknowledgments

CHAPTER 1

INTRODUCTION

1.1 Introduction

(Lesson 1 of 29/09/2020)
Compiled: September 20, 2021

Deep Learning is a leading framework of Machine Learning, which is nowadays successful for many applications. It's mediated by the development of techniques implemented in structures known as *Artificial Neural Networks* (*ANN*s) through what is called *representational learning*[1]. ANNs are the result of the inspiration got from nature, in particular from *Biological Neural Networks* (BNNs), which are largely acquainted with Cognitive Science since the '50s. Through the last years ANNs have acquired a profile defined by multiple disciplines which still inherit the biological point of view.

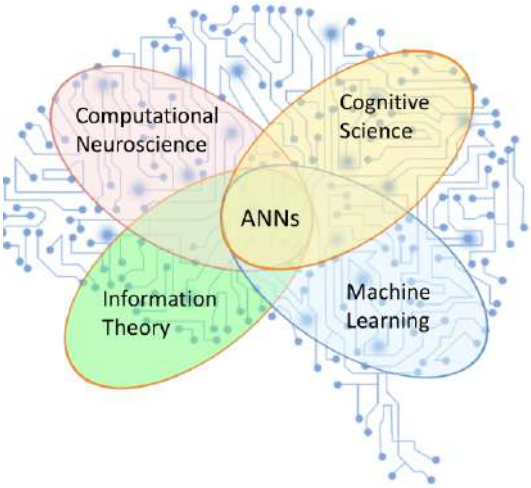


Figure (1.1) – Different areas that contribute to the ANN

The ANNs jointly inherit the following descriptions, leading thus to **multiple definitions**:

- *Computational Neuroscience*: the brain has an hierarchical structure (which is embodied in DL by adding multiple layers to a given architecture), e.g. the

[1] ∧*Feature learning* or *Representation learning* is a set of techniques that allows a system to automatically discover the representations needed for feature detection or classification from raw data, more on this on the next lectures

Cerebral Cortex processes the sensorial inputs from multiple layers of stimuli processing in a hierarchical fashion;

- *Information Theory*: deals with the ways in which neural biological structures encode information, process it, ...
- *Machine Learning*: aims at finding the *best* artificial architecture which guarantees a better performance in terms of prediction and modeling, so, without caring too much about the biological interpretation of ANNs;
- *Cognitive Science*: it studies the physiology of the brain and how it behaves when it learns, memorizes, reacts,...

All this viewpoints are only a partial example of the number of fields that merge in the creation of such a complicated subject.
This course will cover the **theory** and **practice** of artificial neural networks, with an emphasis on **deep learning** and **computational modeling**. In particular, it is interesting to examine how neural networks work in nature and learn better architectures while trying to model them.
The focus will be on **foundations**, not *current practices* or *specific implementations* (such as common deep learning architectures, e.g., VGG-19). This is because while common tricks will eventually phase out when newer (and better) methods will be invented, the foundations will remain the same and will provide a sufficient base to start learning the specifics.

1.1.1 Deep Learning

Deep Learning is a field with roots in many disciplines, from machine learning to information theory, cognitive science, and many more.
At its core, it is a way to **minimize** an error function $\mathcal{E}$, for example through gradient descent:

$$\Delta w_{ij} = -\frac{\partial \mathcal{E}}{\partial w_{ij}}$$

In this case, we have a model with many parameters w_{ij}, and want to "tweak" them so that an objective function $\mathcal{E}(\boldsymbol{w})$ is minimized. In a sense, $\mathcal{E}$ represents how much the model is "far" from whatever we want to do: *learning something* is equivalent to *minimizing errors*.[2]

Historically, ideas about deep learning have come from **neuroscience**. One recent example is that of Demis Hassamis, the CEO of Google's DeepMind (a company at the edge of Deep Learning research which developed, among others, the bot which can play *AlphaGo*), who originally studied how patients with brain damage in the hippocampus, i.e. the site where memories are stored, are unable to imagine new experiences besides having no memory about past. He then used this unusual link for developing new algorithms able to both *store memories* and *generate new ones*.

[2] ∧This process will be guided by the *backpropagation algorithm*, which tailors the right contribution of each weight in the network

This is even more clear looking back at the origins of Deep Learning, where Cognitive Science, Physics, Information Theory, Computer Science and Psychology fields merge with each other:

- 1952: Biophysical model of action potential (Hodgkin, Huxley) to describe how information was transmitted
- 1958: Perceptron model (Rosenblatt)
- 1961: Efficient coding theory (Barlow)
- 1972: Formalization of learning by reinforcement signals (Rescorla, Wagner)
- 1982: Associative memories (Hopfield)
- 1985: Spin glass theory of associative memories (Amit, Sompolinksi)
- 1985: Generative neural networks and Boltzmann machine (Hinton, Sejnowski)
- 1986: Backpropagation algorithm (Rumelhart)
- 1990: Simple recurrent networks (Elman)
- 1994: Synaptic plasticity and catastrophic interference (Amit, Fusi)
- 1996: Sparse coding (Sejnowski, Olshausen, Field)
- 1997: Dopamine and Temporal-Difference reinforcement learning (Montague, Dayan)
- 1998: Convolutional neural networks (LeCun)
- 2002: Contrastive Hebbian Learning (Hinton)
- 2006: Bayesian theory of neuronal responses (Ma, Pouget)
- 2006: Deep unsupervised learning with belief networks (Hinton)
- 2012: Deep supervised learning with CNNs (Hinton, Bengio, LeCun)
- 2014: Deep unsupervised learning with VAEs and GANs (Welling, Goodfellow)
- 2015: Deep reinforcement learning (Google DeepMind)
- 2018: Transformers (Google, OpenAI)

As you can see, up to the 2000s most of the discoveries were made by studying physical models of biological systems. Then, the engineering kicked in, and the higher computational power and availability of data led to more and more complex algorithms. That is why we will start our discussion with concepts from **biology**, in order to understand the basics shared among almost all models.

1.2 Biology

"Why is a nervous system (a brain) needed?"

In nature, we can find examples of *sensory* and *reactive* behaviors without *any* kind of complex brain. For example, a carnivorous plant can catch a pray when it senses it. These, however, are **automatic**, not **controlled** behaviors, i.e. they are fixed, automatic, repetitive.
The presence of a *sufficiently complex* brain enables learning and control on a higher level. Simple brains, such as the ones of insects, are not very *plastic* — they are similar to the plants' case. For example a wasp will check its nest before bringing a prey inside. Suppose the prey is moved while the wasp is inside the nest. When it comes out again, it will bring the prey along, only to check *again* its nest. Repeating the action will lead to the same *repeated* and *unchanged* behavior, since it has not the capability to *store memory* thus learning what has happened.
A more complex nervous system, such as the one of a Drosohpila[3] ($\sim$ 150k neurons) is one of the largest brain we were able to successfully map. We both were able to obtain its complete genome and connectome[4]. Another relevant cases, whose connectome is currently being mapped is the Zebrafish, which is formed by $\sim 1 \cdot 10^7$ neurons. This animal, despite being only 4cm long, is instead capable of learning quite complex tasks during its evolution. But, obviously, the best example of a *complex* brain is that of a **human**.

Actually, not *all* parts of the human brain "can learn". We can distinguish various parts:

- The **brainstem** is responsible for vital functions, and mostly works automatically (not useful for learning).
- **Cerebellum**: motor control, balancing, coordination, schemas, memory. Able to learn, but not complex things and works almost for repetitive actions.
- **Diencephalon**: transmission of sensory and motor information to/from the cerebral cortex, wake and sleep rhythm, alerting, appetite, and regulation of the endocrinal system (hypophysis and pineal gland). Although it is not specifically devoted to learning, its mediating function starts playing a role in behavior.
- The **cerebrum** (or *telencephalon*) is the largest parts of the brain, containing the *cerebral cortex* and other structures. It is devoted to higher order function, such as sensory processing, memory and reasoning. This is where most of actual *learning* occurs.

[3] ∧Drosophila melanogaster, a.k.a. the *fruit fly*

[4] ∧A connectome is a comprehensive map of neural connections in the brain, and may be thought of as its "wiring diagram". More broadly, a connectome would include the mapping of all neural connections within an organism's nervous system.

In its totality, a human brain is made of 100 billions of neurons, plus a lot of auxiliary cells (e.g., glial cells). The cell bodies are contained in the so-called *grey matter*, while their connections occupy the *white matter*. These connections are *a lot*: about 170 000 km of length!
The highest part of the human's nervous system, the **cerebral cortex**, sits at the outermost layer of the brain, over an internal zone made of white matters. Its structure is laminar: there are 6 layers, each with different types of neurons, but their exact function is not yet known. Therefore, the layers in DL models do *not* correspond to this laminar structure (which is not taken into account at all), but it related to a more general hierarchical structure of the brain.
The brain is incredibly *plastic*, i.e. able to change depending on the environment. Neurogenesis (birth of new neurons) occurs mostly during embryonic development, and continues through lifetime only in specific brain areas. Most of the plasticity instead comes from forming *new synaptic connections*, a process that is always active, peaking during childhood. Concurrently, useless connections are constantly being *pruned*, since it is not energy efficient to store them. This makes learning new tasks more difficult (e.g., it is harder to learn a language as an adult), but increases efficiency. We will see as pruning is used in DL, too.
Additionally, in a real brain, not all neurons are active at the same rate and at the same time. Their action is modulated by the diffusion of *neurotransmitters*, which depends on the actual state of the emotions of a person. All these behaviors, such as plasticity, complex connections, pruning, and real-time dynamics, are not considered in current deep learning models, but are subject of *active research*.
Finally, the expression of *learning* in Deep Learning is the *process of dynamic reassignment of synapses* inside biological networks.

The cerebral cortex can be functionally divided in regions:

- The **occipital lobe**, mostly processing visual information
- The **temporal lobe**, devoted to associations, memory, language, hearing, and superior visual areas. Accepts a stream of information from the occipital lobe.
- The **frontal lobe**, devoted to spatial representation and its integration with senses (visual attention, etc). Accepts a stream of information from the occipital lobe.
- The **parietal lobe**, which is devoted to high-level functions that can be modeled through reinforcement learning.

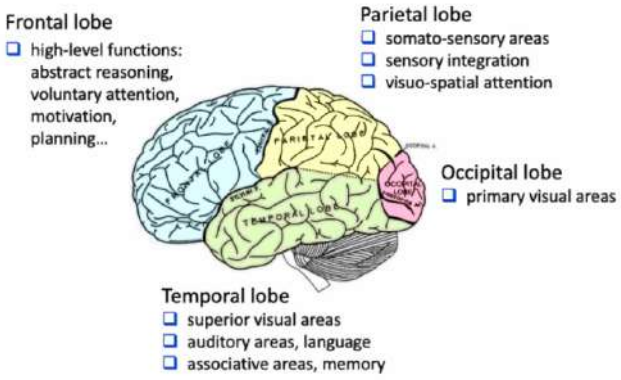


Figure (1.2) – Functional division of human cortex

In current deep learning models, there is no such division — but maybe it will be needed for creating advanced **general intelligence** in future.
Under the cortex, the **limbic system** provides emotional **feedback** and *rewards* to higher actions. For example, it can confirm if an action has been "good" (positive feedback) or "bad" (negative feedback). Perhaps, modelling a similar system could provide deep learning agents with an internal set of guiding *emotions*.

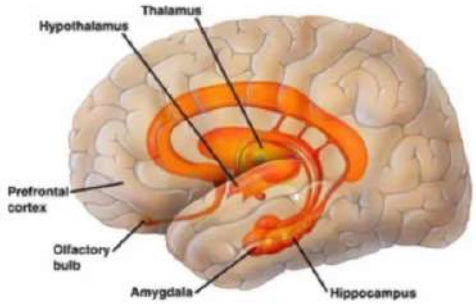


Figure (1.3) – The lymbic system

Recap of the common features/differences among biological neural networks and their artificial counterparts:

BNNs

- A brain can learn by creating new synapses, or by rearranging them;
- A brain gets rid of useless connections by *pruning* them
- A brain's plasticity is not constant in aging (becoming more efficient to carry out tasks)
- Brain contains hundreds of different types of neurons
- $\mathcal{O}(10^{11})$ neurons, $\mathcal{O}(10^{15})$ synapses
- Dynamic propagation of the signals (spikes)

ANNs - First Generation

- An ANN has *usually* a fixed number of neurons and through the minimization of $\mathcal{E}$ tailors its connections (by changing the magnitude of a weight or not assigning it at all);
- A brain emulates this process via *weight decay*.
- Only recently it starts to be considered in DL...
- A few node *types*
- 10-10000 nodes

(Lesson 2 of 01/10/2020)
Compiled: September 20, 2021

1.3 Neuronal dynamics

To study the way the brain works empirically, we measure its *neuronal activity* in various ways:

- **Single-cell recording**: a microelectrode is directly inserted into the brain, and records the rate of change of voltage with respect to time. Electrodes can measure from *outside* the cell (extracellularly) or *inside* (intracellularly). The latter is less noisy, but can damage the neuron. High spatial and temporal resolution.
- **Multielectrode recording**: to be able to record more than one neuron thus having a local network overview, a fine grid of electrodes can be implanted. Spatial resolution is less than with single-cell electrodes, since neurons do not follow the same topology of the grid. The measured signals are called *Local Field Potentials*. Arrays can still cause inflammation and damage the cells, and so cannot be implanted for long period of times. Often, the chip is implemented *in vitro*, i.e. by growing neurons on it.
- **Optical imaging**: with some organisms (e.g. zebrafish[5]), it is possible to make neurons *emit light* when they activate and then record the result as an image. Here the spatial resolution is less than with other methods (and it is more difficult to process data), but more neurons can be captured at the same time.
- **Electroencephalography**: a non-invasive method consisting in placing electrodes on the scalp, which measure the electrical activity *from outside*. This is specially done for humans brain. Spatial resolution is low, and the signals are very noisy, but there is no risk of damage. However, only superficial regions of the brain can be imaged (e.g., cortical region).
- **Magneto-encephalography** (MEG): rather than recording electrical activity, it records magnetic fields produced by neurons instead. It has a higher spatial resolution than EEG, but it is more expensive and not portable.

[5] ∧Genetically modified, making also its skin transparent

- **Functional Magnetic Resonance Imaging** (fMRI): this is a non-invasive *indirect* imaging method, which exploits the fact that brain activity is tied to changes in blood flow (active regions require more nutrients to work). It exploits the magnetic properties of *oxygenated* vs *de-oxygenated* hemoglobin. Spatial resolution is good, and also *deep* regions in the brain can be imaged, but *temporal resolution* is poor (scale of seconds), since blood flow needs some time to change.
- **Functional Near-Infrared Spectroscopy** (fNIRS), similar to fMRI, but measuring the change in blood's oxygen levels through infrared light instead of magnetic fields. It exploits the different scattering behaviors of hemoglobin depending on its oxygenation. This needs a much smaller apparatus, but works only for superficial regions, and the measures are noisy.

The brain can be analyzed at many different levels, using different approaches, thus measuring (directly or indirectly) different quantities, as one can see from Fig. 1.4. There are computational models for almost all levels separately, but a really detailed model taking into account all aspects would need lots of funding and computational power. During our course we will focus at **networks** scale.

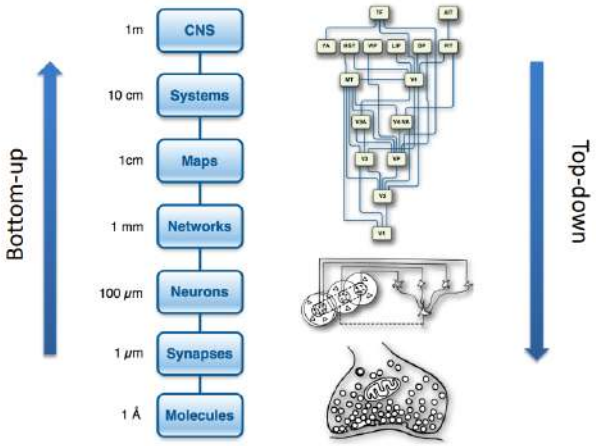


Figure (1.4) – Different levels on analysis in system neuroscience, and their scale: starting from a molecular point of view up to CNS (Cognitive and Neural Systems).

From now on, we will introduce the modeling aspects of BNNs to understand the assumptions, limitations, and approximations of such models. Indeed, we will simplify a lot, neglecting much of the complexity that a neuron itself contains.

1.4 Neuronal morphology

Let us focus on single **neurons**, and understand better which approximations we take to make its modeling efficient, nevertheless keeping it as simple as we can. A neuron is a highly specialized cell for transmitting electrical signals

over long distances, and can form complex networks where each neuron is integrating 10k-100k inputs. *Dendrites* are the part of neurons that allow for integration from many different sources of information.

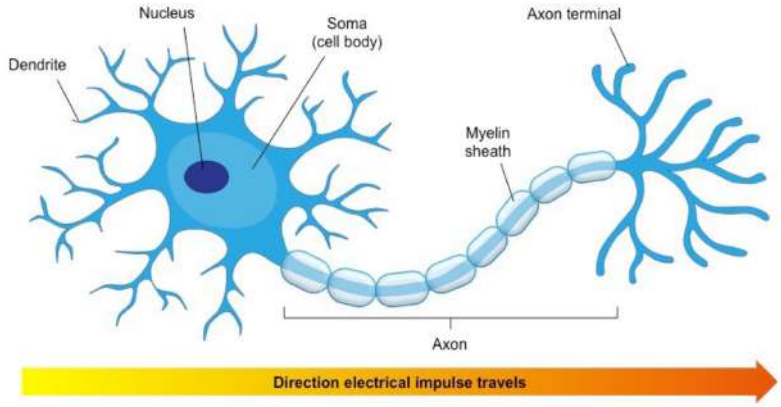


Figure (1.5) – Schema of a single neuron. It essentially consists of a single cell, whose body is named **soma**, which contains a *nucleus*. The ramifications allowing for *receiving* connections from other neurons, finally propagating them to the soma, are named **dendrites**. Conversely, **axons** are the ramification that *transmits* the signal *towards* other neurons. Finally, **synapses** are either chemical or physical junctions placed in the axon terminal, and allow for signal transmission to oher neurons' dendrites. Every neuron has *many dendrites*, but a *single axon*.

The **Myelin** is an isolating layer which covers the whole axon, to improve transmission efficiency. As an example, without myelin, the transmission speed would be only 5m/s, compared to the actual one which is 150m/s. The presence and quantity of myelin depends on the type of cell we are considering.
Electric propagation is accomplished through channels on the cell membrane, that can pump in/out ions, generating a potential difference. A signal begins by *depolarizing* the first part of the axon (closest to the soma), by opening Na^+ channels. Initially, membrane potential at rest is known to be $V_{in} = -70mV$. Shortly after, also the K^+ channels open, and *counterbalance* the potential difference, thus decreasing back the membrane potential. For small voltages increases, the K^+ current exceeds the Na^+ current and voltage returns to its resting value, hence stopping the signal propagation. However, if a critical threshold is surpassed ($V_{th} = -55mV$), the Na^+ ions dominate, and the process *explodes* triggering the propagation: the Na^+ influx spreads to the closest regions, while regions that have already "fired" *close*, allowing the K^+ mechanism to rebalance, thus coming back to the initial resting potential $V_{fin} = -70mV$. If signal has been successfully propagated, the membrane potential voltage exhibits a typical spike, whose maximum is above the threshold. This quite complex feedback loop allows making signals *less noisy*. At the end of an axon, the signal needs to be *transmitted* and "*jump*" to other neurons. This can be accomplished in two ways:

- *Electrical synapses*: very fast, but hard-wired, since they require a physical connection.

- *Chemical synapses*: slower, but flexible, allowing *learning*. Here the electrical signal arrives at the synapse, triggers ionic channels increasing the internal concentration of Ca^{2+}, which then activate proteins that send neurotransmitters outside the cell, where they are gathered by the other neurons and bind to their chemical receptors, thus triggering a polarization. Neurotransmitters can then be either deactivated or reabsorbed in the pre-synaptic cell for later firings. The **learning** is obtained via *modulation of the quantity* of neurotransmitters that are sent through the synapse, which in turn defines the strength of the connection: the larger the number of neutransmitters, the stronger the connection.

Note that the propagation of signals is usually **asymmetric**, i.e. **directional**[6]. Indeed, it does not occur that receptors send neurotransmitters back to the axon, through the synapse. However, there are cases in which it can happen in both ways, such as in *retrograde signaling* (e.g. endocannabinoids).

It is interesting to study *how* neurons decide which connections to form and which are kept once created, thus not being pruned. One hypothesis for this process is the (**minimal wiring**): the formation of axons and dendrites effectively *minimizes* resource allocation while preserving the highest information density (i.e., maximal information storage). Indeed, forming and then maintaining new connections does need some energy and resource allocation.

1.5 Single-Neuron Modeling

The first electrophysiological measurements on the propagation of action potentials were made in the early 1950s by Hodgkin and Huxley using the axons of giant squids, which are very large ($\sim 0.5\,\text{mm}$) since they need to quickly transmit the signals needed to move the mantle and escape predators. Their main discovery was that macroscopic currents in the axon could be understood in terms of changes in ion conductance in the axon membrane.
Several mathematical models exist to describe them. One of the simplest one is that of **integrate-and-fire models**, where we neglect many of the biological aspects present in reality. Here, we model the neuron membrane as a capacitor:

$$Q = C_m V$$

where Q is the charge in the membrane, C_m the capacitance and V is the membrane tension. Differentiating, we get the current needed to change the membrane potential at a given rate:

$$\frac{\mathrm{d}Q}{\mathrm{d}t} = C_m \frac{\mathrm{d}V}{\mathrm{d}t}$$

[6] ∧Contrarily, ANNs trained via backpropagation are symmetric and thus has lead to a harsh debate in the DL community.

An action potential occurs whenever the input current causes the membrane potential to reach a **critical** and *constant* **threshold** V_{th}. Hence, the action potential is modeled as a Dirac's δ function, namely a single pulse. After that the neuron has fired, the membrane potential goes back to *resting* value. One important aspect is that the **firing frequency** of the model **increases linearly** wrt the *input current*, *after that the threshold has been overcome*, without *any* upper bound. The model is really similar to the biological one, but is way much more simplified: indeed, the more a neuron is excited, the more often it fires.
This can be modelled as the following electrical circuit:

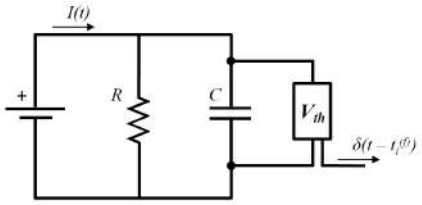


Figure (1.6)

Basically, the input current $I(t)$ charges the capacitor. If the voltage across it reaches a certain threshold V_{th}, then it is *instantaneously* discharged to produce a pulse. This means that a current value is translated into a train of pulses at a certain frequency: the higher the current, the faster the capacitor gets charged, and the more frequent are the Dirac pulses. On the contrary, if the voltage across the capacitance does not exceed the threshold, then it simply charges/discharges without any spike or resulting pulse. This response is **linear**: if we consider the *activation level* of a neuron as the rate of firing, then it will be 0 if there is no input, and then rise linearly for higher inputs (such as in ReLu). This is the simplest model one can think of, in order to preserve linearity above the threshold and the presence of a threshold itself, thus making the study of emergent properties at *network level* easier to tackle. More generally, v_{th} is a non-fixed parameter which has to be tweaked via learning.

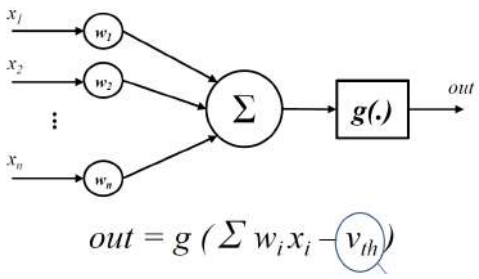


Figure (1.7) – Current inputs coming to an artificial neuron are simply summed with weights, and the output depends on whether this sum is larger than a given threshold v_{th}. If the sum increases, being larger than v_{th}, then the output signal will be larger too.

Biologically, all signals in input are reaching a neuron through the dendrites and contribute to its activation according to the synaptic efficiency. If a neuron

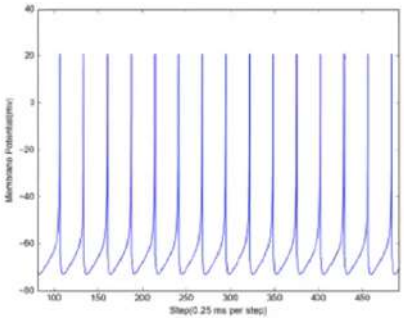


Figure (1.8) – Often, it happens that the signal in output from a neuron is grouped in coherent trains of spikes. Nevertheless, the function is continuous, despite not derivable.

is stimulated enough, then it will fire. Mathematically, this can be modeled as a weighted sum of the inputs. If the result is above a certain threshold, then the artificial neuron will produce a pulse. Its frequency will depend on how much signal (and how many signals) will be transmitted to the neuron, i.e., on how often stimuli occur to make the weighted sum larger than a fixed threshold. Algebraically, this is equivalent to integrating all the incoming currents for a certain time window.
Spikes are usually grouped in coherent trains of spikes (see Fig. 1.8). Therefore, we compute the firing rate and measure how it is changing by integrating over a small time window. More formally, we can compute the **spike-count rate** of the neuron as the *number of action potentials* in a given time window T:

$$r = \frac{n}{T} = \frac{1}{T}\int_0^T d\tau \rho(\tau)$$

where infinitesimally narrow Dirac's δ are integrated over time as $\rho(\tau) = \sum_i^n \delta(t - t_i)$. In real systems, however, the firing rate is **time-dependent**, and is obtained by counting and averaging across trials the number of spikes within much shorter time intervals Δt, i.e., computing how it changes according to the various inputs we feed the system.
However, in **deep learning** we usually deal with *activation levels*, not spikes. Indeed another interesting quantity one can measure is how **activation level** changes depending on the input signal: this defines the **neural response tuning curve**. For example, a certain neuron could be firing at a higher rate when a bar with a particular orientation (e.g. diagonal) is presented into its receptive field. In other words, this allows to measure the correlation between a neuron's activity and a specific *feature*. If we do not see any clear pattern in the firing rate *vs* some property, then, that neuron is particularly not responding to the feature being under investigation. These techniques can be employed also in examining the internal working of an artificial neural network, via gradually changing the input and seeing how activation levels change.

We want to understand how neurons encode information: neural computations might serve the following purposes, sometimes even together:

- **Amplification** of some signals, via reducing the noise

- **Compression**: i.e. encoding different signals in a more contact representation avoiding redundancy
- **Recoding**: a transformation (often nonlinear) of the original signal to highlight internal structure
- **Storing** a signal for later use

In this **bottom-up** view (senses to concepts), neurons can be regarded as passive **filters** that are sensitive to specific features and discard everything else. Then, if no input is present, the network will be "off". However, this is clearly not what happens in *real* cases: by shutting down all senses, neurons still continue to work. This means that *real* neurons try to *anticipate* new input (neurons as **hypotheses**) , and *generate* signals by themselves! This is called **top-down** information: from concepts to senses. Anticipations can then be confirmed/rejected by subsequent sensory experience, or can *dominate* and impose themselves on reality, leading to hallucinations.
It is plausible that real biological networks include both processes to be working successfully. However, models such as **feed-forward networks** are bottom-up only: if no input is given, everything remains at rest.

Since biological systems tend to be energy-optimized by evolution, it is reasonable to think that neural encoding should be efficient, i.e., such that frequent messages are short and rarer, but more informative, signals are longer.
The basic idea underlying an efficient coding principle is that: learning that an unlikely event has occurred is more informative than learning that a likely event has. This translates into the fact, as an example, that to encode the letters which are most used we use less memory (*"e"* in English), while for less common ones we use more.
This helps us to quantify the amount of information a certain event may have: likely events are assigned with low information content, while certain events are having nothing at all. On the contrary, unlikely events should be assigned higher information contents. Whereas, two independent events should have additive information, one with respect to the other.
In information theory, this can be expressed by saying that the likelihood of a message $P(x)$ is related to its self-information content $I(x)$:

$$I(x) = -\log P(x)$$

The **Shannon entropy** of a distribution is defined as the expected amount of information in an event that is drawn from that distribution, that is to say:

$$H(x) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log P(x)]$$

Moreover, it gives a lower bound on the number of bits needed, on average, to encode symbols drawn from that distribution. For example, the distribution with the largest entropy is the uniform one, being unpredictable which event

will occur next. However, as events become more predictable (e.g., according to a Gaussian distribution), the entropy decreases up to a minimal value.
Then, common sensory signals should be assigned to *cheap* brief codes for neural spikes. But this necessarily means that the nervous systems needs to adapt its internal code depending on experience, by **learning** the **statistical** structure of the **environment**.
Experimentally this is what happens: cheap codes are assigned to more frequent sensory signals. For example, the retinal ganglia can adapt to the external level of light by subtracting the average amount of light entering the pupil from the sensory image. In other words: at night, the information about the "dark" will not be transmitted, since a large part of the image will be of such color. Moreover, neurons in the primary visual cortex (V1) have tuning curves similar to those of filters optimized for efficiently coding natural images, and this also has been found to happen in the cochlea for hearing and filtering out the most occurring natural noises.
Latest models consider now that the brain itself can perform **statistical inference** in an optimal Bayesian way. The perception is characterized as unconscious statistical inference, where more uncertain sources of information should be relied upon less: if one can choose between haptic and visual senses, with the latter giving more precise information (i.e. more peaked posterior) rather than the former one, the information acquired visually is relied the most having less variability. However, if some "visual noise" is introduced, the haptic sence will be the most relevant information. The brain is therefore able to perform probabilistic inference.
Another important aspect that is trying to be coded nowadays in computational models is the **predictive** one. When some information coming from sensorial inputs is conveyed as a continuous stream to the brain, it also tries to run some **statistics** over it. Therefore, when we are facing a situation which is already known we can predict what will happen next, provided we have some statistics about it. For example, it can be shown that in the statistical structure of a language, where some letters are quite always followed by some others, the electric potential of the brain *changes* has a spike when some incongruences occur, for example, due to grammar errors.
Hence, it was proposed that sensory circuits might be transmitting **prediction errors**, rather than input signals: indeed the flow of information may occur only in the case of a wrong sensory prediction, while no relevant information is added when a prediction is correct. However, there are some underlying assumptions for this proposal, such as that the input signal can be predicted (i.e., the environment has a limited entropy) and successfully learnt by the brain.

1.6 Neural Networks

(Lesson 3 of 06/10/2020)
Compiled: September 20, 2021

When neurons are *connected* in a network, they have to *communicate* and *cooperate* to encode and process information.
There are several ways to encode information by means of several units:

- **Localist representation**. The simplest way is just to assign one entity in a **single** neuron. In this way, when *recalling* that unit, only one neuron will be active. If we store all activations in a vector, then only one is active (firing) at a time (*one-hot encoding*). In other words, neurons encode **orthogonal** features.

  This kind of representation uses less energy since fewer units are active, but needs a lot of space (i.e. neurons). Indeed, to store n different things, we need n neurons.

  Since a single unit is active at a time, there is no interference between different signals. However, slight changes in the neurons may activate very different units.

  There is some evidence that this happens in nature. For example, in the *grandmother neuron* hypothesis, we suppose that there are neurons in the human brain which activate only in response to a single very specific input, e.g. *seeing a picture of your grandmother*.

- **Distributed representation**: each entity is encoded by a **pattern** of activity distributed over many neurons: every neuron is in turn involved in representing many different entities. This needs fewer units to represent lots of things, and also allows computing *similarities* by seeing how much an activation vector is *close* to another.

  Theoretically, if each neuron can encode 2 states, then n neurons can encode (at maximum) 2^n entities. Realistically, this number will be much lower, since it is not easy to distinguish patterns that are very close to each other.

  Since many neurons process the same inputs, this provides some resistance to noise, at the cost of more energy needed (since many units are active at the same time), and the presence of interference.

  Moreover, in this case there is some empirical evidence: trying to decode an entity from a brain activity turns out to be easier when we analyze the recordings of the whole brain, and not only a single neuron at time.

- A **trade-off** between the two representations is the **sparse distributed coding**, where each entity is encoded by the strong activation of a relatively small set of neurons (not just one, and not the total number). In biology, the coding strategy and the degree of sparseness depend on the brain area being examined.

  A quantitative measure of sparseness is the *kurtosis* coefficient of *activation probability*, which is defined as:

  $$k = \frac{1}{n} \sum_{i=1}^{n} \frac{(r_i - \bar{r})^4}{\sigma^4} - 3$$

  Visually, it means to quantify how much the distribution is *peaked* around the mean. For a normal distribution the kurtosis would be 3, and that is why we subtract 3 in the definition of k: we are interested

in the *excess* kurtosis compared to a totally random process. Indeed, the larger the kurtosis (i.e. less tails), the *sparser* the process since many neurons are not active.

Let us provide an example of **localist** (see Fig. 1.9) and **distributed** (see Fig. 1.10) encodings. Let us imagine that we want to encode only 4 different entities, which are known: a rectangle and an ellipse that might have two different orientations, namely, horizontal and vertical. In the **localist** representation only 4 neurons are needed, every one encoding every possible combinations of our input. However, note as it is impossible to "learn" new geometric shapes which are a mixture of a specific feature, such for example either a rectangle/ellipse elongated both vertically and horizontally (namely a square/circle). In that case, we would need to add some neurons! This approach indeed *cannot* scale up!

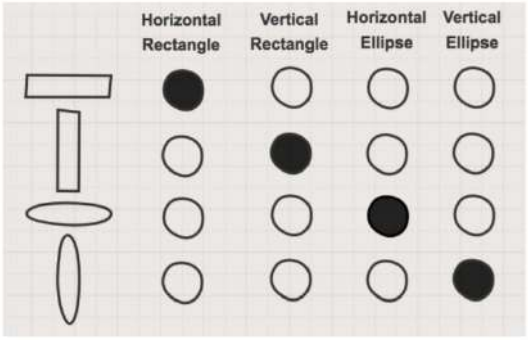


Figure (1.9) – Graphical example of a **localist** representation. Note that concepts that are *similar* are represented by vectors that are still completely different from each other, which is not a very efficient storage. Here all vectors are at the *same distance* (1) from each other.

In the **distributed** encoding, instead, we abstract the representation and store the *micro-features* of our entities. In this way, also new entities can be stored (see Fig. 1.10) such as the circle! In this representation we are exploiting the **componential structure** of the environment, thus facilitating the judgement of **similarity** between concepts allowing for some metrics. Finally somehow allowing the **retrieval with partial information**, which we will explain later on speaking about *associative networks*.

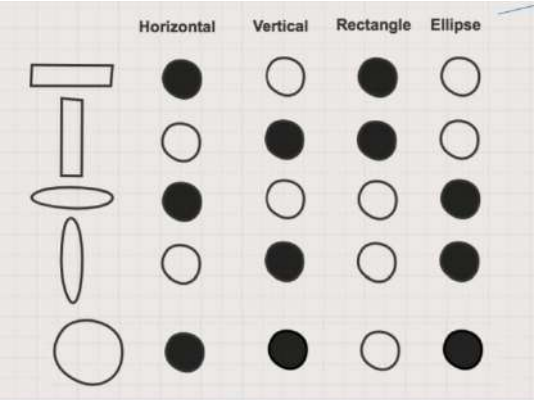


Figure (1.10) – In a **distributed** encoding, entries in the activation vectors can be seen as representing *microfeatures* of entities, for example the *enlongation* (horizontal/vertical) or the shape (rectangle/ellipse). In this way, we can *combine* different sets of features to represent new entities. Then, the *distance* between vectors encodes the *similarity* between the entities they represent.

Previously, hardware limitations constrained the analysis to only the localistic approach, since it is simpler. A seminal example of **localist language model** is the following (see Fig. 1.11), originated back to '80s, and allowed also for the retrieval of signals despite the noise.

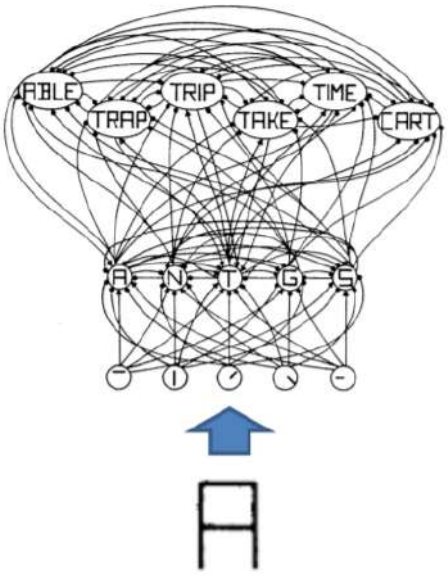


Figure (1.11) – An example of handcrafted network implementing a localistic encoding (McClelland and Rumelhart, 1981). Each unit encodes one *specific* visual feature, or one *specific* combination of features. This kind of network works well also for reconstructing text with some level of *noise*, since the *correct possible words* are encoded in neurons.

Nowadays, we can use **deep learning** methods to automatically *learn* distributed encodings. One such example is that of *Distributed Representations of Words and Phrases and their Compositionality* (T. Mikolov et al.), which starting from a dataset of texts is able to generate activation *vectors* that represent *words*. Indeed distributed representations, are able to capture complex semantic relationships between concepts. In particular, it appears that the

distances between words encode *meaning*. For example, it is possible to do the following vector addition:

$$\text{vector}('\text{king}') - \text{vector}('\text{man}') + \text{vector}('\text{woman}')$$

The result will (surprisingly) be *close* to the vector associated with *queen*. So, in a sense, this kind of distributed encoding allows a network to understand concepts it has never seen (e.g. *queen*) by encoding them as a combination of concepts which are known (*king*, *man* and *woman*). In other words, by rearranging *micro-features*, we can encode new concepts.

Let us see one example of encoding in nature, and get back to **visual system**. We know that neurons in the retina, thalamus (LGN), and primary visual cortex (V1) respond to light stimuli which happen in restricted regions of the visual field called their **receptive fields**. Then, neurons at higher levels of the visual cortical processing (V2, V4, medial and ventral IT area) respond to more complex configuration over the *whole* visual field.
We observe that retinal ganglion cells and LGN neurons are mostly selectively activated by a very basic and simple input: a circular spot of light surrounded by darkness (*on-center* cells) or its reverse (*off-center* cells). Their tuning function can then be modelled by the difference between two Gaussian filters, thereby forming a "Mexican hat". These specific filters are defined as "contrast sensitive", being activated when something falls into at least one of the two regions and thereby creating some contrast. As one can see in Fig. 1.12.

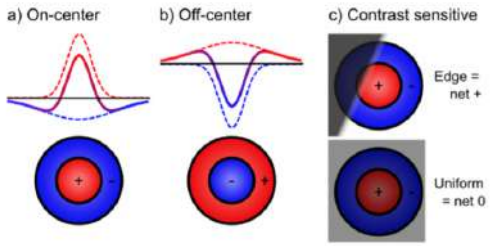


Figure (1.12) – Typical representations that activate retinal ganglion cells and thalamus (LGN).

The empirical data for the case of a *cat* is the one in Fig. 1.13.

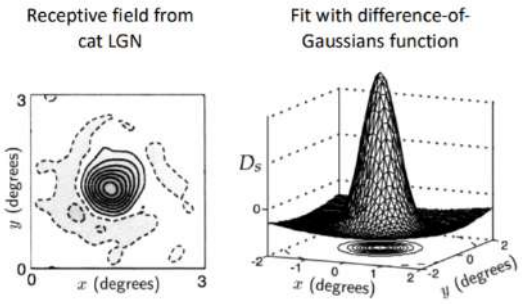


Figure (1.13)

Thus, every LGN neuron is sensitive to *contrast*: if the level of light *changes* going from the center to the edge of the receptive field, then the neuron will activate.
Thus, by connecting many LGN neurons with their receptive fields aligned, we can construct an *edge detector*, as in Fig. 1.14:

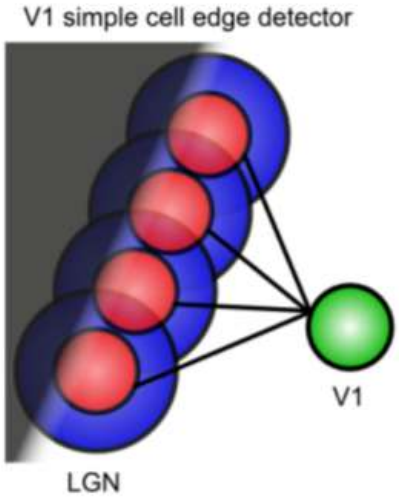


Figure (1.14) – Many LGN neurons, when aggregated, function as an "edge detector" whose information in turn is transmitted to a V1 neuron.

A V1 neuron does such *aggregation*, and every V1 neuron strongly responds to an edge at a specific *orientation*, as it happens in Fig. 1.15. Another filter, specifically created for is the so called *Gabor* filter, that consists of a Gaussian mixed with a sinusoidal curve, in such a way that a polarity can be defined in an easy way. These filters are, for example, the ones implemented in image processing software to detect edges in pictures.

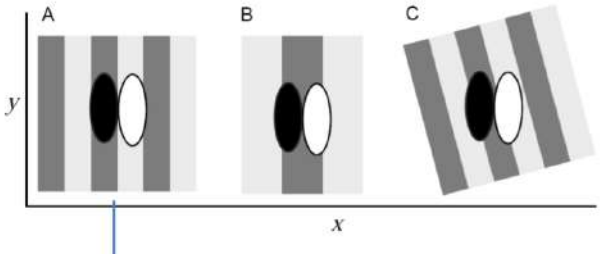


Figure (1.15) – A V1 neuron is maximally activated by specific stimuli. For example a bar with different orientations that might either completely or partially overlap some ON/OFF bands. In this case, the stimulus eliciting maximal firing rate is the left hand most picture, where the dark/light band completely overlaps with the OFF/ON area. Other ones do only partially.

Finally, by joining neurons at higher and higher levels, we can construct more complex filters. Indeed, there are neurons in the brain that are activated by specific pictures of actors. (see Fig. 1.16).

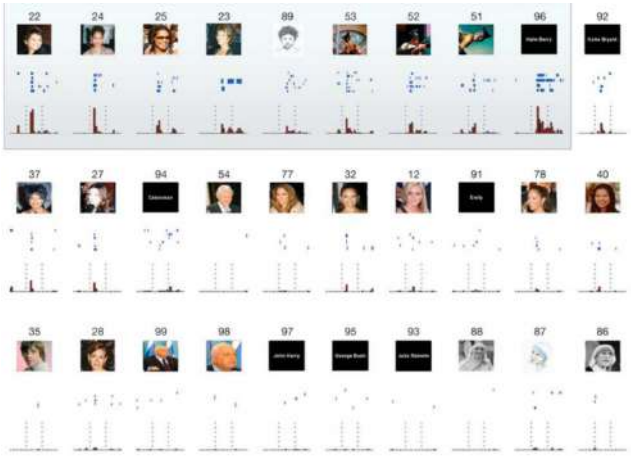


Figure (1.16) – Surprisingly, it was discovered that some neurons fire only when seeing a specific Hollywood actor, as in the mental experiment of the "grandmother neuron".

While this suggests that the brain implements a type of localistic encoding, it is not a complete proof. To properly assert so, we would need to show that every neuron responds *only* to a specific kind input, which is very difficult to do. Sometimes we have discovered that *conceptually similar inputs* still produce an activation (e.g., actors from the same TV series), suggesting that these neurons might instead be related to the TV series itself.

Linear Sparse coding

We can now go on to construct a **mathematical model**, to efficiently code natural images. In essence, we start with an image I, and we want to represent it as the *superposition* of some basis functions, weighted by the coefficients a_i which are specific to I:

$$I(x,y) = \sum_i a_i \Phi_i(x,y)$$

The idea is that the brain naturally contains a world model (functions Φ_i) which can be used to *generate* new entities and compare them to the ones received by sensory experience.
This is a *top-down* approach, and in particular an example of a **linear generative model**. Our aim is to find a good choice for the basis Φ_i forming a *complete* code, so that they are few (efficiency), and that activations are as much *independent* as possible (to reduce interference). Practically, we operate on *patches* of an image and not on the whole view: more efficient models exploit smaller basis functions (also to save space) and convolutions.
We want now to find a set of basis functions, which encode the most information. One way to do that is by using *Principal Component Analysis* (PCA), finding the orthogonal (uncorrelated) vectors that explain the most variance of the dataset. However, PCA vectors are usually *not local*, i.e. they have many nonzero entries, which is not very energy efficient.
Another possibility is **Independent Component Analysis** which, in addition of being uncorrelated as for the PCA, finds *statistically independent* basis functions, thus capturing higher order correlation.

In our case, we will focus on the **sparse coding** approach. The focus is to reduce the number of *active units*, i.e. choose the Φ_i so that the entries in a_i vectors will have a distribution peaked around 0[7] (i.e. most of the entries are null). The basis function code of this approach has to be *overcomplete*: from a large dictionary of possible functions we select a small number of active functions, this choice being optimized iteratively (computationally demanding).
Let us proceed to define this new basis. We start by defining an objective (energy) function:

$$E = -[\text{preserve information}] - \lambda[\text{sparseness of } a_i]$$

The first term deals with the model's ability to correctly reproduce data, i.e., how close can we get to I by linearly combining Φ_i. Of course, the more nonzero activations a_i we can use, the better I we can reconstruct. However, we also want to maximize the sparseness of a_i, i.e. make so that only few Φ_i are activated at once. This *trade-off* is regulated by the parameter λ: the larger the λ, the sparser the coding.

For the first term we can use as an example:

$$[\text{preserve information}] = -\sum_{xy}\left[I(x,y) - \sum_i a_i \Phi_i(x,y)\right]^2$$

which is just the mean squared error between the input image I and the reconstructed image $\sum_i a_i \Phi_i$. However, many more loss functions can be used depending on our problem (*binary cross-entropy*, etc).

For the second term that measures sparseness instead, instead:

$$[\text{sparseness of } a_i] = -\sum_i S\left(\frac{a_i}{\sigma}\right)$$

where S is a *cost* function, which quantifies the penalty of having a sparse activation (many non-zero a_i). One example is the L1 norm, which suffices the most of times.
Generally, by using few basis functions, i.e., having many a_i which are null, we decrease the amount of information preserved but keeping the "cost" of sparseness low (highly efficient). On the other hand, increasing the number of nonzero coefficients a_i leads to a larger sparseness although allowing to save more information. We thus need to find a balance between these two cases.
We want now to understand the various **effects** of different approaches in image reconstruction.

The first set of basis functions is the one returned by PCA (see Fig. 1.17). Thus, only using the first row of such an image, we can reconstruct the most of information of the original image. The main issue arising is that PCA does not develop localized features[8].

[7] ∧we impose a high kurtosis
[8] ∧we see as everything is kind of blurred out and repetitive

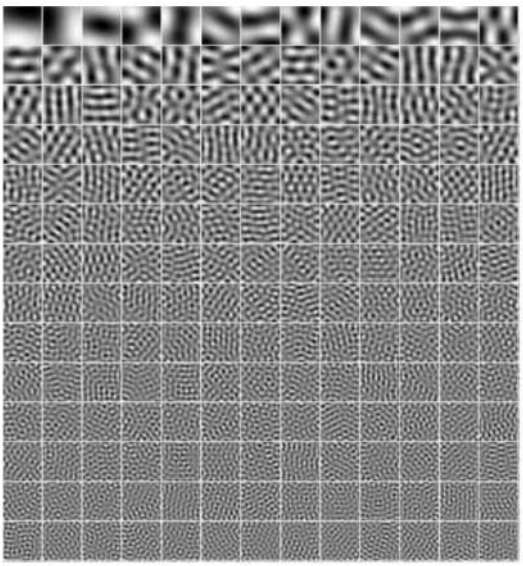


Figure (1.17) – Basis functions found by PCA. They are ordered by their importance, i.e. how much *variability* of the image dataset they can express. Note that they are all *global* features: they require inputs from all the visual field to be activated.

This issue is indeed solved by the **sparse coding** approach: basis functions are mainly related to edges and their position in the squares. One should note that among the *many* basis functions, only **few** are selected ($a_i \neq 0$) for encoding each input.

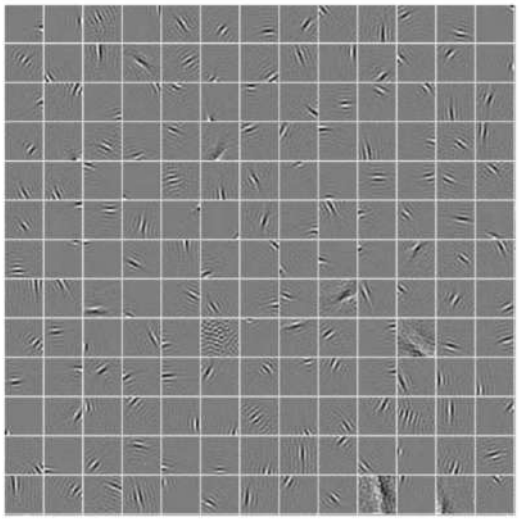


Figure (1.18) – Basis produced by sparse coding. Note how all features are **local**, and mainly describe *edge detectors* with various *orientations* and *repetitions*. White color represents positive weight, while black being negative and grey being around zero. This acts as a sort of Gabor filter.

Applying such decomposition to any input image, we obtain that the information contained in every patch can be rewritten in terms of linear combinations of a bunch of the previous basis functions.

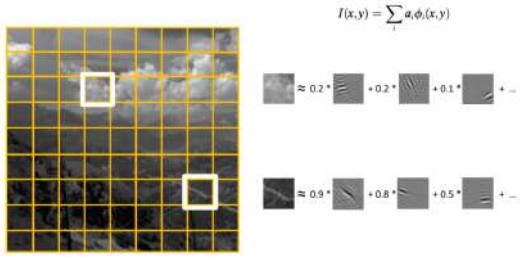


Figure (1.19) – Patches of an image can be reconstructed as a linear combination of basis vectors

For the sake of completeness, the ICA acts very similarly to the sparse coding approach, namely, being an edge detector [9].

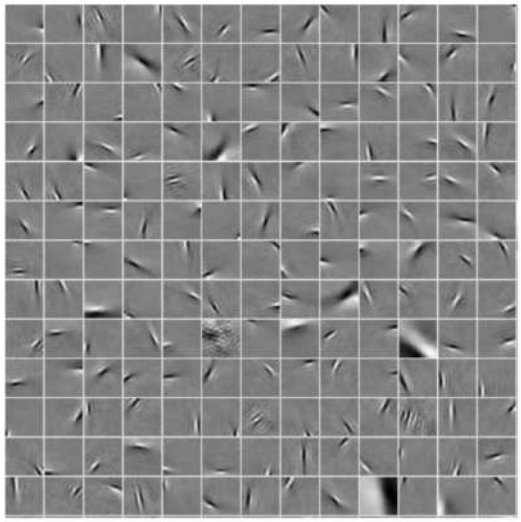


Figure (1.20) – Basis produced by ICA. Also in this case they are quite **localized**.

To simulate the development of *more complex* visual features we need more sophisticated tools, such as **deep learning** models.

To construct a general model of a network, we simply connect the output of a neuron to the input of another. Indeed, we are less concerned about single-neuron dynamics, and rather focus on neuron's **interactions**:

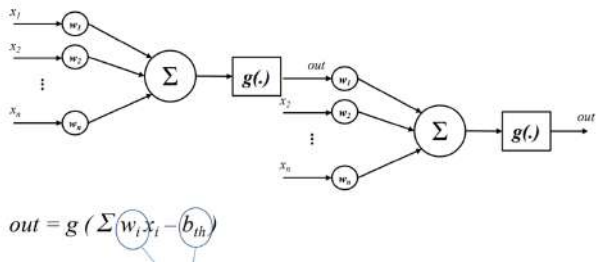


Figure (1.21) – An example of networks of neuron: the output of each neuron is fed as input to the ones linked to it.

[9] ∧being a Gabor filter, this is what mainly happens in the primary visual cortex

The **strength of synapses** (i.e. connection weights) modulate the interactions, and moreover, we assume that the activation of a neuron approximates its firing rate.

A **network architecture** is mainly composed by two aspects: its *topology* and its *directionality*. There are many possibilities for the choice of the network's **topology**, such as pointed out in Fig. 1.22.

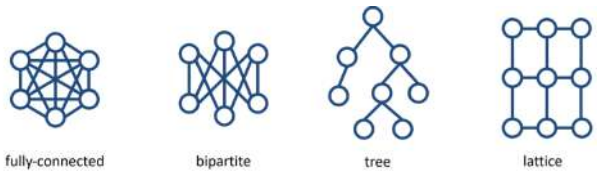


Figure (1.22) – Different topologies that might be chosen for a neural network.

Also the **direction** of connections can be defined. The simplest possibility is that of a **feed-forward** network, where activations flow in one way only (bottom-up processing), but one can add *recurrent connections* going backward allowing for *context encoding*. The more bidirectional a network is, becoming eventually *fully recurrent*, the slower convergence occurs. Thus, requiring a large computational demand to be trained.

Graphs (networks) can also be used to represent the complex statistical dependencies between several random variables. This is the case of **probabilistic graphical models**.
Here each node represents a **random variable**, and the edges define their **relations** (e.g. "causal" relations if directed, or *correlations* when undirected). By *causality* we mean that the activation of a node leads to the activation of another one. Let us analyze the case of **Bayesian network**, a *directed* (acyclic) probabilistic graphical model. The whole network represents a complex joint probability distribution (see Fig. 1.23), which can be factorized only considering the local interactions between variables. Indeed the *topology* of the graph encodes the **conditional independencies** between variables.

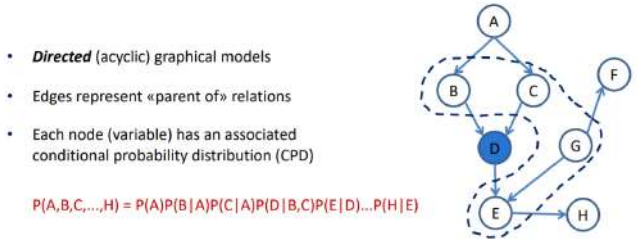


Figure (1.23) – Example of a Bayesian network, a *directed* (acyclic) probabilistic graphical model. Each node is a random variable, and the arrows indicate their relations. So, A is not influenced by anything, but B and C depend on A, and D depends on both B and C. So we can factorize the joint probability $P(A, B, C, D)$ as $P(A)P(B|A)P(C|A)P(D|B, C)$, and so on for the other nodes. Nodes indeed interact according to this *hierarchical structure*.

The **Markov blanket** of a node is the set of nodes (variables) that, when observed, makes the node independent from all the others. Taking inspiration

from 1.23, the Markov blanket of node D will be the set of nodes $\{B, C, G, E\}$: if we know their value, we can compute the value of D even though $\{A, F, G\}$ are unknown. Counterintuitively, it includes both the parents and the children of a node, and the parents of the children.

Let us analyze a **Markov network**, which is an *undirected* graph thus having symmetric edges. Every edge represents an "*affinity*" relations, and its **weight** indicates the **correlation strength** between nodes (i.e. random variables). The **joint distribution** of all variables, for such networks, can be decomposed as a product of local factors thus simplifying computations. Differently from before, the **Markov blanket** corresponds only to the immediate neighbors (parents + children of the node).

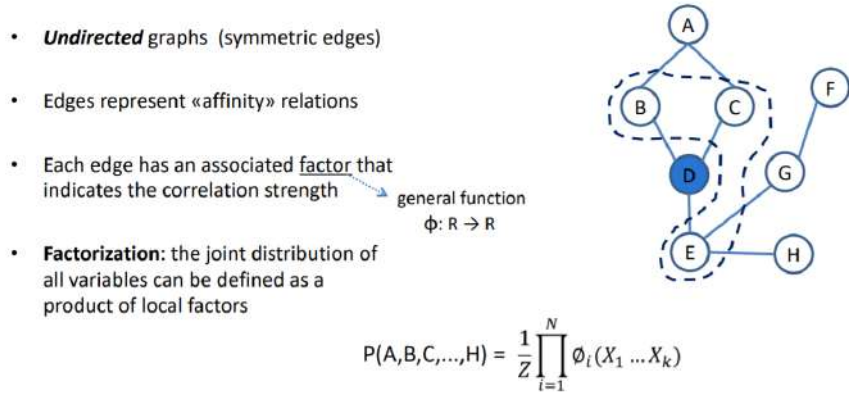


Figure (1.24) – Example of Markov network, a kind of *undirected* probabilistic graphical network.

The **parameters** to be tweaked for these models are the *connection weights*, and its **topology** might be adjusted as the result of some *a priory knowledge* (theory-driven) or, alternatively, of some *learning* process (data-driven) although it is very computationally demanding.

1.7 Machine Learning

(Lesson 4 of 08/10/2020) Compiled: September 20, 2021

Machine Learning is the field that studies how to build *adaptive programs* that are able to **learn**, i.e. improve their **performance** on a specific domain according to **experience** (i.e. a set of *samples* that are provided for *training* the software) and are able to **generalize** to new unseen situations.

This is a quite *general* definition. In fact, also simple algorithms such as *linear regression* fall under machine learning, since they can *update* the estimate of some parameters if new data are provided, and often a fit can be *extended* outside the training domain.

There are 3 main frameworks of machine learning:

- **Supervised learning**[10], where a *supervisor* is available to "teach" the system, providing a set of examples that are already correctly labeled.

[10] ∧ classification, categorization, function approximation, regression...

The output of the learning system is compared with that of the supervisor which is *always* available, and the parameters are tweaked until the two are close, i.e., until the system makes a few mistakes.

Supervised learning methods can be very powerful and efficient, but they are *biologically implausible*: when learning most tasks, a human does not always have a *teacher* available to provide corrections. In the case when a supervisor is constantly available, we shall *always* use *supervised learning*.

- **Unsupervised learning**[11], where there are no correct *a priori* labels for the data, and the task is just to find a **compact representation** of the data, i.e. find statistical regularities and use them to build an *internal world model*. This is a very general method, but quite difficult to use: for example, it is not clear how to decide *a priori which features/aspects* should be learned from data. However, it is more plausible in biological systems: this is, for example, what children do when trying to discover the world by simply abstracting it and without *any* interactions.

- **Reinforcement learning**[12], where the learner can *actively interact* with the environment, and receives *rewards* and *penalties* in response to its actions. This is quite possibly the method used by children to learn in the presence of any interaction, but reinforcement algorithms are computationally the most expensive ones.

Probably, to build a **general Artificial Intelligence**, all three types of machine learning need to be integrated into a single system.

As an example, let us imagine that we deal with an animal and try to categorize it with the three different learning frameworks we have introduced:

- **Supervised**: we simply state the category of the animal, eventually being corrected in the case of a misclassification.

- **Unsupervised**: we try to infer the animal by internally representing its characteristics and finding similarities with other previously known concepts, finally trying to infer its classification and eventually predicting other characteristics by analogy.

- **Reinforcement**: we choose among possible interactions and see what is the one that has led to a better outcome, thus finding how to best behave in such situation.

In practice, **neural networks** are one of the most versatile model to be used in machine learning. Depending on the *type* of learning, we have different possibilities:

- **Supervised**: simple perceptron, feed-forward (convolutional) multilayer networks.

[11] ∧ representaion learning, feature extraction, clusterin, dimensionality reduction...
[12] ∧ control policies, decision making...

- **Unsupervised**: recurrent neural networks (LSTM), Hebbian learning, competitive learning, self-organizing maps, Hopfield networks, generative models (Boltzmann Machines, AutoEncoders, variational autoencoders, GANs)
- **Reinforcement learning**: Q-learning, Temporal-Difference learning

Let us now discuss how to implement a general machine learning algorithm. To **train** a network we need the following ingredients:

- **Loss function**: a performance measure that will be optimized by the training process (e.g. classification accuracy). It should be a *differentiable* function, and it is not trivial to choose the right one for the task at hand (especially in reinforcement learning).
- **Experience**: a (possibly *large*) dataset to be used as basis for learning. The more difficult the tasks, the more complex the resulting model, thus larger the number of samples should be. In case of *supervised learning*, the dataset should include labels too.
- **Minimization procedure**: an algorithm to use the *experience* to iteratively optimize the *performance*, thus reaching the (possibly) the global minimum of the loss function.

To find the minimum of the loss function, we might want to exploit the **Gradient Descent** (GD). Before, one should note that the loss function optimized by a machine learning algorithm often can be *decomposed* as a sum over the m training samples. For example, in supervised learning, each sample consists of a vector of features $\boldsymbol{x}^{(i)}$ and a label $y^{(i)}$. We denote with $\boldsymbol{\theta}$ the model's parameters, and with $L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$ the *error* (loss function) the model makes on the i-th training sample. Then, the total loss $J(\boldsymbol{\theta})$ is the *expected value* of L over the distribution of data, which we approximate as the average value over the m available samples:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

Then the idea is to *tweak* the parameters $\boldsymbol{\theta}$ along the direction which *minimizes* $J(\boldsymbol{\theta})$, which is *minus* the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

Therefore, we update $\boldsymbol{\theta}$ like this:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where ϵ is a *small* number called the **learning rate**. This is the so-called **Gradient Descent** algorithm. A limitation of this approach is that computing the gradient over the whole training set, and for every training epoch, becomes unfeasible as the training set becomes larger.

Indeed, computing $J(\boldsymbol{\theta})$ like this is expensive, since usually $m \gg 1$. It is more convenient (and practically also more efficient in terms of convergence speed) to evaluate the gradient on a *minibatch*, i.e. a subset of $m' < m$ training samples:

$$g = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

And the update rule becomes:

$$\boldsymbol{\theta} \quad \leftarrow \quad \boldsymbol{\theta} - \epsilon g$$

This is, in essence, the **Stochastic Gradient Descent** (SGD) algorithm.

Moreover, on average, the gradient computed on mini-batches will point towards the same direction of the one computed over the whole training set, but at a less computational cost. Indeed, learning is a **iterative** process, and must repeated for many times (*epochs*). Specially when we start training, the more we train the better performance we achieve: this results in the fact that the loss will be ideally decreasing as the number of iterations increases, but for the SGD case being *noisier* as in Fig. 1.25, until we reach convergence and the error is minimum.

Figure (1.25) – The loss function ideally decreases as the number of iterations increases up to a minimum, for two different minimizaion algorithms: blue for GD and red for SGD.

A *gradient descent* algorithm is guaranteed to work if the loss function is **convex**, i.e. if it has a unique global minimum by choosing the *learning rate* appropriately. However, in machine learning with large network this is often not the case: the *loss landscape* is very complex, with *many* local *minima*. Still, in practice, SGD has proven to be good at quickly finding *useful* minima: may be not the *best* solution, but still a very good one.

A good practice in ML is to reserve portion of the training dataset for **evaluating** the model. This is called the **test** set, and has to be decoupled and independent of the *training* set (see Fig. 1.26). It is essentially used after the loss has reached a convergence in the training, to test whether the knowledge extracted from the training examples is able to generalize unseen data.

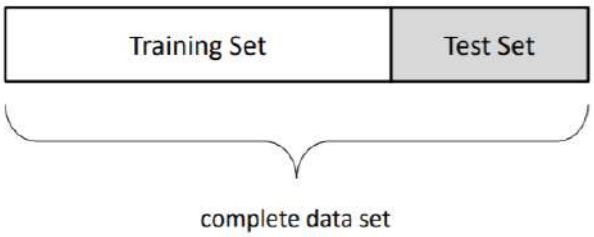


Figure (1.26) – The whole dataset is often split into *training* and *test* sets. Respectively, training and test datasets should contain $\sim 70\%$ and $\sim 30\%$ of the total sample.

Depending on the performance on the train/test dataset, we can know whether learning has been successful. The idea is to use the first part only for training, and then evaluate the model on the second one, which has *never* been seen before [13].

Ideally, we should have (see Fig. 1.27):

- A small *training error*: otherwise the model is not good enough to capture the nuances of the data (**underfitting**)
- A *test error* that is *close* to that on the training dataset. If, instead, it is significantly higher, it means that the model has *specialized too much* on the sample data, and it is not able to generalize (**overfitting**).

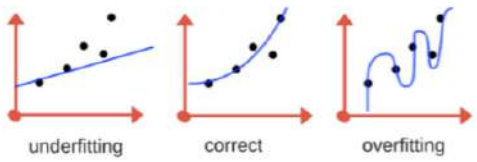


Figure (1.27) – If the model is too simple for the data (left), then we cannot get a good performance, not even on the training dataset. Conversely, too many parameters can perfectly reproduce the training data, but introducing *artifacts* that inhibit generalization (right).

To avoid overfitting, one possibility is to monitor the training process and stop it *before* any overfitting can happen. The loss function with respect to the number of epochs we have been training our model is similar to the one in Fig. 1.28. A hyperparameter related to when *early stopping* shall occur, is the so called **patience** and is defined as the number of epochs with no improvement needed to stop the training.

[13] $\wedge$ i.e. one **must not** use the testing set for training.

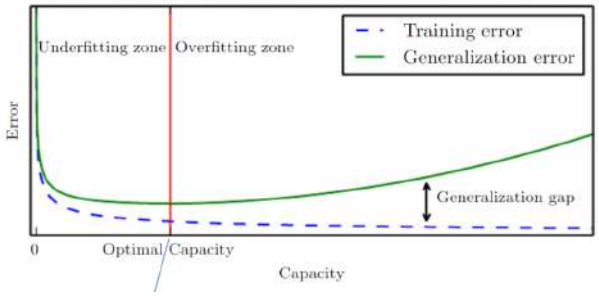


Figure (1.28) – **Early-stopping** can avoid overfitting. Specifically for this image, the training should have stopped around the red line.

The balance between underfitting/overfitting can be controlled by tuning the model's *complexity*. In general, it is better to start with a simple model, and eventually *add parameters only* if necessary. This is an application of Occam's razor: usually simple models are the best ones[14]. Indeed simpler models[15] are more likely to underfit the data, not having already the capacity to describe the training set, and clearly failing during the testing phase. On the other hand, increasing the model complexity may result in fitting the noise of the training set, which obviously is not good: the model will lack of generalization properties.

Another way to reduce overfitting, especially in deep neural networks (which often have *millions* of parameters), is to **constrain** their expressiveness, by introducing terms in the loss function which *penalize* complex models. This approach is named **regularization**. For example, we could focus the optimization over model with *small* parameters, by adding a penalty proportional to the parameters' magnitude $\Omega(\boldsymbol{\theta})$ (**weight decay**):

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where:

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|w\|_2^2$$

Due to the presence of L2 norm, this is *L2 regularization* or *Tikhonov regularization*. Thus, at every iteration we add this regularization term.

Another possibility is to penalize models that have too many nonzero parameters (as in **sparse coding**). In this case:

$$\Omega(\boldsymbol{\theta}) = \|h\|_1 = \sum_i |h_i|$$

To conclude, one should note as the loss function consists of two terms: one related to the *accuracy* of the model, while the second related to its *complexity*.

[14] ∧This can be formalized with the notion of Vapnik-Chervonenkis (VC) dimension, which provides a *quantitative measure* of the complexity of a model. Unfortunately, it is difficult to measure in practice for deep neural networks.

[15] ∧a model complexity is for example whether to fit a curve with a polynomial of 1st grade (2 parameters), 2nd grade (3 parameters) etc.

In any **neural network** there are several **hyperparameters** to be tweaked, i.e., parameters that impact the learning algorithm, but are not directly related to the model's algorithm. For example:

- How to *initialize synaptic weights*.
  - *Zero initialization*;
  - *Randomly* sampled from commonly known distributions such as $\mathcal{N}(0,1)$, $\mathcal{U}(-1,1)$, ;
  - *Heuristics* (still uses random sampled values, e.g. *Xavier initialization*);
- The learning rate and how it evolves during training.
  - Constant learning rate (usually small, such as 0.01)
  - Adaptive learning rate schedule
    - Logarithmic decrease as a function of epoch number
    - Adaptive gradient algorithm (AdaGrad), Adaptive moment estimation (Adam)
- The training pattern (SGD, batch, mini-batch)
  - One at a time ("on-line learning" or SGD)
  - In small blocks ((mini-batch)learning) - but of what size?
  - All together ("off-line" or "batch" learning)
- Other parameters
  - Regularization parameters
  - Optimization parameters (e.g. momentum)
  - Architecture: number of (hidden) layers and neurons
  - Convolutional parameters (kernel size at each layer, stride, pooling size...)

In vanilla[16] Machine Learning, there is no general rule to decide most of them *a priori*, and so it is needed to try different combinations. To test them, however, we need to reserve another *dataset*, which is called the **validation dataset**, in addition to the *training* and *test* ones previously introduced (see Fig. 1.29). Then, once the best hyperparameters are found, we can evaluate the model on the *test* dataset.

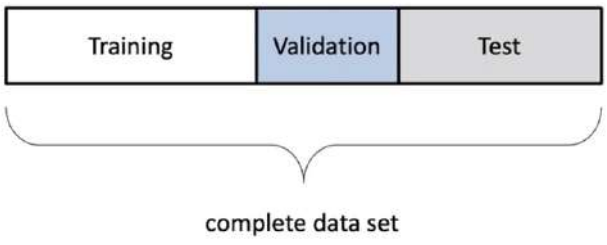


Figure (1.29) – The whole dataset is often split into *training*, *test* and *validation* sets. The latter ones is used to tune hyperparameters in order to avoid overfitting, and this process is done during training.

[16] ∧Standard

This, however, leads to a problem: if the data is limited, splitting the dataset into 3 parts (*train, validation, test*) can lead to *biased estimates* of the weights. This occurs especially when the dataset is small or, by chance, the subdivision of sets has been chosen badly.
We need the train dataset to be big, but if the validation/test are too small, then they will not be representative. One way to solve this issue is to use a 2-fold split in train/test, and then repeat the training, validation, and testing measurements on different randomly chosen splits of the *train* dataset, as one can see in Fig. 1.30. Note as, after having divided the dataset into the (training+validation)/test sets at every step of the k-fold cross validation, the latter one must not be used for training. Since this approach is quite expensive, this is done only for very small datasets: the average performance (and variance) for different choices of hyperparameters is used as metric.

Cross Validation

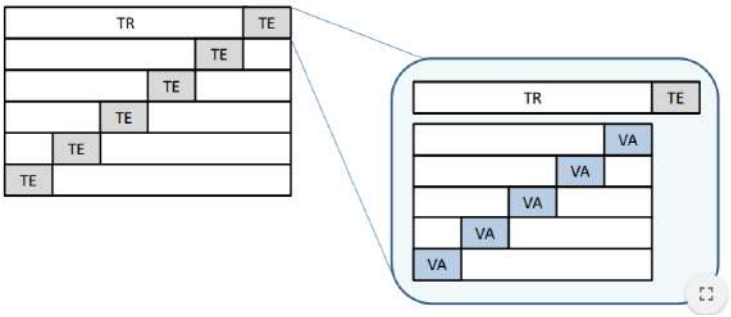


Figure (1.30) – In **cross validation**, to make so that test and validation datasets are *representative*, we can repeat the entire training/validation/testing on different *splits*. For example, start with the first row on the left. Then, we have 5 different possibilities for choosing the validation dataset inside the training set (keeping the test set fixed). We repeat training/validation/testing 5 times, over each possibility (rows on the right). Then we *redefine* the test dataset (second row on the left), and again split the rest in training/validation, for which we again we have 5 possibilities (*folds*). After all the iterations, we can be sure to have some good metrics.

When dealing with an application of Machine Learning to real world problems, we must take into account the *No-free-lunch-theorem*, i.e. we should take into account that the model cannot, in average, perform better than any other in all possible domains.
This means that we should *always* consider different models, in order to find the most suitable ones and potentially use some *Ensemble model averaging* strategy to generalize better the function underneath the process.
In fact, *cross-validation* might result in models with very different hyperparameters (e.g., different number of layers) and so:

- We can consider the set of hyperparameters giving, on average, the best performance

  or

- We can keep all different models, and combine them for getting the final prediction:

→ "committee machines" or "mixture of experts"

→ *Model Averaging*: the final prediction can be obtained by weighted average, by majority vote, ... The main idea is to train several different models separately, then have all models vote on the output of test examples;

In deep learning, this strategy cannot be implemented since it would mean to repeat the process multiple times to find the optimal parameters for a model with thousands of them (or even in the order of millions).

CHAPTER 2

SUPERVISED LEARNING

(Lesson 5 of 13/10/20)
Compiled: September 20, 2021

2.1 Supervised Learning

In *supervised* learning we can distinguish two main types of problems.

- **Regression** $f : \mathbb{R}^p \longrightarrow \mathbb{R}$ map to scalar magnitudes (*continuous* values)
- **Classification** $f : \mathbb{R}^p \longrightarrow C \subsetneq \mathbb{Z}$ map to categorical information (*discrete* values)
  - *Binary classification*
  - *Multi-class classification*

Both of this problems aim to minimize the discrepancy between the **predicted labels** (or values) via the presence of a "supervisor", via *learning*.
Both of this problems can be tackled[1] via the same - and simple - NN architecture. We now analyze the Binary classification case with the **Perceptron**, shown in Fig. 2.1.

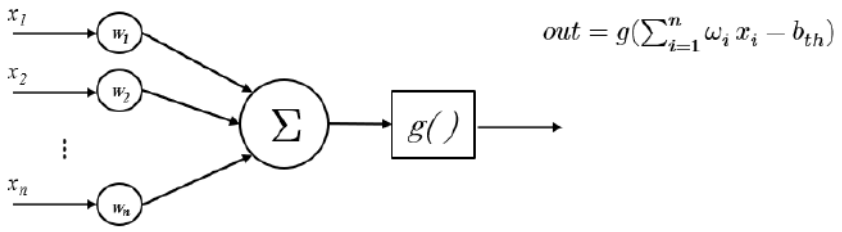


Figure (2.1) – The perceptron architecture

It is a model composed by a single neuron, in which every input is weighted according to some coefficient, namely the w_i (synaptic weights). The weighted contributions of all inputs are then summed (acting as soma's role) in order to obtain a single number. Since the output is binary, the presence of an additional function is needed, and we call it $g(\cdot)$. For the perceptron case,

[1] ∧Under *linear separability* assumption

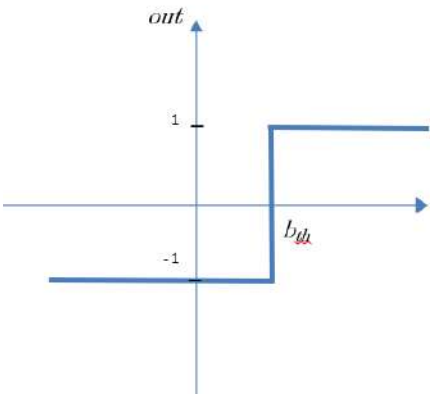


Figure (2.2) – The Heaviside function $g(\cdot)$

the latter outputs a specific value if the overall contribution of the weighted input surpasses a critical value b_{th} (thresholding "voltage", or *bias*):

$$\text{out} = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i - b_{th} > 0 \\ -1 & \text{otherwise} \end{cases}$$

Given that the *Heaviside* function $g(\cdot)$ is *non-differentiable* we should avoid it in practice.
Let us see now how we can proceed to **train** the *perceptron*.

1. Initialize all weights to zero (or to small random numbers)
2. For each training example x:
   (a) Calculate the out value
   (b) Compare with target value (label)
   (c) Update the weights according to $\Delta w = \eta$ (target-out)x

Let us now consider two cases where prediction matches the *correct* label (left column), and the one where it does not (right column), assuming the correct label being either -1 (1st row) or $+1$ (2nd row).

If $t^{(i)} = o^{(i)}$

If the predicted output *matches* the desired label, no changes are made since prediction is **correct**:

$$\Delta w = \eta(-1 - (-1))x = 0$$

$$\Delta w = \eta(1 - 1)x = 0$$

If $t^{(i)} \neq o^{(i)}$

Otherwise, weights are pushed towards the direction of the positive or negative target class [2], since the prediction is **wrong**:

$$\Delta w = \eta(1 - (-1))x = \eta(+2)x$$

$$\Delta w = \eta(-1 - 1)x = \eta(-2)x$$

Namely, the **update rule** for *perceptron*, given a certain input x_i can be schematized as:

$$w_i \quad \leftarrow \quad w_i + \Delta w_i$$

[2] $\wedge$ NB: input x is *positive*

with:

$$\Delta w_i = \overbrace{\eta}^{\text{learning rate}} \; (\; \underbrace{t}_{\text{target value}} \; - \; \underbrace{o}_{\text{peceptron ouput}} \;) \overbrace{x_i}^{\text{input}}$$

It can be shown that if the data points are **linearly separable**, the perceptron learning rule will *always* converge to a *correct solution*, among many possible ones.
Geometrically, the perceptron tries to find the coefficients describing an hyperplane that correctly weights the input points, in such a way that perfectly separates. This is feasible, and so is a solution can be found, *only* if data is linearly separable: that is to say that there exists a $p-1$-dim hyperplane which divides the R^p space into two regions, which contain points of only one type. In this case, there must not be outliers.

However, there are some problems arising when using this algorithm.

- **Multiple correct solutions** may exist, i.e., we can obtain different hyperplanes which correctly classifies the points. To solve this issue, we would need more training samples or use SVMs methods which turn out to be more suitable.
- Data may **not be separable**. Therefore the perceptron can still be used, though never reaches a converge. A possible solution is to set a maximum number of epochs, and then take the set of parameters which has minimized the loss function (*pocket algorithm*).

Let us try now to solve another problem, namely the ($g(\cdot)$ *non-differentiability*) of the activation function. We can replace the Heaviside function, with a *linear activation function*, also known as "*Widrow-Hoff*" or "*Adaline*" Rule:

$$out = \sum_{i=1}^{n} w_i x_i + b_{th}$$

which is a continuous and *differentiable* function, such that:

- Allow us to use **gradient descent** to minimize the loss
- The error cannot take discrete values anymore, but rather a *continuous* value that encodes the *actual* distance between the prediction and the target.

So[3], using as metric the Mean Square Error) *MSE*, namely using as **loss**: $J(\boldsymbol{w}) = 1/2 \sum_{i=1}^{n} (t^{(i)} - o^{(i)})^2$, we can proceed to compute its derivative to understand how **backpropagation** actually works:

[3] ∧Omitting the bias term $o^{(i)}$ for simplicity

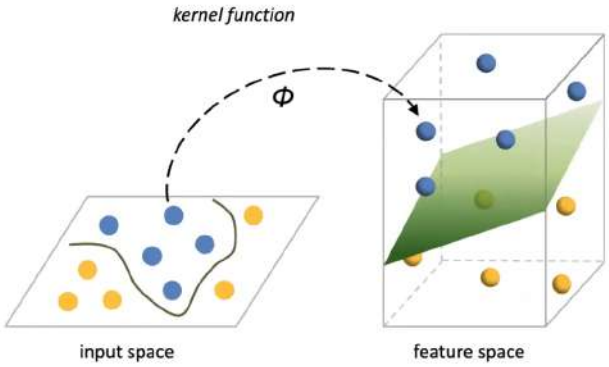


Figure (2.3) – In order to correctly learn non-linear models, there must be a Φ *kernel function* that projects data from input space to the feature space.

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \left(t^{(i)} - o^{(i)}\right)^2 \\
&= \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}\left(t^{(i)} - o^{(i)}\right)^2 \\
&= \frac{1}{2}\sum_i 2\left(t^{(i)} - o^{(i)}\right)\frac{\partial}{\partial w_j}\left(t^{(i)} - o^{(i)}\right) \\
&= \sum_i \left(t^{(i)} - o^{(i)}\right)\frac{\partial}{\partial w_j}\left(t^{(i)} - \sum_j w_j x_j^{(i)}\right) \\
&= \sum_i \left(t^{(i)} - o^{(i)}\right)\left(-x_j^{(i)}\right) \\
\rightarrow \Delta w_j = -\eta\frac{\partial J}{\partial w_j} &= -\eta\sum_i \left(t^{(i)} - o^{(i)}\right)\left(-x_j^{(i)}\right) = \eta\sum_i \left(t^{(i)} - o^{(i)}\right)x_j^{(i)}
\end{aligned}$$

Which is formally resembling the perceptron update rule, but nevertheless is really different. Indeed, the changes in weights are not as abrupt as in the Heaviside function case, but steepness changes little at every iteration. This method is called **Delta rule**.

However, even if using the Delta rule, the perceptron is not able to solve a **non-linear problem** like the **XOR** function, which requires a type of nonlinear classifier. It was proved that these types of problems could not be solved by perceptron.
A solution was found by **non-linearly** projecting the points in a *feature space*, via a **kernel function** (see Fig. 2.3). In Deep Learning, the architecture of neural networks essentially are constructed through the same principle. However, rather than using a kernel function, they "create" an internal representation of the input data via an hierarchical representation of it, by stacking layers.
Let us increase the *complexity* of our models, and introduce the **multi-layer** perceptrons. A simple example of the latter is the *Feed-Forward Neural Network*, which contains an additional layer of *hidden neurons* which extracts (and represents) some features of the data in a different space: the so called *feature* space.
One should note that all activation functions must be *differentiable*, since learning is based on *GD* (or *SGD*):

$$\nabla_\theta J(\boldsymbol{\theta})$$

and the weights are changed according to the **error back-propagation**, which is a computationally efficient procedure for computing the gradient exploiting the **chain rule**. Once we know *gradient*, weights are updated according to *GD* or *SGD*.

2.1.1 Activation functions

Most popular

As we know, *activation functions* are mathematical expressions that determine the **output** of a neural network. This function is attached to every neuron of the network and determines whether it will fire, based on the signals coming from either the input data or other previous layers. Activation function also comes handy in *normalizing* the output of the neurons, constraining it to be positive, negative, or in some range (for example $out \in [-1, +1]$) depending on the choice of such function.
An additional aspect to be taken into account about activation functions is that they must be *computationally efficient*. Indeed the outputs are calculated at every iteration across thousands or even millions of neurons and for each data sample. [4]
Input layer:

Identity $f(x) = x$

Hidden layer:

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

Advantages

Smooth gradient, preventing "jumps" in output values. Output values bound between 0 and 1, normalizing the output of each neuron. Clear prediction—For X above 2 or below -2, tends to bring the Y value (i.e., the prediction) to the edge of the curve, very close to 1 or 0. This enables clear prediction.

Disadvantages

Computationally expensive (with respect to **ReLU**)

Sigmoids saturate and kill gradients. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would

[4] ∧*For a visual representation of the following functions we suggest the following website*

become saturated and the network will barely learn, leading to a problem known as *vanishing gradient problem*. This is a reason why it is not commonly used for hidden neurons but only for the output ones when we deal with the binary classification problem.

Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network (more about this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming from a neuron is always positive (e.g., $x > 0$ elementwise in $f = w^T x + b$), then the gradient on the weights will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights.

However, notice that once these gradients are added up across a batch of data, the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

Tanh

$$\tanh(x) = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} = \frac{1 - e^{-2x}}{1 + e^{-2x}}.$$

– Like the *sigmoid* neuron, its activation saturates, but unlike the *sigmoid* neuron its output is **zero-centered**. Therefore, in practice the *tanh* nonlinearity is always preferred to the *sigmoid* non-linearity. Also note that the *tanh* neuron is simply a scaled *sigmoid* neuron, in particular, the following holds: $tanh(x) = 2\sigma(2x) - 1$.

Rectified Linear Unit (ReLU)

$$ReLU(x) = x^+ = \max(0, x)$$

Advantages

Computationally efficient — It was found to greatly accelerate (e.g. a factor of 6 in *Krizhevsky et al.*[5]) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its partly linear, non-saturating form. Only comparison, addition and multiplication. Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero. **Non-linear** — although it looks like a linear function. **Sparse activation**: For example, in a randomly initialized network, only about 50% of hidden units are activated (which have a non-zero output).

Better gradient propagation wrt sigmoid - fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions. Indeed, the gradient of $ReLU(x)$ is always large in half

[5] ∧http://www.cs.toronto.edu/ fritz/absps/imagenet.pdf

of the function domain (good for limiting gradient vanishing). For leaky ReLU, the gradient is actually never at zero.

In 2011,[6] the use of the rectifier as a nonlinearity has been shown to enable training deep supervised neural networks *without requiring unsupervised pre-training*. Rectified linear units, compared to sigmoid function or similar activation functions, allow faster and effective training of deep neural architectures on large and complex datasets.

Scale-Invariant $\max(0, \alpha x) = \alpha \max(0, x)$ for $\alpha \geq 0$

Disadvantages

Non-differentiable at zero - however, it is differentiable anywhere else, and the value of the derivative at zero can be arbitrarily chosen to be 0 or 1 (and this won't have a serious impact on the final result)[7];

The Dying ReLU problem - Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the *learning rate is set too high*. However, setting a *proper of the learning rate*, this issue happens more rarely. This is a form of the *vanishing gradient problem*. It may be mitigated by using leaky ReLUs instead, which assigns a small positive slope for $x < 0$, however the performance is reduced.

Leaky ReLU

$$f(x) = \mathbb{1}_{(x<0)}(0.01x) + \mathbb{1}_{(x\geq 0)}(x)$$

Advantages

Prevents dying ReLU problem - this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values.

Disadvantages

Results not consistent — leaky ReLU does not provide consistent predictions for negative input values.

Parametric ReLU

$$f(x) = \mathbb{1}_{(x<0)}(\alpha x) + \mathbb{1}_{(x\geq 0)}(x)$$

Leaky ReLUs' extensions are one attempt to fix the "dying ReLU" problem, letting the neuron to learn the parameters for the negative part of the input. Some

[6] ∧http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf

[7] ∧This aspect holds also for **LeakyReLU** but **not** for the **Heavyside Function**. In practice, learning in neural networks will never stop at a "perfect" local minimum of the loss function, so the minimum might correspond to a point where the gradient is not well defined and we will not have any issue.

people report success with this form of activation function, but the **results are not always consistent**. The learning of such parameter was introduced in Delving Deep into Rectifiers, by Kaiming He et al., 2015. However, the consistency of the benefit across tasks is presently unclear.

Summing up:

Advantages

Allows the negative slope to be learned — unlike leaky ReLU, this function provides the slope of the negative part of the function as an argument. It is, therefore, possible to perform backpropagation and learn the most appropriate value of α.

Disadvantages

May perform differently for different problems - Sensitive.

ELU

$$f(x) = \mathbb{1}_{(x<0)}(\alpha e^x - 1) + \mathbb{1}_{(x\geq 0)}(x)$$

where $\alpha \geq 0$ is an hyperparameter to be learned.

Exponential linear units try to make the mean activation closer to zero, which speeds up learning. *ELU*s can obtain higher classification accuracy than *ReLU*s.

Scaled Exponential Linear Units (SELU)[8]

$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha \exp(x) - \alpha & \text{if } x \leq 0 \end{cases}$$

It is a specific kind of AF which allows to do the **internal normalization** of NN. The main idea (see the original paper[9] for details), is that each layer preserves the mean and variance from the previous layer. The activation function needs both positive and negative values for y to shift the mean. Both options are given here. ReLU is not a candidate for a self-normalizing activation function since it can not output negative values. Although it looks like a ReLU for values larger than zero, there is an extra parameter involved: λ. This parameter is the reason for the S(caled) in SELU. When it is larger than one, the gradient is also larger than one, and the activation function can increase the variance. The implementations in Tensorflow and PyTorch use the value from the original paper, which is about 1.0507.

Swish

$$\text{swish}(x) := x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

Swish is a new, self-gated activation function discovered by researchers at Google. According to their paper[10], it performs better than ReLU with a similar level of computational efficiency. In experiments on ImageNet with identical models running ReLU and Swish, the new function achieved top -1 classification accuracy 0.6-0.9% higher

[8] ∧https://towardsdatascience.com/gentle-introduction-to-selus-b19943068cd9
[9] ∧https://arxiv.org/pdf/1706.02515.pdf
[10] ∧https://arxiv.org/pdf/1710.05941v1.pdf

Output layer:

Linear - regression: $f(x) = x$

Sigmoid - binary classification: $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$

Softmax (also known as normalized exponential function) - used for multi-class classification [11] We simply apply the exponential function to the activation value for each unit, and divide by the sum over all the resulting K values to get a proper pdf:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$

Finally, we choose the output class by **sampling** from this distribution. This introduces some *stochasticity* in the model: for a given input, output can even be different. In addition, stochasticity can be tuned by introducing a **temperature** factor.

It is used in *multinomial logistic regression* and is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over the predicted output classes. It is a generalization of the logistic function to multiple dimensions. As a consequence, it is used in *soft* classification tasks, i.e., a kind of classification in which the learning function associates an input to each class with a specific probability measure of belonging. Soft classification comes in handy when data is *noisy*, allowing us to perform classification in such context. e.g.

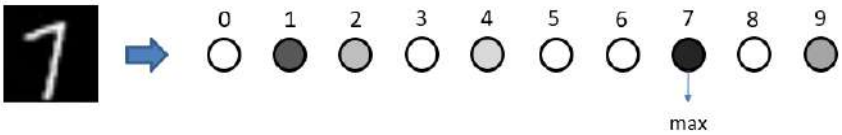


Figure (2.4) – In this example we can see how the input image can be easily associated to 7 or to 1

The final take home message is:

Q. *"What neuron type should I use?"*

A. Use the ReLU nonlinearity, be careful with your learning rate, and possibly monitor the fraction of "dead" units in the network. If this turns out to be an actual problem, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

[11] ∧it actually replaces the "hard max", returning a probability over a possible class for a given input and not a "hard" value.

Mathematical properties

Aside from their empirical performance, activation functions also have different mathematical properties:

> **Nonlinear** – When the activation function is nonlinear, then a two-layer neural network can be proven to be a universal function approximator. This is known as the Universal Approximation Theorem. The identity activation function does not satisfy this property. When multiple layers use the identity activation function, the entire network is equivalent to a single-layer model.
>
> **Range** – When the range of the activation function is finite, gradient-based training methods tend to be more stable, because pattern presentations significantly affect only limited weights. When the range is infinite, training is generally more efficient because pattern presentations significantly affect most of the weights. In the latter case, smaller learning rates are typically necessary.
>
> **Continuously differentiable** – This property is desirable (ReLU is not continuously differentiable and has some issues with gradient-based optimization, but it is still possible) for enabling gradient-based optimization methods. The binary step activation function is not differentiable at 0, and it differentiates to 0 for all other values, so gradient-based methods can make no progress with it.
>
> **Monotonic** – When the activation function is monotonic, the error surface associated with a single-layer model is guaranteed to be convex.
>
> **Smooth functions with a monotonic derivative** – These have been shown to generalize better in some cases.
>
> **Approximates identity near the origin** – When activation functions have this property, the neural network will learn efficiently when its weights are initialized with small random values. When the activation function does not approximate identity near the origin, special care must be used when initializing the weights.[10] In the table below, activation functions where $f(0) = 0$ and $f'(0) = 1$ and f' is continuous at 0 are indicated as having this property.

These properties do not decisively influence performance, nor are they the only mathematical properties that may be useful.

2.1.2 Loss Functions

Learning is generally implemented as a form of **Maximum-Likelihood** (ML) estimation procedure, where the loss function corresponds to the negative log-likelihood of the data:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y}|\boldsymbol{x})$$

In *supervised learning* the aim is to estimate a conditional probability distribution: note as $\vec{x}$ is given, as it is an input data. The GD (or SGD) algorithm aims at minimizing the dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$ (defined by the examples in the training set) and the model distribution (p_{model}), as measured by KL divergence:

$$D_{\text{KL}}\left(\hat{p}_{\text{data}} \| p_{\text{model}}\right) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}\left[\log \hat{p}_{\text{data}}(\boldsymbol{x}) - \log p_{\text{model}}(\boldsymbol{x})\right]$$

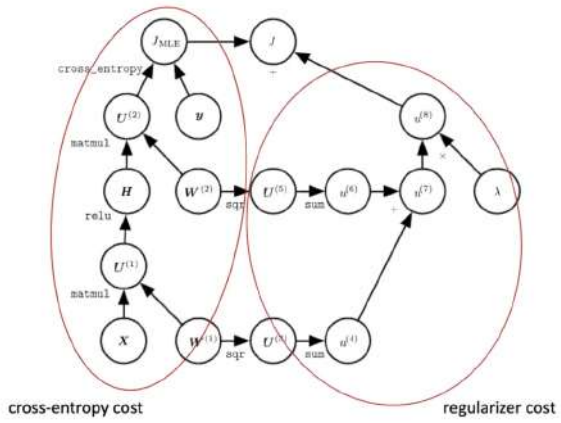


Figure (2.5) – Example of a computational graph of a single-layer and feed-forward network. One can recognize two parts of the graphs related to accuracy the accuracy and regularizer costs of the final functions, finally summing " + " them.

For *binary classification*, this corresponds to using the cross-entropy as the loss function. If the output layer has linear units, maximizing the log-likelihood (or minimizing the cross-entropy) is equivalent to minimizing the mean squared error:

$$\text{MSE}_{\text{train}} = \frac{1}{m}\sum_{i=1}^{m}\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2$$

Given the model with Gaussian error:

$$p(y \mid \boldsymbol{x}) = \mathcal{N}\left(y; \hat{y}(\boldsymbol{x}; \boldsymbol{w}), \sigma^2\right)$$

To minimize a loss function, we make use of the **error back-propagation**. It consists of, as a first step to compute the error, and later to understand *how much* every weight *contributes* to the error. In other words, the magnitude of the gradient with respect to each weight in the network tells us how sensitive is the loss function to changes in that weight. Visually, to better understand how this algorithm works, one should watch [12] [13].
Intuitively, it exploits the **chain rule** of calculus to compute the derivative of composite functions. This allows to recursively determine the contribution of each variable to the final loss. However, to simplify computations, the drawing of a computational graph is strongly suggested, as it is for a single-layer feed-forward network in Fig. 2.5. This is automatized and automatically done in the background by libraries such PyTorch/Tensorflow.

[12] ^https://www.youtube.com/watch?v=Ilg3gGewQ5U&ab_channel=3Blue1Brown

[13] ^https://www.youtube.com/watch?v=tIeHLnjs5U8&ab_channel=3Blue1Brown

2.1.3 Data Pre-Processing

Input normalization

To improve the convergence of gradient descent, the input signals must be *normalized*.
In this way ,all input features are on the same scale. This operation can be done when features are **independent** (or at least partially independent), i.e., when features do not share any kind of relationships among them. A counterexample when this must not be done, is in image processing: indeed the pixels scale, once normalized, may not reflect the same pattern of the original data thus losing information.
The most common ways to perform normalization are the following ones:

Features rescaling: Stretch the range of features in the range $[0, 1]$ or $[-1, 1]$ (depending on the data)

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Features standardization: Subtract the mean $\bar{x}$ and divide by standard deviation $\sigma \rightarrow$ will then have zero-mean and unit-variance

$$x' = \frac{x - \bar{x}}{\sigma}$$

being this one approach most widely used.

2.1.4 Universal Approximation Properties

It can be proved that feed-forward networks with at least one hidden layer composed by ***non-linear*** *units can approximate* ***any continuous function*** *from one finite-dimensional space to another with arbitrary precision, provided that the network is* ***given enough hidden units*** (Hornik et al., 1989; Cybenko, 1989)

This is an essential result, since it describes how much powerful these kinds of models are, but we should not forget about the limitations in the number of hidden units. This indeed overcomes the limitations we encountered when speaking about the perceptron, since now we are able to tackle also *non-linear* problems.
However, some considerations follow:

- Being able to represent a function in principle $\neq$ being able to learn the function
- Learning might fail due to poor optimization and/or overfitting
- Learning might require an *exponentially large number* of *units*
- Learning might require an *exponentially large* number of *iterations* (learning epochs)

Therefore, even if in the 90s it was proven that given a shallow neural network it was possible to *represent* any *Borel-measurable* function with an arbitrary approximation, the practical issues related to the training of such neural networks were many. In addition, at the same time, ANNs were put aside again due to the fact that computers were not sufficiently resourceful to train them.

2.1.5 Deep Learning Neural Networks

The last obstacle we introduced was partially solved via the introduction of another type of NNs: *Deep Neural Networks* **DNN**s (ANNs with more than one hidden layer). They differ from the more basic Neural networks from the fact that they are based on some **hierarchical organization of knowledge**, implemented by using multiple levels of representation. Indeed simple hypotheses (*basic features*) are gradually combined into more complex ones (*abstract concepts*), thus involving some sort function decomposing the input into "smaller" features and then combining them.
e.g. if we want to classify a face, we need a layer to classify the edges that are present in an image containing a specific kind of face, another one for the colors, some more complicated ones for detecting the peculiarities of the eyes, nose, ... Finally, an hidden layer that combine all this information establishes the topology of a face.

Having stacked of multiple layers leads to **knowledge sharing** across the layers, allowing the neurons to use the prior information provided by *previous neurons*. In such a way that some higher *hierarchy* in the features is introduced, instead of recreating new ones. Indeed when decreasing the number of layers, we also decrease the degree in which the NN's units are able share information with each other, in order to better solve a task as whole. Theoretically, it was found that *"functions that can be compactly represented by a depth k architecture might require an exponential number of computational elements to be represented by a depth $k-1$ architecture* [14] In other words, if we decreased the number of layers by 1, we would need to add an *exponentially* large number of units to encode the information we have lost. Indeed, one achieves the following **empirical result**: in practice, deep networks largely *outperform* shallow networks with few hidden layers. This hierarchy introduced was already present in some intuitions about how the *brain* **self-organizes** the information, but how to practically implement it in a *learning system* was a later discovery.
Nevertheless the stack of multiple layers may lead to a new *problems*.

Problems

In a network of n hidden layers, n derivatives will be multiplied together. If derivatives have *large* values, then the gradient will increase exponentially as we propagate down the model, and eventually explode. This is what we call the *problem* of **exploding gradient**.

[14] ∧ [Bengio 2009; Montufar et al. 2014]

Conversely, when derivatives are *small* then the gradient will decrease exponentially as we propagate it back through the model, until eventually vanishing. From this, its name of **vanishing gradient** problem. This may completely stop the learning process, since we are not able to correctly understand what weights, and in which measure, contribute to the loss function. Therefore, weights close to *input layers* update less often and with more difficulty, having the gradient already vanished. Mostly, this occurs with *sigmoid* and *tanh* activation functions, whose derivative is null far from 0. These problems has led to consider *Deep Learning* to be **unfeasible**, since these functions were used mainly because they were differentiable.

Nowadays, we can instead increase the depth of a network by stacking *several* **non-linear** layers on top of each other, in a **Deep Neural Network**. *Learning* always occurs via GD (or SGD) and back-propagation, as we have already stated, but some attention has to be provided to prevent both **Gradient vanishing** (learning signal gets weaker) and **overfitting** (many parameters increase the model complexity).

To tackle the first issue, it is often performed some **Unsupervised pre-training**: initially, the first layers of the network are fed with the input without providing any label, thus making the model to encode the input characteristics by itself. Once this has gradually been performed layer by layer, one proceeds with the normal supervised training. This is however, an *obsolete* solution, and now people make use of the **ReLU**s activation functions, since gradient is not vanishing. However, using *ReLU* might make the gradient *explode*, thus returning *NaN*s, or making the learning unstable. A solution for this new issue is to check and eventually limit the size of gradients during training via either **gradient clipping** (when the norm of the gradient exceeds a threshold, set it to a maximal value) or **gradient scaling** (normalize the norm of the gradient to a predefined value e.g. 1, but it is really *computationally demanding*). Alternatively, to solve the vanishing gradient problem, one could exploit some tricks for improving SGD, such as optimizers, etc. Finally, increasing the computational power (for data and hardware), we could solve partially the issue of vanishing gradient.

The **solution** to the *overfitting* problem is, as we already know, the introduction of the **regularization** term in the loss that is related to the model parameters.

To have a more precise description of these aspects of DNN, one should take a look at the following paragraphs.

Vanishing Gradients

During backpropagation, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value.

In the worst case, this may completely stop the neural network from further training.

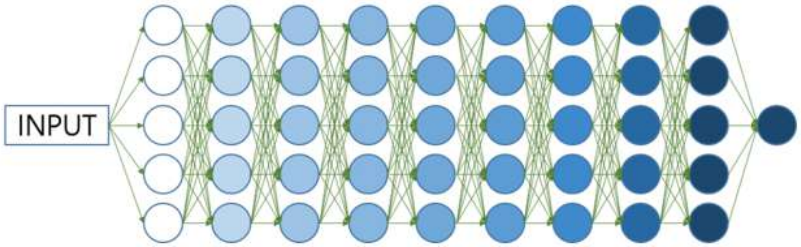


Figure (2.6) – Vanishing Gradient problem

As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range $(-1, 1)$, and backpropagation computes the gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers[15] in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly. In fact it *mostly happens with sigmoid or activation functions*, which could easily *saturate*.

But what we mean as saturating function?

The **intuition** is that a saturating activation function squashes a real-valued number to a finite range of values (e.g., $[-1, 1]$ for the $tanh(x)$). However, let us see the mathematical definition of a saturating function:

Given a continuous differentiable on traits function f , we define it as a non-saturated one iff

$$f\left(\left|\lim_{z\to-\infty} f(z)\right| = +\infty\right) \vee \left(\left|\lim_{z\to+\infty} f(z)\right| = +\infty\right)$$

As a consequence, f is saturating iff f is not non-saturating. These definitions are not specific to neural networks.
As you will see, *saturating* functions' derivatives assume *non-zero* in a small interval for this type of functions.

How to spot this problem on practice?

- The model will improve very slowly during the training phase and it is also possible that the training stops very early, meaning that any further training does not improve the model.
- The weights closer to the output layer of the model would witness more of a change, whereas the layers that occur closer to the input layer would not change much (if at all).
- Model weights shrink exponentially and become very small when training the model.

[15] ∧The first hidden layers in the network - near the input one

- The model weights become 0 in the training phase.

How do we solve this problem?

- Accurately *select non-saturating activation functions* (e.g. **ReLU**), Being careful of dying neurons and exploding gradients).
- *Unsupervised pre-training*, a process that consists of reconstructing the input from the "front" layers before learning (Generative Models);
- *Increasing computational capabilities* (GPU)
- *Improving SGD* according to the kind of problem (momentum, ADAM, ...)
- Adopting new *inizialization schemes* (*Xavier*,...)

When activation functions are used whose derivatives can take on larger values (e.g. with **ReLU**s), one risks encountering the related *exploding gradient problem*.

Exploding Gradients

In the case of exploding gradients, the accumulation of large derivatives results in the model being very unstable and incapable of effective learning, the large change in the model weights creates a very unstable network, which at extreme values the weights become so large that is causing overflow resulting in NaN weight values of which can no longer be updated. When faced with these problems, to confirm whether the problem is due to exploding gradients, there are some much more transparent signs, for instance:

- Model weights grow exponentially and become very large when training the model
- The model weights become `NaN` in the training phase.
- The derivatives are constantly increasing over time

Solutions?

- **Gradient clipping**: if the norm of the gradient exceeds a given threshold, set it to the maximum value
- **Gradient scaling**: normalizing the gradient vector such that vector norm equals a defined value (e.g. $||w_i|| = 1$)

Overfitting

There are several ways of controlling the capacity of Neural Networks to prevent overfitting:

Ridge

L^2 **regularization** is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. That is, for every weight w in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. It is common to see the factor of $\frac{1}{2}$ in front because then the gradient of this term with respect to the parameter w is simply λw instead of $2\lambda w$. The L^2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring *diffuse weight vectors*. Also, due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot. Lastly, notice that during gradient descent parameter update, using the L^2 regularization ultimately means that every weight is decayed linearly: `W+=-lambda * W` towards zero.

Lasso

L^1 regularization is another relatively common form of regularization, where for each weight w we add the term $\lambda|w|$ to the objective. It is possible to combine the L^1 regularization with the L^2 regularization: $\lambda_1|w| + \lambda_2 w^2$ (this is called *Elastic net regularization*). The L^1 regularization has the intriguing property that it *leads the weight vectors to become sparse during optimization* (l.e. very close to exact zeros). In other words, neurons with L^1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the "noisy" inputs. In comparison, final weight vectors from L^2 regularization are usually diffuse, small numbers. In practice, if you are not concerned with explicit feature selection, L^2 regularization can be expected to give superior performance over L^1.

Max norm constraints Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector $\vec{w}$ of every neuron to satisfy $\|\vec{w}\|_2 < c$. Typical values of c are on orders of 3 or 4. Some people report improvements when using this form of regularization. One of its appealing properties is that the network cannot "explode" even when the learning rates are set too high because the updates are always bounded.

Dropout Simple and recently introduced regularization technique by Srivastava et al. In *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* that complements the other methods (L^1, L^2, *maxnorm*). While training, dropout is implemented by only keeping a neuron active with some probability p (a *hyperparameter*), or setting it to zero otherwise.

2.2 Advanced Optimization

(Lesson 6 of 15/10/20)
Compiled: September 20, 2021

Let us now try to understand how it is possible to **optimize** the *back-propagation algorithm*, in such way that we can as an example avoid vanishing gradient problem or any other issue related to Deep Learning architectures we previously encountered. We want, in other words, to understand what **tricks** to use to improve GD performance.

Weights Initialization

According to this **optimization technique**, we initialize the values of *synaptic weights* randomly and draw them either from a Normal distribution $\mathcal{N}(0,1)$ or a uniform $\mathcal{U}(-1,1)$.
This has to be done to *break the symmetry*: in the case two hidden units were connected to the same input, their initial parameters must be different or else, if learning is deterministic, they would converge to the *same* value.

However, increasing the weights even more might lead to a *saturation* of the *activation function*, thus slowing the learning, and specially in the case of *sigmoid* or *ReLU*. In order to avoid this issue, the following **heuristic** was introduced [16]:

$$W_{i,j} \sim \mathcal{U}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

where m and n are, respectively, the *fan-in* (number of inputs) and *fan-out* (number of outputs) of every layer. However, having many units connected to a single neuron might lead to a very small weight. Therefore, another possible initialization was introduced by Martens: **sparse initialization**. Each unit has exactly k non-zero weights, defined a priori, and are drawn according to any distribution. This allows to control the total amount of activation as we increase the number of units, but without making weights to small as m, n increase. Finally, some more schemes have been thought specifically depending on the activation function, but this is still an **open research** problem.

Momentum

Another optimization is to add a **momentum term** during the computation of SGD to make it converge *faster*. In this way, in analogy with *Newtonian dynamics*, also a velocity term $\vec{v}$ is introduced. It both contains the information of the *direction* and *speed* at which parameters are moving in the *optimization* space (see Fig. 2.7).

[16] ∧Glorot and Bengio (2010)

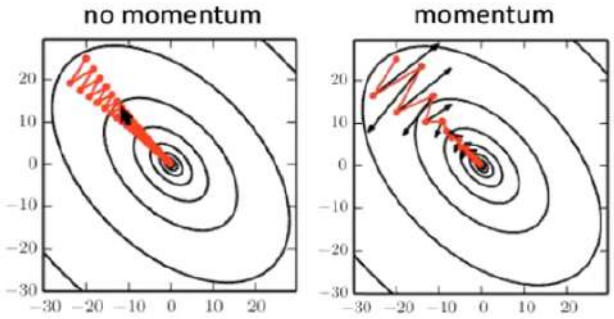


Figure (2.7) – Adding a momentum term helps accelerating gradient vectors in the right direction, thus accelerating SGD convergence. Indeed steps taken are less,.

Formally, the *velocity* is measured using an *exponentially decaying moving average* of *past* gradients, and adding it to the *current* gradient estimate $\boldsymbol{\theta}$:

$$v \leftarrow \overbrace{\alpha v}^{\text{cumulated gradient}} - \overbrace{\varepsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})}^{\text{current gradient}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + v$$

and usually $\alpha \in [0.5, 0.99]$, but is an hyperparameter to be tweaked.
This is based on the idea that if *successive* gradients are parallel, we are moving down a long valley, and so we are allowed to take bigger steps. Therefore momentum and *learning* rate are intimately connected with each other [17].

Adaptive learning rate

The **learning rate** can not even be kept fixed, but might adaptively *change* to improve performance: the larger it is, the faster the convergence should occur. However, when too large, it will lead to instability!
The *learning rate* can be set according to some **schemas**:

- *Constant* and usually *small* (~ 0.01), not particularly good (even if we add momentum).
- *Logarithmic decrease* as function of *epochs*, which is slightly better but still heuristic
- **Adaptive**, such as AdaGrad, AdaDelta, RMSProp, Adam

Let us focus on the third point, which is the *best* technique. The idea behind: is to use the *gradient* to **adapt** the **learning rate**. Hence we introduce **different learning rates** for *each* connection *weight*, according to their importance in the network.
The first techniques introduced were *AdaDelta* and **RMSProp**, and they essentially scale the learning rate of a *certain* weight by a running average ($1^s t$ moment of gradient distribution) of the magnitudes of recent gradients for *that* weight.

[17] ∧ have a play with https://distill.pub/2017/momentum/

Comparing the convergence velocity performances, the SGD is way slower. Hence, a type of optimization is strongly suggested to be used, taking care of choosing the appropriate set of parameters.

The most widely using *adaptive* learning rate techique is **Adam**. It computes *individual* adaptive learning rates for different parameters, using the estimates of first **and second** moments of the gradients (respectively $\mathbb{E}[g_t]$ and $\mathbb{E}[g_t^2]$):

$$\mathbb{E}[m_t] = \mathbb{E}[g_t] \qquad \mathbb{E}[v_t] = \mathbb{E}[g_t^2]$$

The update rule is therefore the one in Fig. 2.8.

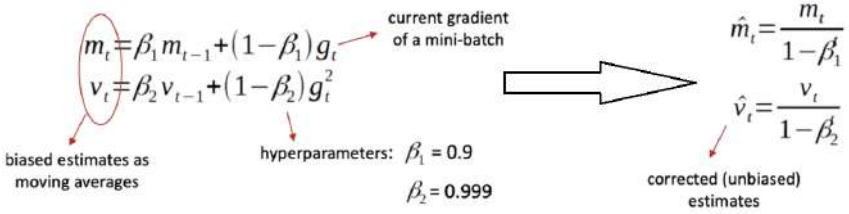


Figure (2.8) – Adam update rule. Theoretically, the quantities in the most lhs, are initially biased. Therefore needs to be unbiased, as one can see from the most rhs.

Finally, the **weight update** is given by:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where the hyperparameter $\eta \sim 0.001$ takes the role of a *learning rate*, and usually $\epsilon \sim 10^{-8}$, introduced to improve *numerical stability* for the method. Adam is usually *faster* rather than other methods, but there are some cases when it turns out to be slower.

The moral is: "start with Adam, but *always* try *several optimizers* for your model!".

Model regularization - Weight decay and sparsity

Other possible optimizations for regularizing the model, thus avoiding it to be too much *complex*, is to impose a **norm penalty** term $\alpha\Omega(\boldsymbol{\theta})$ for the parameters in the loss function:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \overbrace{J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})}^{\text{accuracy term}} + \overbrace{\alpha\Omega(\boldsymbol{\theta})}^{\text{regularization term}}$$

Here, the hyperparameter α defines the relative contribution of the **penalty** term to the *overall* loss function. Usually, it takes into account only the *weights*, which are regularized, and not the biases. **Different norms** can be used, producing different effects, such as:

$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|w\|_2^2$ L^2 norm (weight decay) *Ridge Regression*

$\Omega(\boldsymbol{\theta}) = \|w\|_1 = \sum_i |w_i|$ L^1 norm (coefficient sparsity) *Lasso Regression*

Model regularization - Dropout

Another possible **regularization** is related to the topology itself of the network: starting from a *fully connected bipartite* topology, one may want to randomly remove (*dropout*) **input** and **hidden** units during the processing of each pattern, as seen in Fig. 2.9.

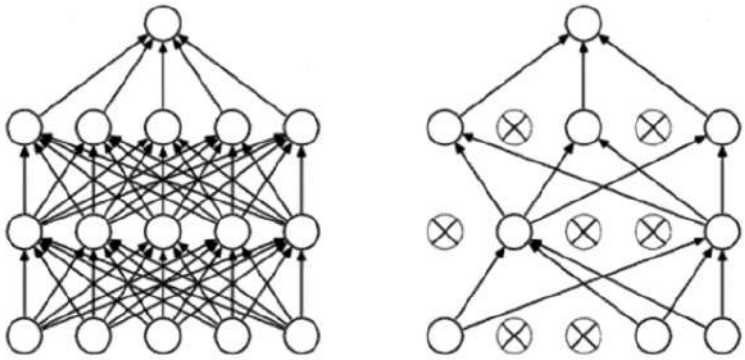


Figure (2.9) – Dropout example in a neural network: some neurons are randomly removed to decrease the complexity, thus avoiding overfitting.

It is a **very efficient** implementation: one simply needs to multiply the output activation by zero, thus neglecting the input coming from that neuron. Practically, one *draws* a *binary* mask $\{0, 1\}$ according to some probability p (usually $p \in [0.1; 0.6]$), which is related to the fraction of nodes we want to remove. If p is too large the model will become too simple, thus leading to *underfitting*.
The **rationale** behind this method is to make *multiple* units to learn the same feature, and eventually share/use it also in many other contexts. This is done by **regularizing each** hidden unit.
However, one should take care that this technique might interfere with other *regularizations* methods, such as **batch normalization** or L^2 norms. Hence, the parameters have to be tuned paying attention.

Hyperparameters tuning

Let us recap the hyperparameters we have seen so far, and one wants to tune: learning rate, momentum coefficient, size of moving average windows for adaptive learning rates, regularization factor, dropout probability, number of hidden layers, number of hidden units. It is quite a large set... Note as they do affect the model, but they are not the parameters (e.g. the *weights*) one wants to optimize.
To proceed to find the **optimal** hypterparameters, we will tackle two different approaches (see Fig. 2.10):

- **Grid search** is mainly used when the set of hyperparameters is limited, indeed we regularly (but not *exhaustively*) explore the hyperparameter space. The model is trained for every possible combinations of values the hyperparameters can take. This approach comes soon *unfeasible*.

- **Random search** actually performs better, and is less computationally demanding, when we increase the number of hyperparameters. The number of trainings we want to make is fixed *in advance* (~ 100), and the values of hyperparameters are drawn randomly. This is particularly *efficient* when some parameters are *more important* rather than the others, since we might not systematically miss it as it would be for the grid search.

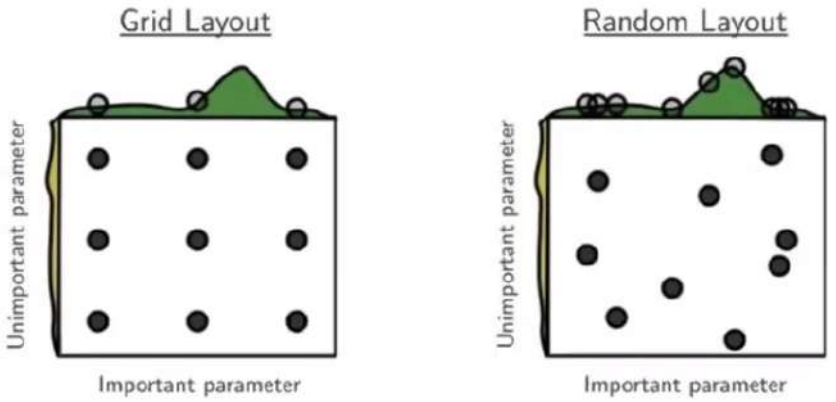


Figure (2.10) – Grid Search. Given two set of hyperparameters a, b, the Grid Search performs the Cartesian product $a \times b$ for every possible pair, thus making it unfeasible when the number of parameters and the possible values a single parameter can take scale up, since the number of explorations step diverges exponentially.
Random search: having fixed how many trainings one wants to make, the actual values of hyperparameters are sampled randomly. The advantage is that one can figure out what parameters are the most important for the model.

Second-order optimization methods

It is an other class of **optimization methods**, very computationally demanding though powerful. They are based on the fact that **second derivatives** actually give some information about the *curvature* of the *cost* function wrt parameters change. Indeed, if the curve exhibits *negative* curvature, it will decrease *faster* than its first derivative, allowing for *larger* steps in the SGD thus speeding up convergence. Conversely, if the curve exhibits *positive curvature*, too large steps will lead to an increase of the loss function, since the curve decreases slower than expected, eventually beginning to increase. As a last possibility: if the second derivative is null, the gradient predicts the decrease correctly. These arguments are visually represented in Fig. 2.11.

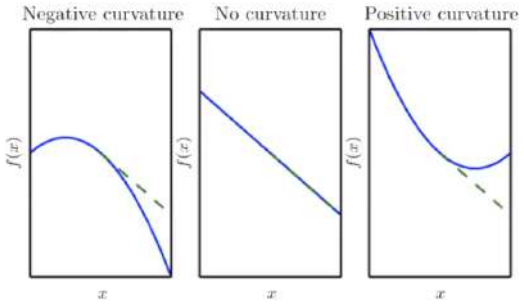


Figure (2.11) – Second order optimization methods are based on the behavior of the second derivative, that can tell how well the gradient approximates the loss function, so whether it underestimates or overestimates it.

To obtain the second derivative of the loss function, we must make use of linear algebra knowledge and compute the **Hessian**. It is basically the *Jacobian* of the *gradient*[18]:

$$J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$$

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x)$$

where $J_{i,j}$ and $\boldsymbol{H}(f)(\boldsymbol{x})_{i,j}$ are the elements, respectively, of the Jacobian and the Hessian matrix.
We recall now the definition of the **condition number** of any matrix, which is the *ratio* between its *largest* and its *smallest* eigenvalue. It measures *how much* the second derivates in a certain point are different from each other. When the Hessian has a **very large** condition number, the gradient descent performs poorly: in one direction the derivatives increases rapidly, while in another direction it increases slowly.
Some methods that exploit this information and guide SGD are:

- **Newton's method**: if the loss can be locally approximated using a **quadratic** function[19] we can **directly jump** into the minimum by *rescaling* the gradient by $\mathbf{H}^{-1}$:

  $$\theta^* = \theta_0 - \boldsymbol{H}^{-1} \nabla_\theta J(\theta_0)$$

  Some problems that arise are, as an example, that the *quadratic function* is quite strong: in DL it is never the case, since the loss function surface exhibits *many* saddle-points. Moreover, the *size* of $\boldsymbol{H}$ scales quadratically wrt the number of parameters in the model, and this will lead to memory-related issues when the networks are large.

- **Conjugate gradients/BFGS**: it is similar to Newton's method, but implements a more efficient way to compute $\boldsymbol{H}^{-1}$, approximating it via low-rank matrix which is *iteratively* refined.

[18] ∧ which is essential the matrix of first-order partial derivatives
[19] ∧ whose Hessian is positive definite

- **Hessian-free optimizers**: relieve from the burden of computing H^{-1}, and resort to finite-difference methods to approximate the product $H^{-1}\nabla_\theta J(\theta_0)$

(Lesson 7 of 20/10/20)
Compiled: September 20, 2021

2.3 Convolutional Networks

Let us now try to fix some **constraints** in the *input* data, that has a consequence to constrain the type of the network architecture. For example, by saying that our input data is an *image*, we expect a network that is able to accept and process something that is *bi-dimensional*. In addition when constraining the image resolution, and so the number of pixels, one is also fixing the number of units in the input layer. The most general structure of a network that is able to process image, is the one depicted in Fig. 2.12, and is known as **Convolutional Neural Network**.

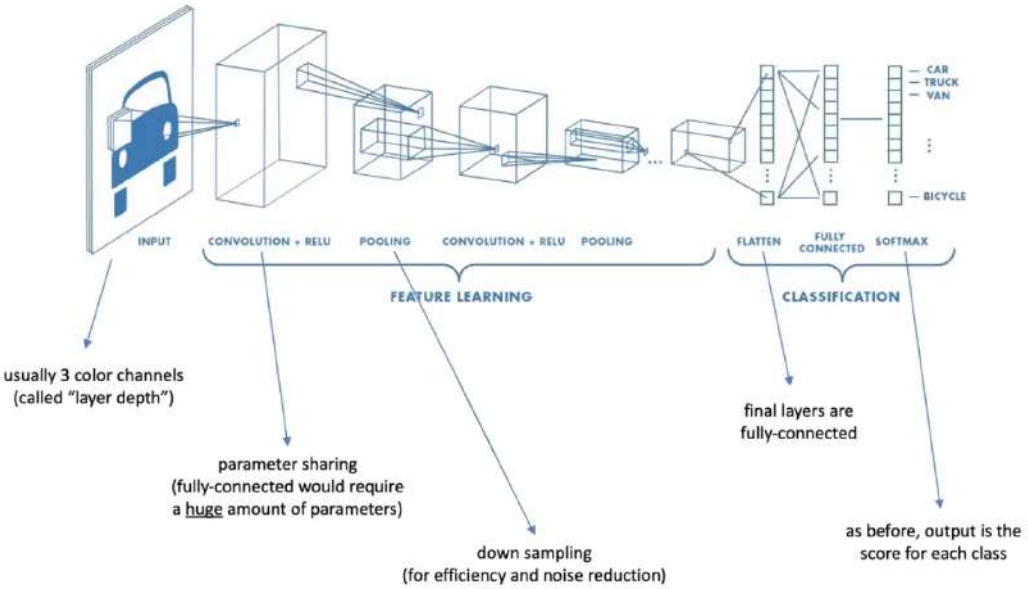


Figure (2.12) – Most general structure of a *Convolutional Network*, with the role of every layers block.

One should consider also that the input might have different layers, too: an image indeed has 3 different layers corresponding to three natural colors (R, G, B). Hence, every layer of the input *must* be in **one-to-one correspondence** with the number of filters of the initial **convolutional layer** in the network. The latter usually exhibits *parameter sharing*, in order to compress information thus keeping the number of parameters small. Later, **pooling layer**, performs some down sampling thus reducing the size of image [20]. Finally, the input is *transformed* into a vectorial form and fed to some **fully-connected layers**, resembling a *multi-layer perceptron*, which returns as output the score for every class. In convolutional networks, as one can notice from Fig. 2.13,

[20] ∧ now this approach is no more often used, and increasing the *stride* has taken the lead achieving the same result.

we make use both of **local connectivity** and **hierarchy**, the latter leading to a reuse of information.

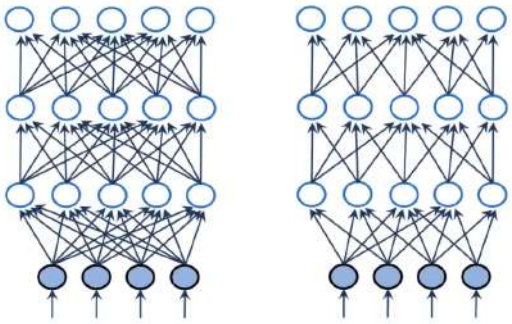


Figure (2.13) – Biologically, neurons in the *visual cortex* are **not fully-connected** to their "input-layer": it is expected for *close* pixels to be strongly correlated, whereas being uncorrelated with distant ones. One may want also to exploit this **local connectivity** when processing 2-dim images.

Let us take a closer look on how **convolution** works by inspecting what happens in a *hidden layer*.
Every hidden neuron has a **local receptive field**, that encodes a specific feature (*kernel* or *filter*). The **number** of hidden neurons (everyone having its own weight) defines how many features (*kernels*) will be represented at every processing layer. However, each kernel is **convolved** with the **entire input image**: with the same neurons (weights) we process *different* portions of the input (*parameter sharing*). Thus, the image is no more compressed into a 1-dim vector, but into a 2-dim filtered image.
The parameters of a convolutional layer are:

- **Number of "hidden neurons"**: defines how many kernels are used at each layer
- **Kernel size**: defines the receptive field of the kernel
- **Stride**: size of the step by which the convolution kernel moves (overlap)
- **Padding**: layers of (usually zero-valued) inputs surrounding the image, thus preserving the image size in the feature map.

Generally, to avoid too large feature maps, one may want to increase the stride, thus *skipping* some pixels. Moreover, at early stages of processing it is better to use *little* stride, in order to preserve image dimensions.

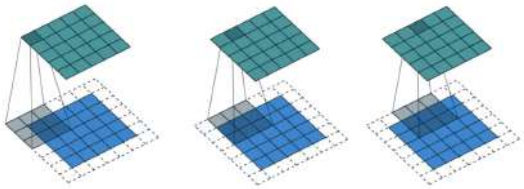


Figure (2.14) – Three different, and consecutive steps, of the convolution with stride$= 1$, kernel size$= 3 \times 3$

Computationally, one obtains a scalar value which is the result of the convolution, as depicted in Fig. 2.15. This value describes how well the portion of the image being considered actually matches a certain filter. It is often done in *edges detection*: the matching indeed is maximum when a portion of the input image exhibits the edge with a given orientation. The number of feature maps to be filled is equivalent to the total number (and types) of kernels we have (vertical edge, horizontal edge, 45° edge etc...).

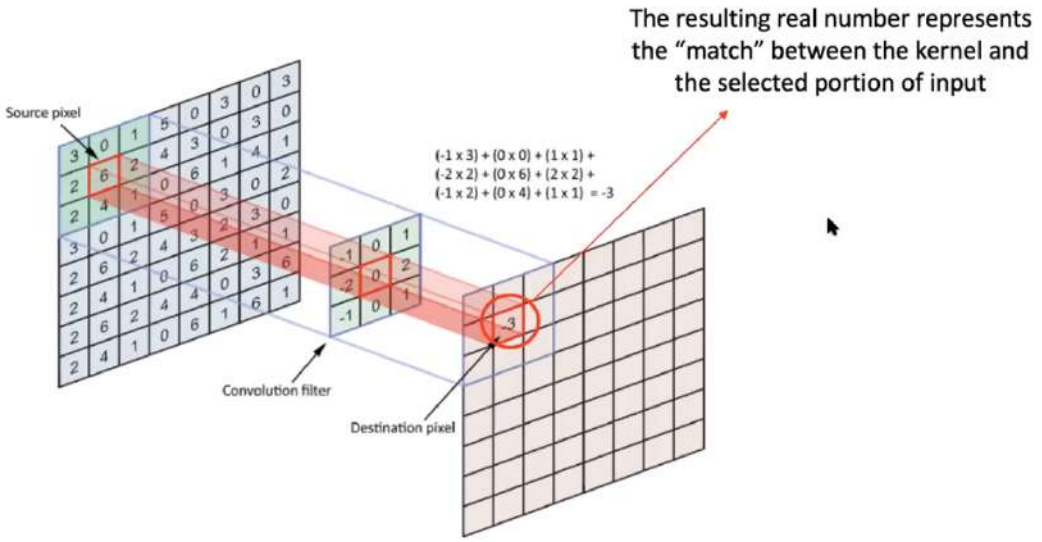


Figure (2.15) – How convolution computationally works at a single step and for *one* channel. This has to be repeated once the *feature map* has been filled out and for all the feature maps.

However, in the **most general case** the input image has **multiple** channels (i.e., usually the color *RGB*). In that case, every filter needs to have a **depth** that is equal to the number of channels it has. Every channel can have a *different* set of weights, but finally the result of the convolution is **combined** in order to obtain a *unique* number, thus a *unique* feature map. If one wants to go deeper into details, shall take a look at https://cs231n.github.io/convolutional-networks/. For a "static" example, see Fig. 2.15.
The *same* sets of weights are commonly shared with every portion of the image: this turns out to be efficient, therefore *convolutional layers* (i.e. the filters) are not that many.

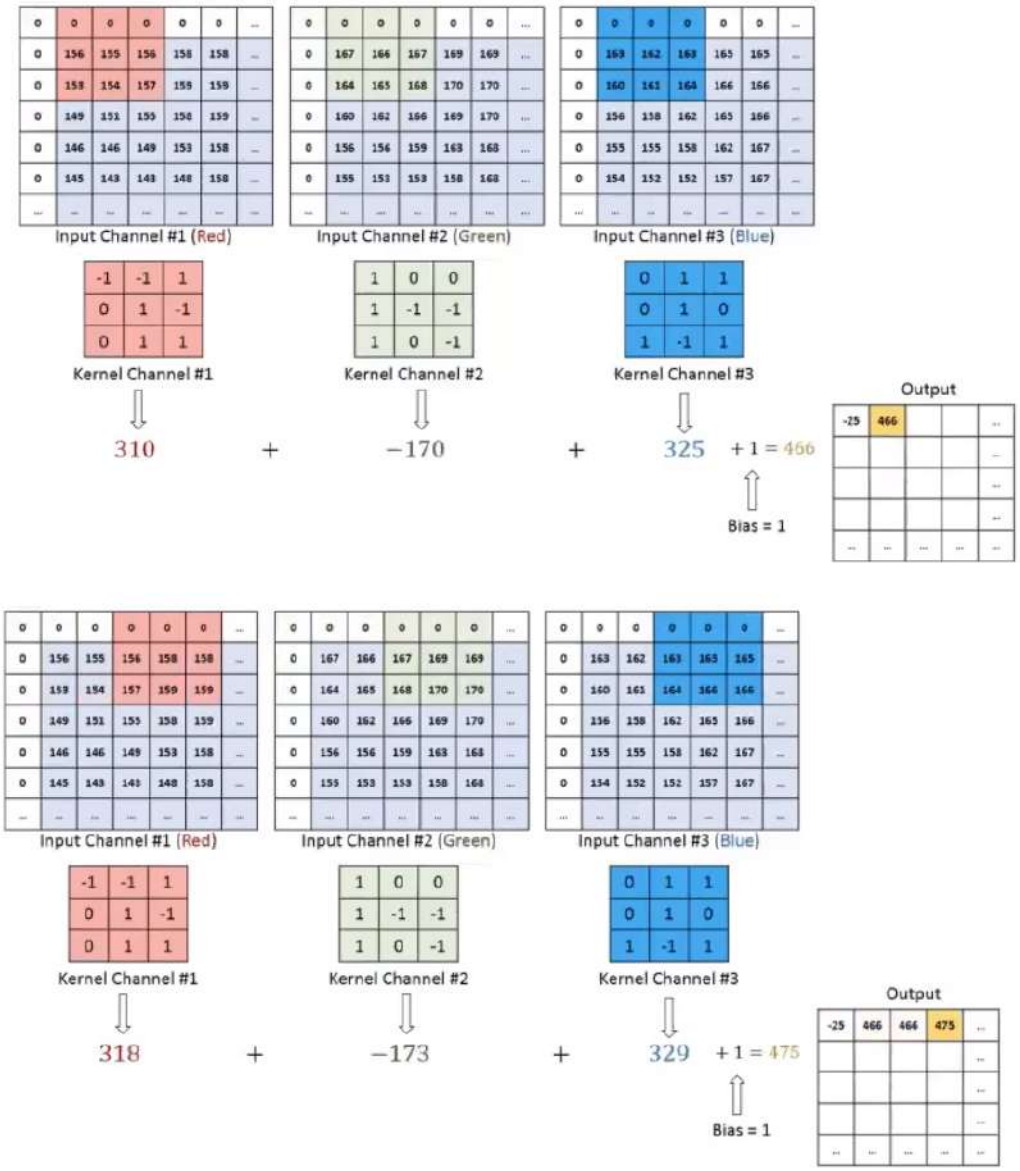


Figure (2.16) – How convolution computationally works for an image with $n = 3$ channels, related to natural colors **R**ed, **G**reen, and **B**lue and for two different time steps. Once the filters have been applied, the result is in turn summed which is the simplest way of combining results one can think of.

Some time ago **pooling layers** were used to *gradually* **reduce the dimensionality** of a filtered image, in such way that it reduced the number of parameters of filter themselves, controlled overfitting [21] and built invariances wrt translations. In other words it is like performing the **downsampling** of the image according to some rules, which might be taking maximum/minimum/average value or any other transformation (see Fig. 2.17). However, nowadays pooling is usually avoided in favor of the **stride**.

[21] ∧ indeed to detect a face rather than a car, the feature map does not need to be such detailed

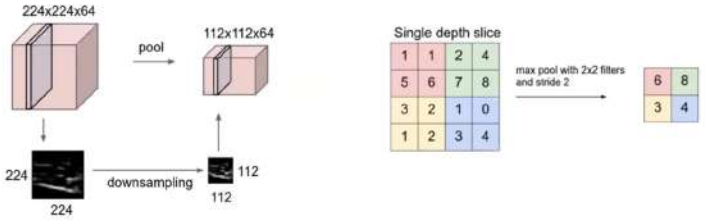


Figure (2.17) – Pooling layers were often used to reduce the dimensionality of an image, here it is implemented the *max* pooling, but some other forms exist.

Hall of Fame

Let us provide some examples of very famous and well-known architectures that made history of *Deep Learning*.

LeNet-5

It was first developed by LeCun et al. in 1998. The idea behind was to stack 4 layers, 3 convolutional while 1 fully connected. It was trained using MNIST dataset (32×32 pixels) to *digits recognition*, and was later on commercially used for automatic reading ZIP codes.

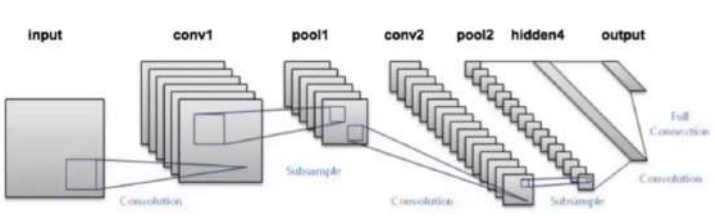


Figure (2.18) – LeNet-5 Architecture

AlexNet

It was first developed by Krizhevskij et al. in 2012. It consisted of a total of 8 layers, 5 convolutionals, and 3 fully connected. Moreover, many tricks (batch normalization, momentum, etc...) to improve SGD were implemented. At that time, it was the *state-of-the-art* object recognition accuracy for fully-sized real images, and was trained using a *huge* amount of data [22] for several days and exploiting GPU computations. The error percentage on ImageNet database was still large ($\sim$ 16.4%), but starting to be comparable to human level one, which is ($\sim$ 5%).

[22] $\wedge$ `http://image-net.org/explore`, with human labelling, organized hierarchically in different layers

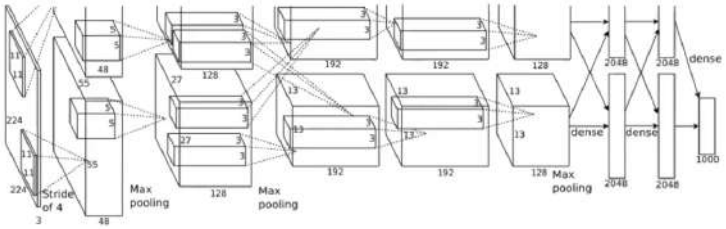


Figure (2.19) – AlexNet Architecture

GoogLeNet

It was sponsored by Google and published by Szegedy et al., 2015. It has an **inception** architecture with 22 layers. One should note the **auxiliary classifier**, that is a tool introduced to limit gradient vanishing effect: during training some *loss* quantity can be *injected* in the middle of the network, for example, considering some intermediate classification, in such a way that also the initial part of the network can be trained. This was one of the first architectures that needed considerable computational power, thus making their training unfeasible unless one possesses a supercomputer.

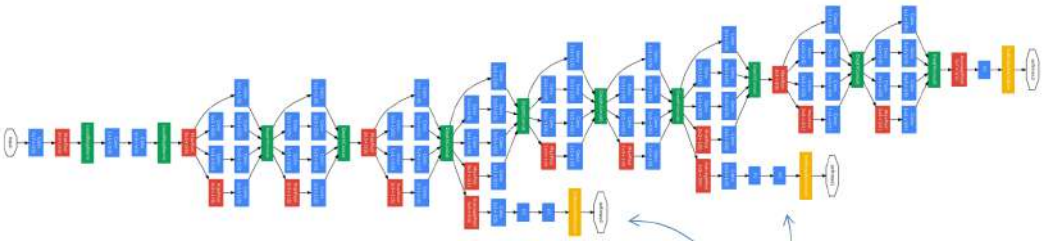


Figure (2.20) – GoogLeNet Architecture. The arrows highlight the auxiliary classifiers introduced to limit the effect of vanishing gradient.

An **inception** module is based on the intuition is that we might need different kernel sizes *in the same layer*, depending on the specific instance of the image that is being processed.

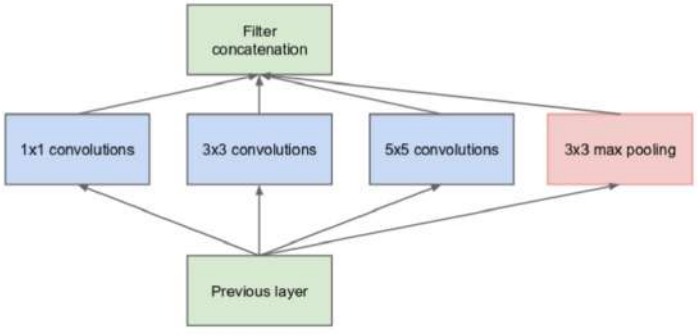


Figure (2.21) – Inception module, naive version.

ResNet

It was published and developed by He et al. in 2015. It was one of the **deepest architecture**, consisting of 152 layers. However, adding too many layers might be problematic due to *gradient vanishing*. A possible **solution** is to **skip connections** (aka shortcut connections) (see Fig. 2.22). In this way, the architecture of the network becomes much more flexible: during training, layers that are not useful for reducing the loss are skipped. The error is therefore directly propagated to other layers.

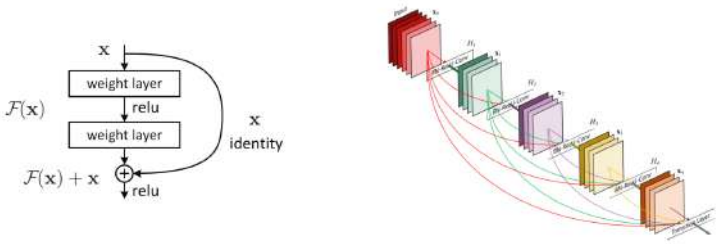


Figure (2.22) – How skip connections work: if a layer does not contribute to the loss it is skipped. This helps stacking many layers one after another, contributing to build a **densely connected** CNNs.

Transfer learning

The **rationale** behind transfer learning is that *shared representation* across tasks can better support inference and generalization, obviously when tasks are somehow related. As an example, if we are able to detect *strokes*, we can transfer this knowledge in language recognition, thus applying it to Chinese, Italian, Arabic, etc.
Hierarchical models, such as Deep Learning ones, make easier to share representation: rather than learning each task from scratch, we simply "fine-tune" the high-level features characterizing each task. The resulting architecture is the one similar to the one in Fig. 2.23. Practically, one can implement a **machine vision** system, and then **recycle** the visual knowledge *already learned* by some state-of-the art deep networks, such as the ones we have just encountered. In such way that we train *only* the last, fully connected layers.

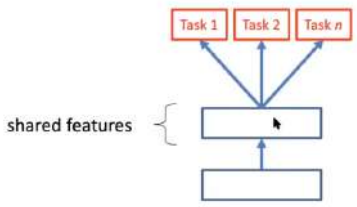


Figure (2.23) – Transfer learning most basic example: given the input in the bottom, the intermediate layer allows to abstract some shared features, which in turn are fed to more specific tasks.

How much to train what and how many layers to stack after the *already pre-trained* model depends on our needs, as one can see from Fig. 2.24.

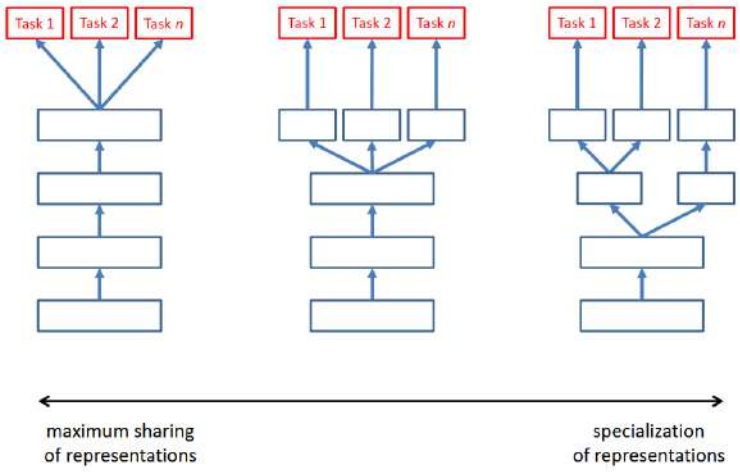


Figure (2.24) – Transfer learning applied to other more complex architectures, with the trade-off between accuracy (right) and generalization (left).

2.3.1 Interpretability

Let us focus now on the **interpretability** of *Deep Networks*, mostly focusing on *Convolutional* DN, that is to say to understand tasks are carried out. This will also point out *eventual vulnerabilities* either of the neural network itself, or the processing, therefore it must be analyzed wisely.

Adversarial samples

The following topic is related to an interesting phenomenon, discovered not much time ago [23]. It was shown indeed that we can **strikingly fool** a DN by *injecting* a **small percentage of noise**. As an example, one can look at Fig. 2.25 where, by adding some noise not distinguishable to human sight, the deep network ends up misclassifying the image.

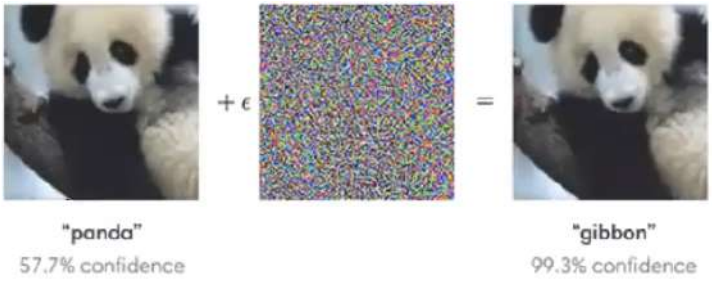


Figure (2.25) – Adding a *particular* noise to the image, appropriately chosen, we can **strinkingly** fool a deep network.

However, note that noise must be **created carefully**: the image must be modified in the *direction* of the gradient that **maximizes** the loss function with respect to the input image:

$$x^{adv} = x + \epsilon \cdot (\nabla J(x, y_{true}))$$

[23] https://blog.openai.com/adversarial-example-research/

This is *exactly* the opposite as we were doing in GD, but this time we are **changing the input** and not the weights. The *new* input, which have been perturbed, are known as **adversarial samples**.
This **vulnerability** is challenging the use of DL in sensitive contexts, such as *self-driving cars*. Indeed they can be easily fooled by placing stickers on road signs. An other example, is in *access control systems*: by adding strange noise, it might lead to a misrecognition of a person's identity.
Let us understand *how* and *why* this could happen. What the model does is to try to set some boundaries for different class outputs (as in Fig. 2.26) in a really high-dimensional space. When we produce adversarial samples, we are moving towards these boundaries, eventually trespassing them in such a way that a possible misclassification can be corrected during training.

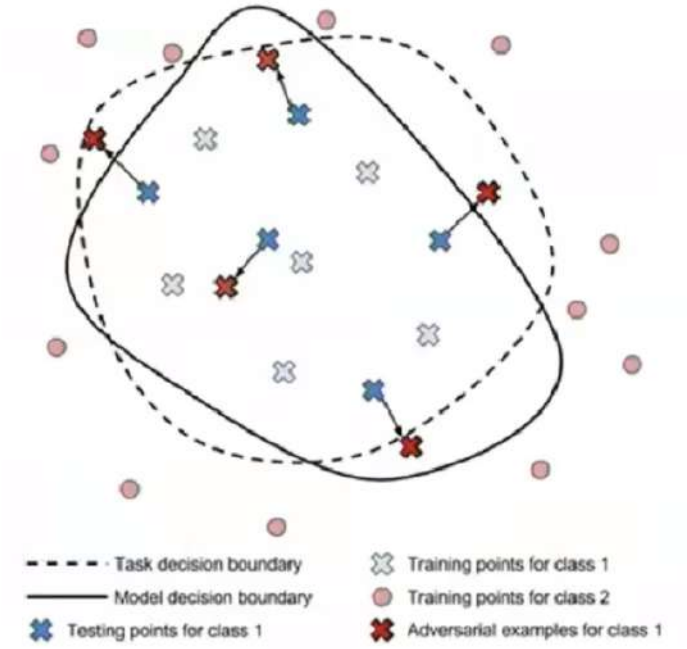


Figure (2.26) – Producing adversarial samples means to slightly move towards the boundaries, eventually moving outside them, in such way that this would lead to a misclassification.

Since this topic has been recently discovered, the ways we can **defend** from *adversarial attacks* are still *open research*. However some possibilities are:

- **Adversarial training**: the neural network is optimized via a *min-man* objective to achieve high accuracy on adversarially-perturbed training examples. In other words, we improve the robustness of our model by systematically attacking it using *adversarial samples*.
- **Randomized smoothing**: the neural network weights (and eventually the input) are smoothed by convolution with Gaussian noise:

  $$\hat{f}_\sigma(x) = \mathbb{E}_{\epsilon \sim \mathcal{N}(0,\sigma^2\mathcal{I})}[f(\boldsymbol{x} + \epsilon)]$$

  where f is the "original" network, and $\hat{f}_\sigma$ is the "smoothed" one, whose predictions for the input x are obtained by averaging the predictions in

neighboring points, weighted according to an isotropic Gaussian centered at x and with variance σ^2. This would result in a correction of the boundaries for the multiclass classification task, with little perturbation of the samples.

Pruning and quantization

Dense connections are needed for the training of large networks, but they are not required for achieving high accuracy at the end. In particular, connections with low weights can be *removed* (**pruning**) after training without impacting significantly the performance. Pruning can reduce the network's size by a significant margin (factors ~ 50). Additional compression can be achieved by **quantizing** weights, i.e. reducing the number of bits used by parameters.
A possible schedule for this algorithm would be:

1. Train the network
2. Remove the weights such that are below a fixed threshold
3. Train the network
4. Repeat (2, 3)
5. Quantize weight values ($32 \rightarrow 8$ bits, or even fewer)

Note that, surprisingly, if we started our training with the architecture where the pruning has already occurred, we would not be able to reach the same precision. In other words: if we randomized the remaining weights and repeat the training, usually convergence would not be reached. Several rounds of *pruning* and *retraining* can be done to achieve even smaller networks and better performances. Finally, the **threshold** is a *parameter* which has to be in turn tweaked, and eventually kept fixed or increasing/decreasing depending on our needs.

The issue of model interpretability

Due to their hardness to be interpreted, deep networks are often called **black box models**. However, as long as they are getting more diffused in real-life applications, more pressure is being put in building **explanaible models**, specially in *critical fields* (e.g. self-driving cars, clinical diagnosis, military technologies...). Thus sometimes an *understandable* model with *less accuracy* is **more preferred** than a *high-performance* but *mysterious* deep network.

It is really **difficult** to *interpret* the *internal code* of a deep net, and there are no clear instructions for doing this.
One possibility is to **visualize features learned** by the network. In the case of the *first hidden layer* it is a quite **straightforward** procedure: we simply project the weights in the input space, as in Fig. 2.27, where we can understand that this specific first layer acts as an *edge detector*. When plotting first layers weights, one notices that this is a common feature for *CNN*s: they mostly

detect edges with different orientations, color detectors and *simple features* etc.

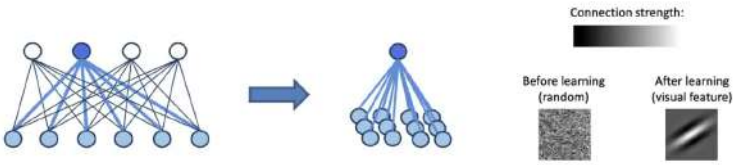


Figure (2.27) – For the first hidden layer it is straightforward to understand the feature learned: in this case, it simply detect an oblique edge.

In order to *extend* this approach to **deeper** layers of the network, we still **project** the weights in the input space, by *linearly* combining the weight matrices of *all lower* layers. This should be done with attention, since we know as layers perform *non*-linear operation. Moreover, it works only for quite *shallow* networks such as the one in Fig. 2.28. For more complex networks, one can take a look at `https://distill.pub/2017/feature-visualization/`: the basic idea behind this, is to generate synthetic images that **maximally activate** the response of certain neurons, thus being able to interpret what feature they encode. Finally, one can look at the example that *mostly activate* the neuron, to see how the resulting feature is related to the original image content.

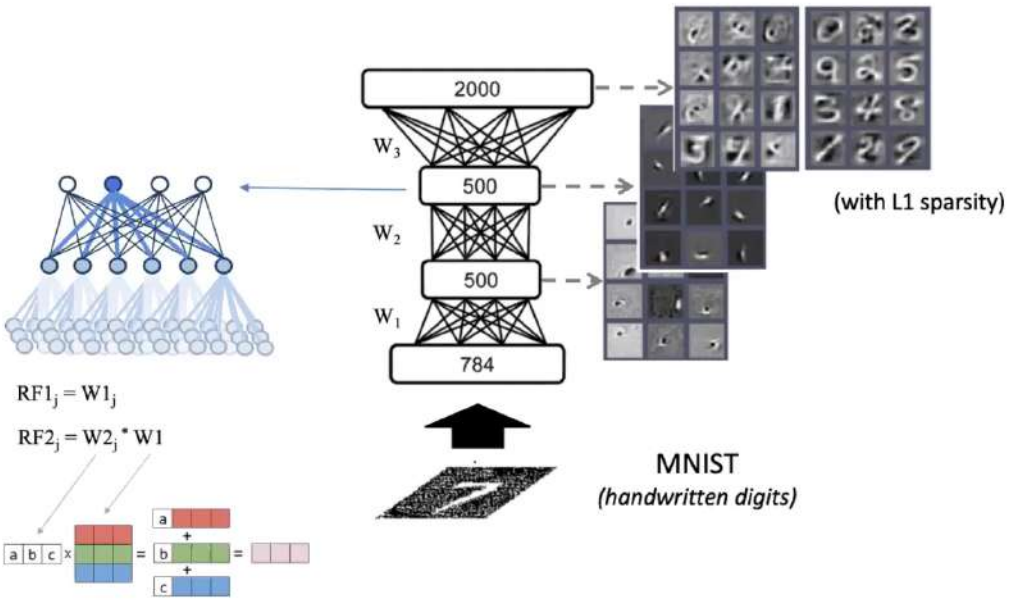


Figure (2.28) – We want to visualize the *Receptive Field* (RF) of the *second hidden* layer. We must combine the weight matrices of also previous layers, thus obtaining some sort of *average*. Specially for the first hidden layer, the feature learned are really simple and are mainly focused on specif *spots* of the image. The *second* hidden layer, however, has more elongated figures reminding of an *edge detector*. In the very *last* layer, structures are way more difficult to be interpreted. A possible help for this is to introduce L^1 sparseness regularization term, thus making units more specialized to classify very specific inputs.

Using the same techique to an **unsupervised autoencoder** trained by Google

using Youtube frames in 2012, returned very interesting results. *Without any supervision*, the network was able to learn extremely abstract features, such as *prototypical faces*, and there was even a neuron that maximally fired when a *cat* appeared!

CHAPTER 3

RECURRENT NEURAL NETWORKS

3.1 Recurrent Networks

(Lesson 8 of 27/10/20)
Compiled: September 20, 2021

Let us consider a new class of models. In feedforward neural networks, the *order* of samples did not matter: training data (i.e. every sample) was considered to be **independent** of each other.
However, this is not what happens in real life. Usually, we have access to some **context** (i.e. past or background information) which can aid in making predictions. This means that *sensory states* are **strongly correlated** with previous ones: instead of having a single input, we need to consider a **sequence** of samples temporarily related. Previous states can be used also for **predicting**, or anticipating, new inputs, "priming" a more informed response (top-down processing). Thus, a model, trained using such approach, could be used also to **generate** new data even in absence of stimuli. In the brain, sequences can be processed thanks to **feedback** connections, forming *cycles* in the neural network and allowing a backward propagation of information.

In deep learning, we use instead **Recurrent Neural Networks**, that exploit a set of *time-delayed* feedback connections to *store past information* in the *hidden* layer. Here the **inputs** are an *ordered* **time series** of vectors $\boldsymbol{x}_t$. The **hidden layer activation** $\boldsymbol{h}_t$ depends both on the new input $\boldsymbol{x}_t$, and the activation of *previous* timestep $\boldsymbol{h}_{t-1}$ [1], behaving like a **dynamical system**:

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1},\ \boldsymbol{x}_t,\ t)$$
$$\boldsymbol{o}_t = g(\boldsymbol{h}_t,\ t)$$

The **simplest** of such models, is the so called **simple recurrent network** (see Fig. 3.1). In practice, the hidden layer $\boldsymbol{h}_{t-1}$ can be implemented as a **context layer**, which stores the *activation* of *hidden units* during the previous timestep $t-1$. Therefore, the activation at the *next* time step t is influenced by the state of the context layer. The output for such networks *only* depends on the *hidden* layer, as in feed-forward case. These networks are indeed exploited in **supervised** learning.

[1] $\wedge$ and eventually even further in the past $t-2$, $t-3$...

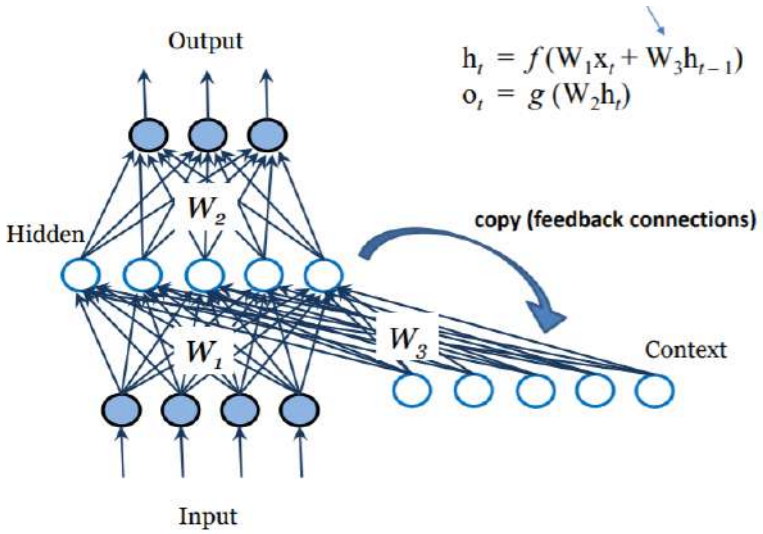


Figure (3.1) – Diagram of a simple Recurrent Neural Network for **supervised** learning. One possible application is machine translation: the input is a sequence of word in one language, and the output should be the same phrase, but in a different language.

However, one could generalize this basic structure to solve **unsupervised** learning tasks, in which we can train the network to **predict** what will be the *next input*. In this case, the only output is will be the prediction of the next input. Moreover, due to the unsupervised nature of this framework, there are no labels at all. (see 3.2)

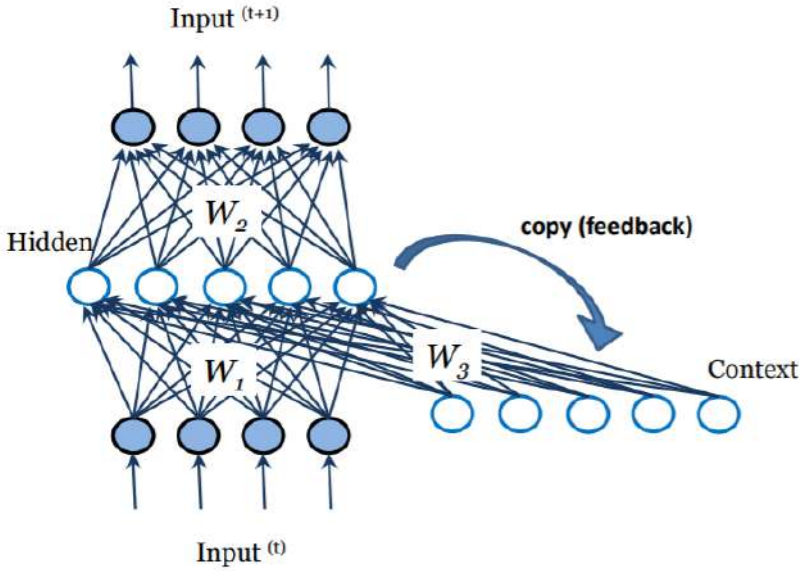


Figure (3.2) – Simple Recurrent Neural Network for **unsupervised** learning. Here the model is used to *predict* the next sample in a sequence, and so it has to *encode* statistical characteristics of the samples.

Hence, recurrent neural networks can be exploited *both* in **supervised** and **unsupervised** learning tasks.

As already said, one possible application is related to *inferring* structure taking

into account time. One implementation[2] was to train the network with *letter sequences* forming English sentences. Its **task** was therefore to learn to predict the *next* element in the sequence based on the context (see Fig. 3.3).

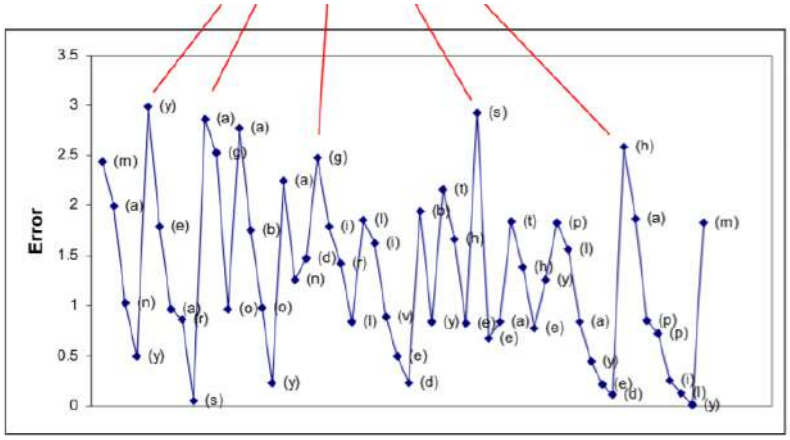


Figure 6. (a) Graph of root mean squared error in letter-in-word prediction task.

Figure (3.3) – Unsupervised RNN trained on letter sequences forming English sentences. Note how after a few letters the error decreases, since it is easier to predict a *completion* for a word. However, when a new word is starting, the error jumps again to a high value: predicting consecutive words require not just a knowledge of the vocabulary, but also grammar and semantics.

According to this implementation, the *hidden activations* of an RNN were **encodings** of *input sequences*, so essentially vectors. Thus, when trying to understand what features the network had learned, we obtained something that was similar to Fig. 3.4. Moreover, by inspecting the (Euclidean) distances between such vectors, we could see how much the model considered two sequences to be similar: as one can see, the network was able to implement also some semantic analysis by clustering nouns, verbs according to their statistical usage (i.e. based on the surrounding context and not on letters).

[2] ∧ Elman, 1990

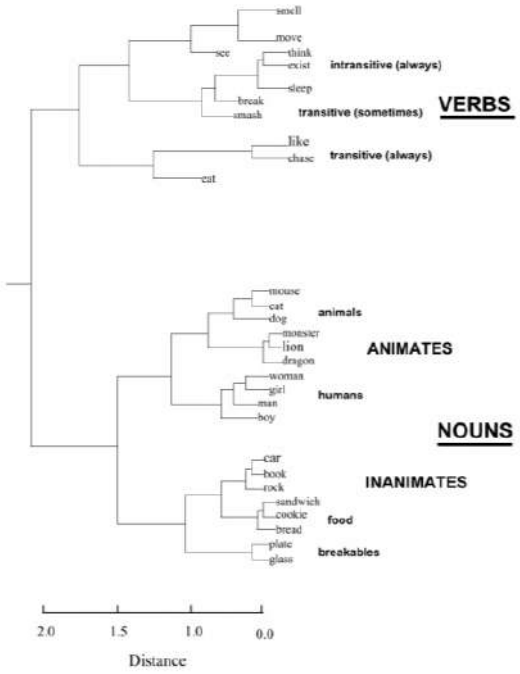


Figure (3.4) – Distances between internal representations (activations) for different words encode similarities. In the case of an RNN trained on sequences of letters, these distances encode a *syntactic similarity*, e.g. close vectors represent words falling in the same syntactical category (verbs, nouns, etc.).

Training RNN - Backpropagation through time

Let us now see how we can **train** such netwoks. One can show that RNN can be *unrolled* into a *Directed Acyclic Graph* (DAG), so basically a Feedforward Net, as in Fig. 3.5.

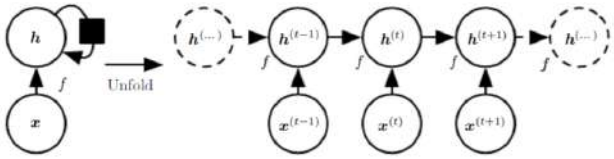


Figure (3.5) – DAG returned by any Python Library (TF, PyTorch) of a RNN and its unfolding. The black box points out a delay is present.

This is a really important and useful point: **algorithms** used for training feed-forward networks can be adapted to RNNs. In particular, backpropagation is extended as **backpropagation through time**. However, the direct consequence is that any RNN can be seen as a very *deep* network, with **parameters sharing** (i.e. replicated), see Fig. 3.6. Nevertheless, since we are trying to train a deep neural network, we can encounter the same problem we have already faced: *vanishing gradient*. A possible solution was therefore *breaking down* the *time sequence* into shorter pieces (*truncated BPTT*), so updating weights every n timesteps, thus the resulting network having less number of hidden layers.

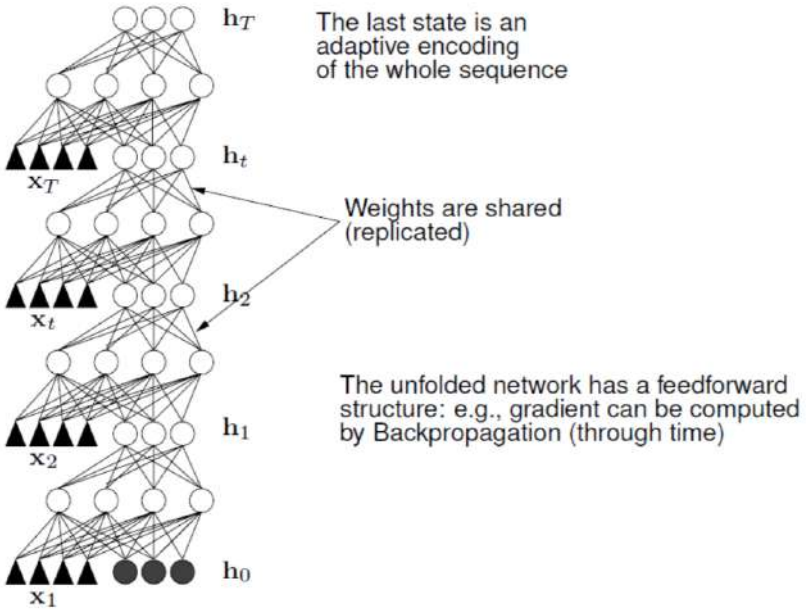


Figure (3.6) – Backpropagation through time is just backpropagation on the *unrolled* RNN, with weight sharing for the corresponding connections.

Thus, there are several **ways** to *train* such a network. A list of the possible *learning regimens* is:

- **Sequence-to-sequence**: each input generates an output (a prediction), converting a starting sequence into an output sequence. The sequence to be predicted could be the input sequence itself.
- **Encoder-decoder**: each input is first encoded into a fixed-length vector, fed to the network, and only *at the end* the entire output sequence is generated. This is useful when important information can be present in the *last samples* of the starting sequence, for example in machine translation[3]. However, this requires *compressing* all the input information in just a single activation vector (the one at the end), which will then generate all the output. So, significant information (especially of *older* samples) can be lost.
- **Sequence-to-output**: one single output is produced at the end of the sequence. This is useful in sentiment analysis, where *all* the context is used to determine whether a sentence is positive/negative/hilarious etc.

There are also some **universal approximation** properties, though mainly theoretical, also for the case of RNNs:

- Any **computable function** can be approximated by a finite RNN (assuming unbounded precision for the activations). The basic idea for the proof: it is possible to use a recurrent network to simulate a pushdown

[3] ∧ as an example, some languages present articles/prefixes, which are not really useful to either understand the meaning or predict the following words. So *all* context information has to be exploited.

automaton with two stacks, in an equivalent way of a Turing machine. However, one should take into account that the neuron's state must be a rational number with unbounded precision.

- Any **finite time trajectory** of a given n-dimensional dynamical system can be approximately realized by the state of the output units of a continuous time recurrent neural network with n output units, some hidden units and an *appropriate initial condition*.
- Any **open dynamical system** can be approximated with an arbitrary accuracy by an RNN defined in state space model form.

But, as always, there is no guarantee that these results are reachable in practice being all these statements mainly theoretical. However, this should point out how much **powerful** are RNN.

3.2 Long-Short Term Memory (LSTMs)

Unfortunately, RNNs in practice are not *as powerful* as we would expect. In fact, they can easily learn **short-term** temporal dependencies, in which the required information for prediction is *close* to the output in the unrolled graph (see Fig. 3.7).

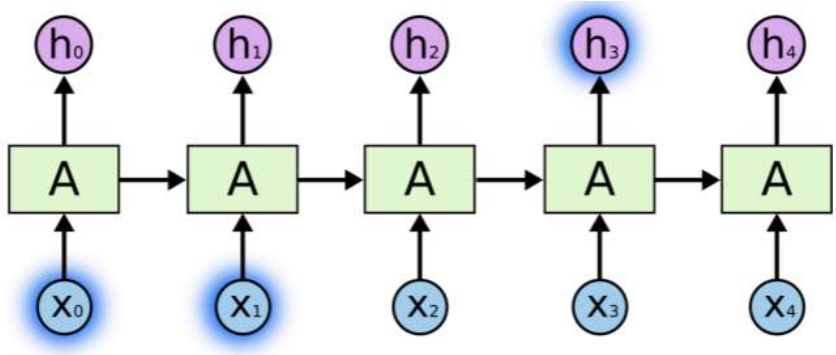


Figure (3.7) – If critical information for h_3 is contained in x_0 and x_1, it can be easily retrieved, since these vectors are *close* in the graph.

However, learning **long-term** dependencies is practically hard due to **gradient vanishing** (i.e. traversing *repeated links* leads to a *decay* of the gradient information).

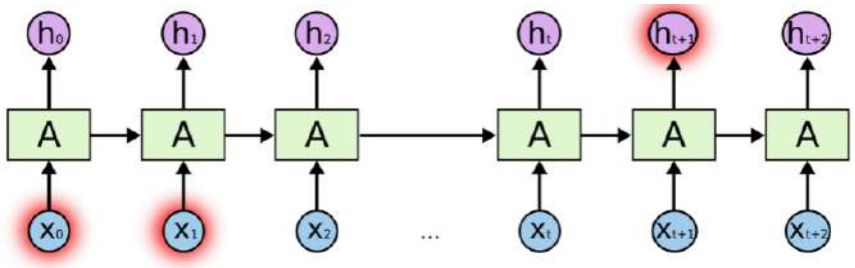


Figure (3.8) – If critical information for the output h_{t+1} is available only in inputs that are *far away* in the graph, then it will be hard for the network to retrieve it.

The size of the *time-window* we want our network to store information might be a simple sentence, or even a whole paragraph, depending on the specific task we want us to solve. There is no a priori *optimal* size: it is a *design choice*. Therefore how much context to use depends on the *domain*. However, the larger the input, the more issues may arise.

A possible *solution* to these problems is by using a different architecture, called **Long-Short Term Memory** (LSTM) (see Fig. 3.9).

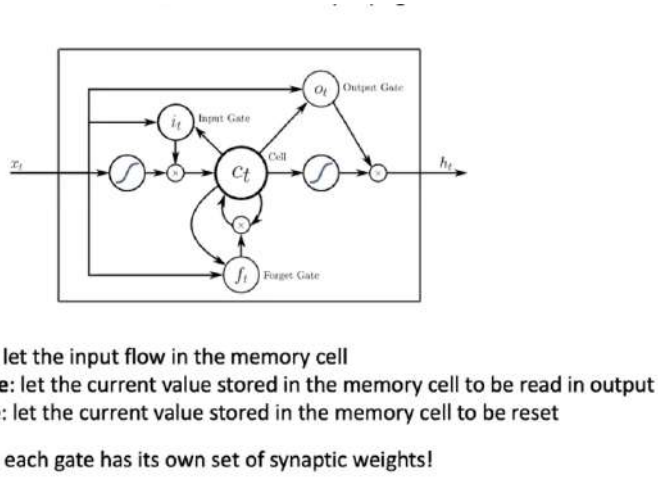


Figure (3.9) – One of the first LSTM architectures, and by what *gates* it is composed.

Here, the network inside each cell consists of *several layers*, which act as **control gates** on the cell state. These units must learn *how much information* should be retained in temporary memory, and *how* to *forward* it, i.e. how it should be propagated.
Let us define what are the **types** of *gates* implemented in a LSTM network, and compare with Fig. 3.9:

- **Input gate**: once the input has been processed according to some weights, it is usually fed to a *sigmoid* activation function. According to its value, which is forced to be in the range $[0, 1]$, it is multiplied with the *input gate* in such way that acts a filter on what to keep/ignore. The output of this stage is finally added to the actual state of the *cell*. Essentially, this gate acts as a *filter*.

- **Output gate**: as in the previous case thanks to the *sigmoid*, it selectively chooses what to return as the output of the cell.

- **Forget gate**: its role is to selectively decide whether to manipulate the actual state of the *cell* itself.

A more **recent visualization** for these networks is the following. Let us firstly introduce a **standard recurrent chain** (see Fig. 3.10), in order to more easily spot the difference with a LSTM network.

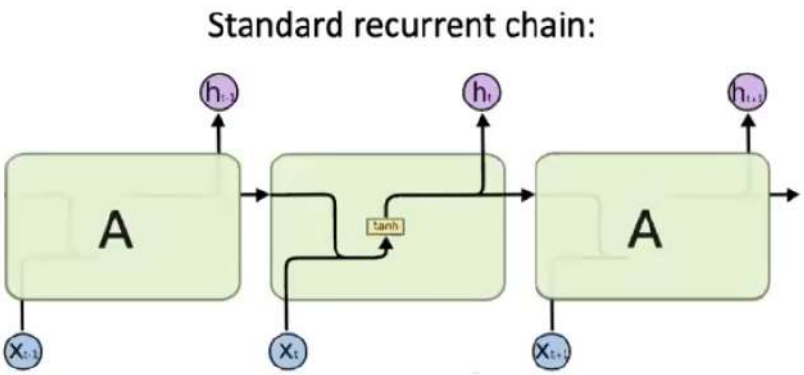


Figure (3.10) – Standard recurrent chain architecture. The input at recurrent timesteps is combined with the previous hidden layer activation in order to produce some output and the new hidden layer activation, which will be used together with the next input.

Instead, in an **LSTM** we have the gates shown in Fig. 3.11.

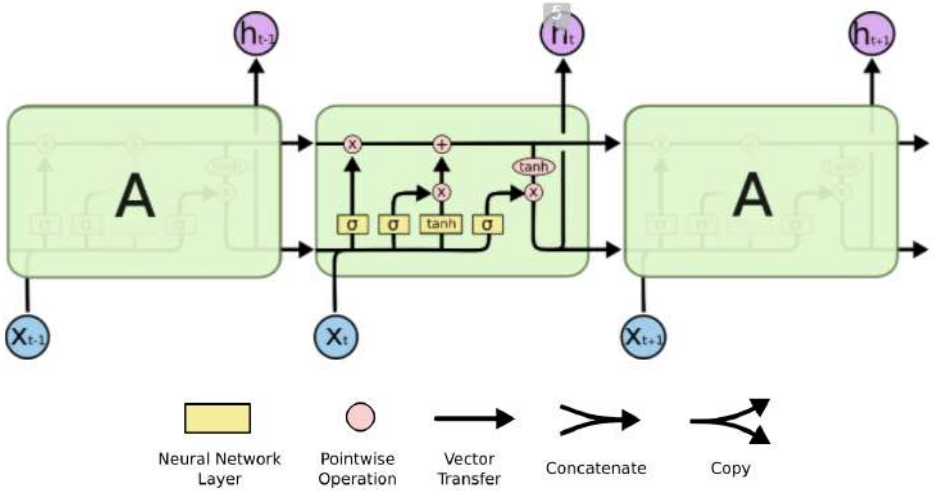


Figure (3.11) – Diagram showing the internal structure of a cell in an LSTM network.

The basic entities involved in this network are:

- **Cell**: its role is to propagate information over time.
- **Gates**: they can alter the *cell state* by adding or removing information through sigmoid and multiplicative connections.

Indeed, a cell has an internal state C_{t-1} which is updated during each time-step by using the available activations h_{t-1} and the new input sample x_t.

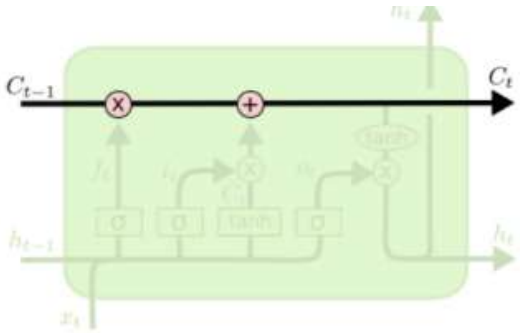


Figure (3.12) – The internal state of a cell is represented as the top black line in the diagram. Element-wise operations (pink circles) are used for updating its content during a time-step.

Updating the state involves two phases:

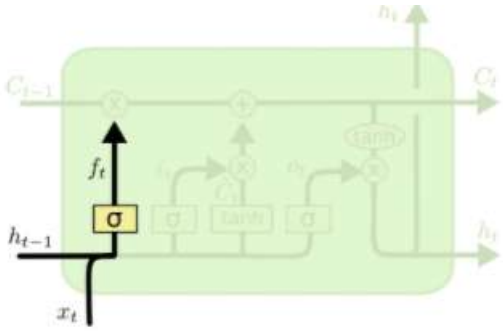
1. First, thanks to the **forget gate**, the **unnecessary information** is **forgotten** by multiplying all the entries $\boldsymbol{C_{t-1}}$ with the entries of a *mask* $\boldsymbol{f_t}$, which are all $(\boldsymbol{f_t})_j \in [0, 1]$.

   This means that if $(\boldsymbol{f_t})_j = 0$, the j-th component of $\boldsymbol{C_{t-1}}$ is zeroed and thus forgotten, while if $(\boldsymbol{f_t})_j = 1$, it is kept unchanged (remembered). In other words, $\boldsymbol{f_t}$ acts as a "filter", letting only the necessary information in $\boldsymbol{C_{t-1}}$ flowing through.

   The mask $\boldsymbol{f_t}$ is computed by a hidden layer having as inputs the previous activations $\boldsymbol{h_{t-1}}$ and the new sample $\boldsymbol{x_t}$. This computes a linear combination of the inputs, which is fed through a **sigmoid** activation σ, so that the results will all be in the range $[0, 1]$ as required:

   $$\boldsymbol{f_t} = \sigma(\mathrm{W}_f \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_f})$$

   Here W_f is the weight matrix for this **forget** layer, and $\boldsymbol{b_f}$ is the bias vector. These are all parameters learned during training. The notation $[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}]$ denotes the *concatenation* of the two vectors, which are the two inputs of the hidden layer.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure (3.13) – Diagram for the **forget** layer. The black lines of the inputs converge into one line (indicating *concatenation* of the vectors), which is processed by the yellow rectangle (hidden layer), with a σ activation function.

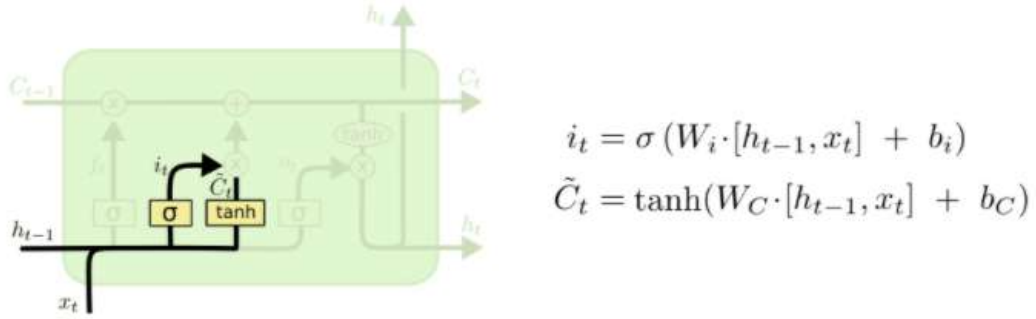
2. The next step is now to **update** the **cell's state** using new data. This happens in two steps.

First an input mask $\boldsymbol{i_t}$ is computed, controlling *which entries* of the cell's state are to be updated and to *what degree*. This is done by a hidden layer (the **input gate**) similar to the one used for $\boldsymbol{f_t}$, but with a new set of parameters:

$$\boldsymbol{i_t} = \sigma(\mathrm{W}_i \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_i})$$

Then, another hidden layer computes the *new data* to be inserted, this time using a tanh activation, which produces output entries $\in [-1, 1]$ (the state entries do not need to be all positive):

$$\tilde{\boldsymbol{C}}_t = \tanh(\mathrm{W}_C \cdot [\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_C})$$



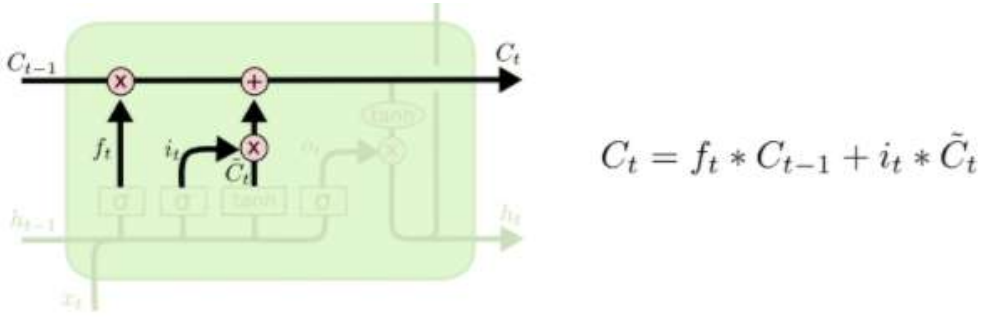
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure (3.14) – Diagram for the **input** gate, which inserts new data into the cell's state.

So, the new state $\boldsymbol{C_t}$ becomes, after *forgetting* past information and *inserting* new data:

$$\boldsymbol{C_t} = \boldsymbol{f_t} * \boldsymbol{C_{t-1}} + \boldsymbol{i_t} * \tilde{\boldsymbol{C}}_t$$

where $*$ denotes element-wise multiplication of vectors.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

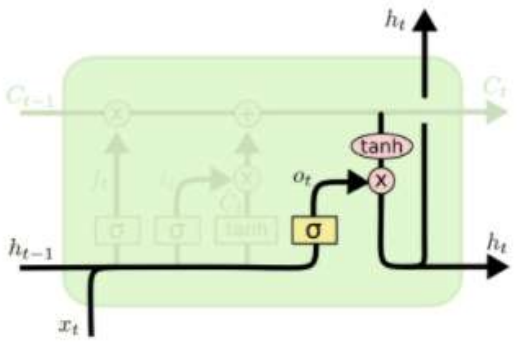
Figure (3.15) – Diagram for the cell's state update rule.

3. Finally we need to **compute** the **new activation** $\boldsymbol{h_t}$ for the cell. The idea is to again generate a mask $\boldsymbol{o_t}$ to select *which part* of the *current state* should be included in the activation:

$$\boldsymbol{o_t} = \sigma(\mathrm{W}_o[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + \boldsymbol{b_o})$$

The new activation $\boldsymbol{h_t}$ is computed by *activating* the cell's state $\boldsymbol{C_t}$ with a tanh function (since the entries of $\boldsymbol{C_t}$ may not be normalized) and then *filtering* the results with the newly computed mask $\boldsymbol{o_t}$ (**output gate**):

$$\boldsymbol{h_t} = \boldsymbol{o_t} * \tanh(\boldsymbol{C_t})$$



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure (3.16) – Diagram for computing the **output** of a cell, i.e. the **output gate**.

This kind of architecture proves to be very powerful in many domains (speech recognition, NLP, sequential processing). For some examples and a discussion of several variants, see http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Let us, as an **example**, mention some interesting results returned by one of the first **character-level text generation models**. An important implementation of LSTM network dates back in 2011 by Sutskever et al., it was trained using various large-scale corpora taken from Wikipedia, NY Times and ML papers. The input sequence was analyzed at a character level.
Surprisingly, using as input even *particularly* uncommon sentences (i.e. *bias*) such as "the meaning of life is ..." to be completed, the structure of the output was really well formed despite the general lack of meaning. This was remarkable since all text-generation past models, relying on Markov chains, were returning very stereotypical and basic sentences.
However, in NLP, it is a common practice nowadays to use *word embedding*, i.e. not to work at *character-level*. That is to say: we feed the network with a word at every time step, but only after having **encoded single words** into *fixed-length vectors* via word embedding algorithms, which are often based on *statistical analysis*.

3.3 Attention-based models

(Lesson 9 of 03/11/20) Compiled: September 20, 2021

While LSTMs are good at capturing *long-term dependencies*, they are still quite limited by the necessity to *compress* all the *dynamical information* needed into a single *static state* of fixed size. Hence, LSTMs have already become *obsolete*, in favor of **attention-based models** (which are known as *transformers*).

A better idea is indeed to provide a way for the model to *selectively* "look back" to previous states during decoding, and decide how much **attention**

"pay" to each of them. To do this in practice, we take a *weighted sum* of all the encoder inputs so far, then passing it to the *decoder hidden state*.

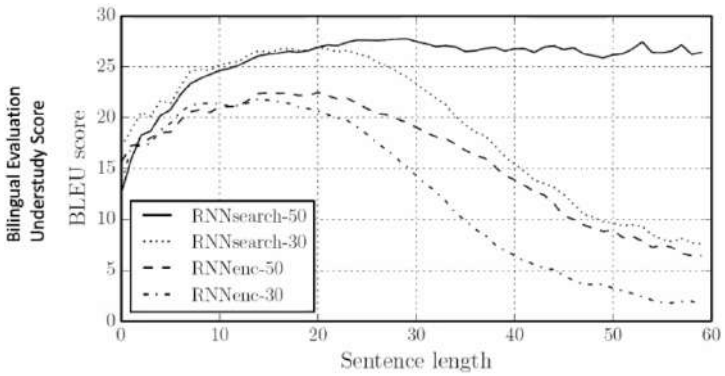


Figure (3.17) – LSTM based encoder-decoder (seq2seq models) have trouble in translating sentences that are too long (BLEU is a metric for translation accuracy), since they map an *entire sequence* into a *fixed-size* vector [Markovian assumption]. On the other hand, an attention-based system, such as RNNsearch-50, does not have this problem, since it can selectively "look back" to previous states.

The first idea for attention appeared in a paper by Bahdanau et al. (2014). Later on, the famous paper "Attention is all you need" (Vaswani et al., 2017) demonstrated that "attention" can be used to entirely replace the recurrent connections in RNNs!

Let us firstly try to define what happens in a *encoder-decoder* architecture (*seq2seq* model), when implemented for a **Neural Machine Translation** task. Let us assume we want to translate a sentence from language "A" to a language "B". For every word contained in the first sentence, the encoder (*Encoding stage*) tries to compress the information into a single vector which will be used in the *decoding stage*, in order to form the translated sentence in language "B" word by word. Note as, since the whole information is retained in a vector, the number of words of the sentence to be translated is *not* important. The crucial point, and therefore the **limitation** of LSTMs, is that between the *encoding* and *decoding* stages the only information exploited is contained in a *single* hidden state, resulted from the compression of *all* previous input.

The basic idea of **attention mechanism**[4] is to *store all* hidden states produced during the *encoding* of a sequence in a **buffer**. Then, at *every decoding step*, the **decoder** assigns to each hidden state a **score**, which is used to amplify the *most relevant hidden* states corresponding to specific portions of the input (for example via a *softmax*). Finally, all these states are combined into a single sum, weighted by such scores. This will constitute the **context vector**, which will be used for the output construction, after having combined it with the hidden state. For a visual explanation of what happens during a single timestep of decoding, see Fig. 3.18.

[4] ∧ one may want to take a look also at https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

Figure (3.18) – Sketch depicting a *decoding* timestep of an attention-based model, thus returning a *context vector* used at timestep t_4. Once inputs have been already prepared (i.e. encoded) by the *encoder* (returning h_1,h_2,h_3), the *decoder* at every timestep gives, based on its hidden state, some weights to the input vector. The weights do depend on the *ordering*. By combining the weighted inputs, it returns a **context** vector, which will be helpful for constructing the output.

The **attention mechanism** operates over all the *stack* of hidden representation for the input sequence, namely h_1, h_2 and h_3 in Fig. 3.19. These, combined somehow, return a *context* vector which is used with the *normal* **hidden** state of the cell-decoder[5] to produce some *output*.

[5] ∧ "normal" in the sense that we have already encountered when studying LSTM

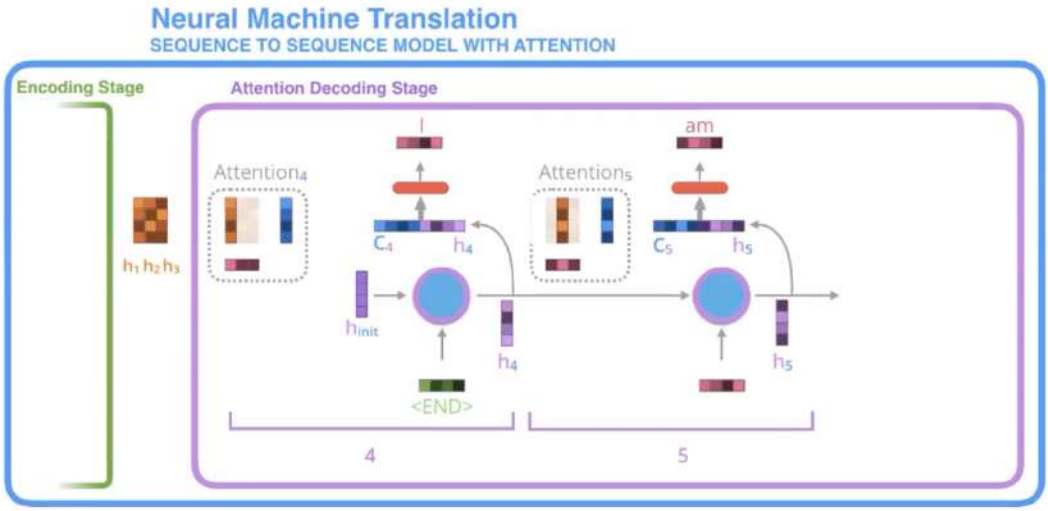


Figure (3.19) – Sketch depicting two consecutive timesteps *decoding* of an attention-based model in a translating *French→English* model. The original input was "Je suis étudiant", which stands for "I am a student".
The cell state h_4 is combined with the context vector C_4 to produce an *output*. Note as the latter is fed as *input* to the cell state at the next stage, as one can see below the cell in step 5: in this way one takes care of the subject of the sentence (*"Je"→"I"*) when trying to translate the verb (*"suis"→"am"*).

Results using *attention mechanism* are way better, rather then using a simple LSTM network, and the *score* does depend on the **order**, as one can see in Fig. 3.20.

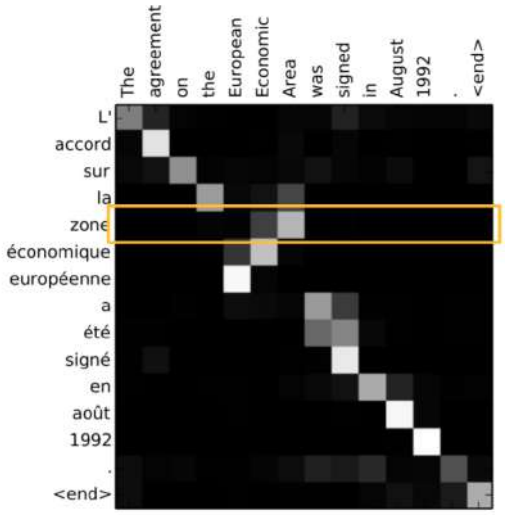


Figure (3.20) – Attention scores during a machine translation task from English (up) to French (left). Note how high scores denote *corresponding* words, and they do not go exactly in the original order. Semantic order might indeed vary in different languages. In this way, the model can account for difference in the *order* of words: this action performed by hidden units. In more complex cases (e.g. words having multiple meanings), more than one entry can be active at a time, i.e. the model can "pay attention to" multiple states at once.

Let us now introduce some **noticeable examples**.

GPT (OpenAI)

Transformers performance can be further improved by exploiting the considerable amount of unlabeled "raw" text available (order of GB, TB). The language model is initialized using **G**enerative **P**re-**T**raining, thus learning an internal representation of text. Finally it is fine-tuned on specific discriminative tasks (semi-supervised learning). It returns accurate performances in many different areas: natural language inference, answering questions, text classification and semantic similarity. Given some context it is able to produce **really high-quality** texts, furthermore their latest implementations have not been publicly available due to ethical concerns: texts written by human and by GPT-2 (and nowadays GPT-3) could not be distinguished from each other.

BERT (Google)

Bidiretional **E**ncoder **R**epresentation from **T**ransformers takes into account **future** as well during *on-line* decoding. Clearly, this is very computationally demanding, but it models what usually happens in the brain: the internal representation of a word dynamically changes according to what will be said next.

CHAPTER 4

UNSUPERVISED LEARNING

In general, most of the data in the universe is **unlabelled**. However, some information about its structure can be learned by **unsupervised models**. As an example, when were trying to learn the *statistics* of data finally making predictions using recurrent networks, in a certain way, we were making some unsupervised learning due to the absence of any *ground truth*. For these reasons these models are considered in between of *supervised* and *unsupervised* learning.
Generally, a *neural network* will help reducing dimensionality of the data, via **extracting** its most **important features** to better **clusterize** it.
The **major advantages** in using such a learning framework are:

- **No need of labeled data**, which often requires lots of effort to be obtained. Moreover, *raw* and *unstructured* Information is the most available in the environment
- **Generalization also for supervised tasks**: once the environment has been learned through a model (*representation learning)*, it can be applied to learn more easily *supervised tasks*
- **Biologically/Psychologically plausible**: it is the most children and animal modality for learning.

On the other hand, there are some **shortcomings**:

- **No hints** on *what* features are the most relevant to be learned
- **Computationally demanding**, needing both powerful hardware and big data (and experience on how to proceed...)
- **Impossibility to infer causality**: by mere and passive observation, one can not infer *causal* relationships but only *correlations*

To better extract the data, we need a **good representation** for it. In this way, it becomes easier to extract and spot **shared features** that belong to a certain class (see left picture in 4.1, where visually one can tell the *color* is the discriminant feature). However, as a better approach, we can even extract useful descriptive features, finally learning some mappings from these

"higher-level" internal representations which might turn out to be useful to discriminate objects. (see Fig. 4.1 (right))

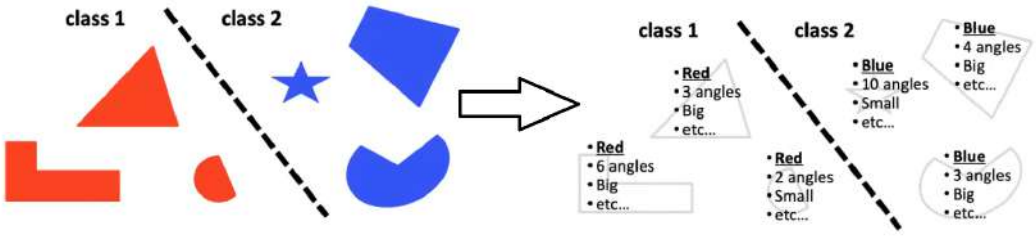


Figure (4.1) – Starting from the representation on the left, where we know that there are *two* classes, we try to extract features as on the right. However, the *most important* one that is discriminant of the class appears to be the color: if we are asked to classify a new shape being either *blue* or *red*, we are quite confident which will be its class regardless the complete list of feature we make (right side). For more complex task, however, this more abstract approach is needed.

However, sometimes in real world situations things are not so simple and one cannot always exploit an immediate property (e.g., color) to "learn" the data: for more complex tasks, such an approach, namely extracting *all* features thus building an *internal representation* of an instance, is needed. Indeed by applying **supervised learning** one is able to tell what are the *most relevant features*. This is what is usually done by our brain when trying to discriminate objects in *visual object recognition*: initially it tries to *extract features* (e.g. the context [1]), and finally places the *margin* thus being able to state the nature of an object [2]. The brain, in other words, **disentangles** the problem by learning some useful (non-linear) descriptors to perform better classification tasks. Note as there are two *complementary* approaches for learning representations, thus **reducing the dimensionality** of the dataset:

- **Feature extraction**: reduces the *number of descriptors* by only considering the *most informative* ones, or replacing them via the creation of *more abstract features* when some correlations are spotted.
- **Clustering**: reduces the *number of samples* by grouping them and considering "group prototypes".

In unsupervised learning we **assume** that data nearly always lies on (or close to) a **much lower-dimensional, smoothly curved manifold**, that locally appears to be a Euclidean space. Every point implies the *existence of* (linear) *transformations* that can be applied to move ourselves on the manifold from one position to a neighboring one (see Fig. 4.2). Biologically, this is what **ventral visual stream** does: images are entangled in the retina and are *gradually projected* in order to finally abstract them in the *associative area* of the brain (temporal cortex).

[1] ∧ background, or any other useful information

[2] ∧ for example whether "it is a car" or "it is not a car". Note as "being in the savanna" slightly changes the context information, thus allowing to hypothesize it is more likely it can be an elephant rather than a car. The context would conversely if the information were "being in a city".

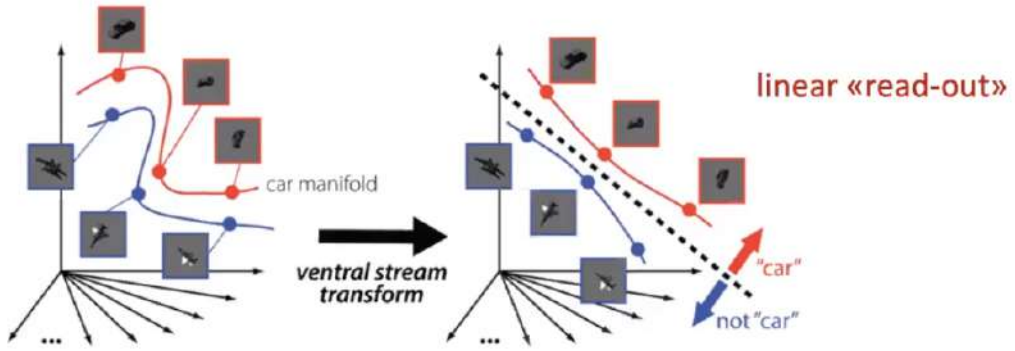


Figure (4.2) – Input data manifolds are strongly entangled (i.e. the manifold exhibits many curves) and we almost know nothing: we want to disentangle them, thus extracting features into a new space that is able to *linearly discriminate* inputs. For example, one might want to apply the discrimination of "being a car".

Our goal is now to show how it is possible to **build abstract representations**, without the presence of *any* supervisor.

4.0.1 Autoencoder

It is a **deterministic** model trained using *error back-propagation*, like feed-forward networks. The only *difference* relies in the **output**, which ideally is a *copy* of the input itself. Its goal is therefore to build a compressed **internal representation** of some input, trying to encode it as better as possible using a reduced number of variables. A simple sketch of such architecture is the following (see Fig. 4.3).

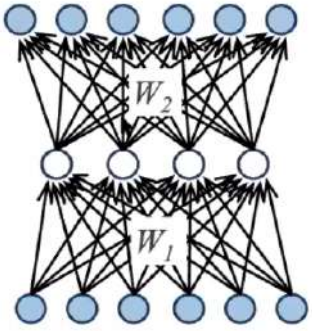


Figure (4.3) – Sketch of a simple *autoencoder* architecture.

The loss function is simply the **reconstruction error** (e.g. MSE):

$$MSE_{train} = \frac{1}{m}\sum_{i=1}^{m}\left\| \underbrace{\hat{y}^{(i)}}_{\text{training pattern}} - \underbrace{y^{(i)}}_{\text{model reconstruction}} \right\|^2$$

With referral to Fig. 4.3, we can use *different functions* for encoding and decoding, namely $W_1 \neq W_2$. In order to further simplify the model, one can regularize it by *imposing* the *same function* for encoding and decoding: $W_2 = W_1^T$

According to the **number of hidden** units n_{hidden}, the network may exhibit two different *behaviors*:

- $n_{hidden} \geq n_{visible}$ without any additional constraint, the network will simply learn to *copy* the input (**overcomplete code**)
- $n_{hidden} < n_{visible}$ the network will extract *relevant features* from the data (**undercomplete code**).

Let us focus on the **undercomplete code**, which is the most interesting to us. It was shown that using *linear activation functions* the network will learn to return as output something very close to PCA result, with the dimensionality of the spanned subspace being dependent on n_{hidden}. Conversely, using *non-linear* activations, the features learned can even be nonlinear!
Moreover, the **feature extraction** process can be further improved by introducing additional **regularizers**, which allow to efficiently use also *overcomplete codes*! In this case, when imposing sparseness (L^1 penalty on hidden *activations*) one obtains **sparse autoencoders**. Alternatively one may want to try to artificially *add some noise* (and changing its type every time) to the *input* thus creating a **denoising autoencoder** to filter it out, or another possibility are **Generative (stochastic) Autoencoders** which will be a topic for later in the course.
Finally, one should mention that also *stacked* autoencoders with *multiple processing layers* have been developed, due to latest improvements in training deep architectures, and *Convolutional Deep Autoencoders* shall be used for image-related domains.

4.1 Energy based models

(Lesson 10 of 10/11/20)
Compiled: September 20, 2021

In general, networks are not divided into layers, but consist of some units connected with every others with no hierarchy: they indeed lack of **directionality** in processing. Indeed sometimes it might be needed to change weights *only* in the input layer, but *only* because of the input itself, which can be noisy, and not according to some algorithm of backpropagation. In this case, the "need" to change weights comes from the input, and not, as an example, from a misclassification. Moreover, in a biological framework, it is really unlikely that the human brain learns by backpropagation.
The idea of using "more generic" networks leads to **energy-based models**. They are very different from all other architectures and are still difficult to train, but perhaps in the future might lead to some more powerful algorithms.

Associative (content-addressable) memories

Let us consider a network of neurons. How can we *learn* how to connect them? From biology, we know that neurons that *fire together* are usually *wired together*. This basic idea is the so-called **Hebbian rule**, in which we tweak the

connection weights between two units *proportionally* to the product of their activations v and u, according to some factor η:

$$\Delta w = \eta \, v \cdot u$$

This can be used to *encode memories* inside the network or, in other words, to **store patterns** of a complex network.
For example, suppose that every neuron represents a pixel in an image. We fix their activations according to some target image, and then iteratively tweak their weights according to the Hebbian rule. Now, if we *corrupt* the activations and *iteratively update the neurons' activations* according to some dynamics depending on the links, which are now fixed, the *neurons* will re-configure themselves to the *original target* image thanks to the weights in such a way that some energy function is minimized. (see Fig. 4.4)

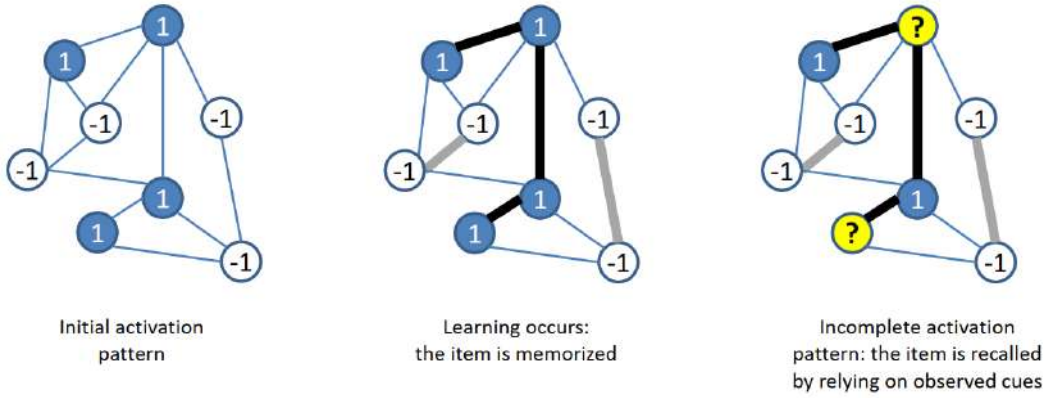


Figure (4.4) – How Hebbian learning works in practice.

Due to the fact that the network can associate a pattern, tweaking its weights, to a specific input, this process is also known as **associative memory**. To better understand it, let us make an example. This process indeed occurs in human brain when it **associates** the *faces* of people with their respective *names*, keeping memory of them. In addition to this, we are also able to *complete* the *missing* information from our memory, indeed we can imagine a friend's face when hearing their name, or conversely recall their name once having seen their face.

For sufficiently large networks, several *memories* can be encoded at once. However, now reconstruction might become more difficult: the model can find itself stuck "between two memories". This kind of behavior is peculiar of well-studied models in *statistical mechanics*. The core idea is that we are studying a system in which local interactions (i.e. the local connections between neurons) produce global phenomena (e.g. the reconstruction of a given pattern). So, the interaction of many *simple* units produces an emergent *global complex* behavior.

In Physics, the **Ising model** is one example of such systems. Here we are considering *units* (spins) which *locally interact* (they are like little magnets, which try to be either parallel/anti-parallel to each other). In the simplest model,

these spins can take only two possible values: either "up" or "down". The dynamics of the system is governed by an **energy function**, which depends on the system temperature T [3]. The total *Energy* of the system is returned by the **Hamiltonian** $\mathcal{H}(\sigma)$, and depends on the values $\{\sigma\}$ the spins can take. Moreover, this operator can be rewritten using two terms:

$$\mathcal{H}(\sigma) = -\sum_{<i,j>} J_{i,j}\sigma_i\sigma_j - \mu\sum_j h_j\sigma_j$$

where $J_{i,j}$ denotes the *local couplings*, h_j the *external magnetic field strength*, and the first term sum runs over *nearest-neighbors* i, j.
The **probability** of a given configuration at a certain temperature β is given by the *Boltzmann weight*:

$$P_\beta(\sigma) = \frac{e^{-\beta\mathcal{H}(\sigma)}}{Z_\beta}$$

where $Z_\beta = \sum_\sigma e^{-\beta\mathcal{H}(\sigma)}$ is the so called **partition function** and acts as a *normalization term*. This allows us to quantify *how likely* a configuration is with respect to another, through *global quantity* (i.e. the energy associated with a given configuration).

Over time, the system evolves towards a *low energy state*, which depending on the specific Hamiltonian can be either *ferromagnetic* where *all* spins are aligned either up or down ($J_{i,j} > 0$), or *antiferromagnetic* ($J_{i,j} < 0$) where spin directions are alternating, thus the system exhibiting some *macroscopic behavior*.
However, the interaction strengths are **randomized** (e.g. spin glass), it will not be able to reach a uniform state *global minimum* in which the total energy is minimized. Indeed energy will be minimized only *locally* (heterogeneous configuration): we call this phenomenon **geometric frustration**. The idea is that there are *many* possible (stable) local minima that can be reached. Thus, one can **exploit** the presence of *geometric frustration*: by choosing the appropriate local interactions, one can select these minima and use them to encode useful information (memories or patterns).

4.1.1 Hopfield networks

In practice, we define a **Hopfield network** as a set of **fully-connected** *neurons*, each representing a single input [4], without self-connections (to ensure stability). Activations of nodes can be only $\{\pm 1\}$, and there are **no** *hidden (latent) units* [5]. Moreover, we allow *adjusting local interactions* $J_{i,j}$ through learning: its goal is to assign high probability to the configurations observed during training. In such network, the *memory* corresponds to *dynamically*

[3] ∧ or alternatively $\beta = 1/k_B T$, with k_B being the Boltzmann constant

[4] ∧ in image processing, this is equivalent to state that every neuron corresponds to a single pixel in a digital image.

[5] ∧ it is a strong assumption. A way to overcome this is represented by Restricted Boltzmann Machines (RBM)

stable **attractors**: when presented with a new pattern, the network will recover the most similar one by gradually settling into the closest attractor, thus reaching *thermal equilibrium*. At the end, the **global dynamics** of the network is governed by *local* interactions as it usually happens in human brain [6].
Let us define the **energy function** $\mathcal{E}$ for a certain configuration as follows:

$$\mathcal{E} \equiv -\frac{1}{2}\sum_{i,j} w_{ij} x_i x_j$$

which is similar to the *Hamiltonian* used in the spin-glass Ising Model in Physics: we are accounting for *local* couplings between the neurons (and neglecting biases for simplicity, but eventually they could be included). As said, **energy landscape** is *complex*, thus having *many local attractors* (i.e. minima, with their basins of attraction), each one representing different stable patterns of activity and encoding different stored items.

We want now to understand *how* to minimize the energy, that is to say to *move towards* local minima. There are *two ways* to achieve this result: either during *learning* or during *inference*.
During **learning**, we **change the connection weights** $w_{i,j}$ so that the minima of $\mathcal{E}(\boldsymbol{x})$ correspond to the target vectors $\boldsymbol{x}_{\text{true}}$. This is done by first setting the activations to one of the desired target [7], and then iteratively applying the **Hebbian rule**:

$$\Delta w_{i,j} = \eta x_i x_j$$

with η being the usual learning rate. However, with some algebras, it can be shown that the optimal weights can be learned in one *single step*, thus sparing computations:

$$w_{i,j} = \frac{1}{k}\sum_{p=1}^{N} x_i^{(p)} x_j^{(p)}$$

where the index p denotes one of the N patterns to be learned, and k is the input size. Graphically, this means that we are "carving" the energy landscape, making $\mathcal{E}$ minimum around $\boldsymbol{x}_{\text{true}}$, and raising the energy of all the other unlikely configurations, which are not related to any patterns. This can be seen in Fig. 4.5.

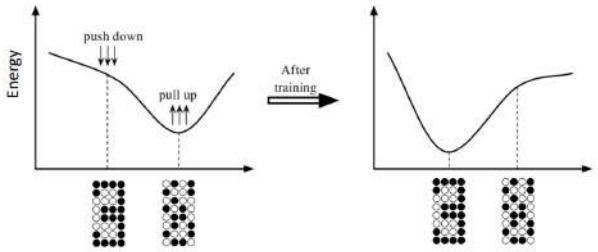


Figure (4.5) – Training a Hopfield network amounts to "carving the desired minima" in the energy landscape. After training, if the network is fed with the noisy image, it will recover the correct one (i.e. the one at the left).

[6] ∧ indeed such models have been widely used to represent human learning and memory
[7] ∧ in other words every training pattern is iteratively clamped to the network's neurons

During **inference**, instead, we provide a certain input $\boldsymbol{x_0}$, and iteratively **tweak the input** to *minimize* $\mathcal{E}$, which will lead to one of the *stored* $\boldsymbol{x}_{\text{true}}$. This is done taking into account that the activation of a neuron is computed according to the hard threshold rule, namely, according to the following local dynamics:

$$x_i = \begin{cases} +1 & \sum_j w_{ij} x_j \geq \theta_i \\ -1 & \text{otherwise} \end{cases}$$

Note as, since we are not computing any gradient, the activation function does not need to be differentiable. The idea is that each neuron *receives* the activations of the neurons it is connected to, weighted by the link's weights, as in the *perceptron*. If this weighted sum exceeds a certain *activation threshold* θ_i, then i will activate, producing a $x_i = +1$. Otherwise it will remain "off", and so $x_i = -1$. Since the network is a *dynamical system* neurons are updated: this can be done either *synchronously* (all together) or *asynchronously* (one at time).
Moreover, Hopfield demonstrated that if the weights w_{ij} *are proper*, the network activations will **always converge** toward a **stable state** (attractor) where the energy is minimized, thus recalling a correct memory. Then, only minor fluctuations will occur, with the average activation remaining fixed (thermal equilibrium).

Training can be either **completely unsupervised**, by simply *encoding patterns* into the network, or **supervised**, by *reserving some neurons as output*. In fact, when presented with a incomplete input, the Hopfield network will try to complete it and, as *output*, may be activating the relevant output neuron for a certain input class.

Let us now mention some **disadvantages** of *Hopfield networks*. Despite they are very general, this comes with the price of a **low storage capacity**. In fact, the *largest number of patterns* that can be stored at once is $0.138 \cdot k$, where k is total the number of neurons of the network (upper bound). However, in this case there will be (small) retrieval errors. To have even **more accuracy** and be able to reproduce *perfecly* patterns, we need to reduce the capacity to $k/\log k$. The *second issue*, also is that as we *increase the number of stored patterns*, the network can develop **spurious memories**, i.e. additional local minima between two "good minima". See Fig. 4.6.

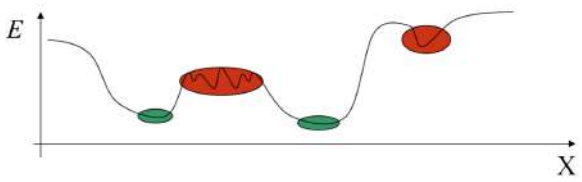


Figure (4.6) – Sometimes it happens that spurious minima (red) appear between good minima (green), the latter ones corresponding to learned patterns. Note as the spurious attractors are still higher in energy, than the correct ones.

This problem can be mitigated by introducing some **stochastic dynamics**. The idea is to **replace** the *deterministic activation* (e.g. Heaviside activation) of neurons during inference with a **stochastic function**:

$$P(x_i = 1) = \frac{1}{1 + \exp\left(-\frac{1}{T}\sum_j w_{ij} x_j\right)}$$

This is a sigmoid function [8] whose *slope* is controlled by the **temperature** parameter T. The larger this parameter, the larger the stochasticity. Mathematically it controls the slope of the activation function, see Fig. 4.7.

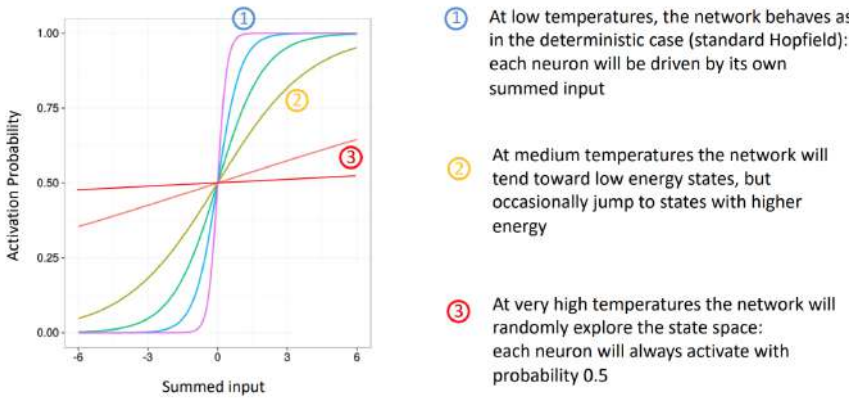


Figure (4.7) – Probability of activation as a function of temperature. Note that higher temperature means *more randomness* in the system.

In order to *reach* the best (i.e. more stable) **energy minima**, during inference we start by simulating the dynamics at a high temperature and slowly decrease T until the activations converge to a stable solution. This is the so-called **simulated annealing** algorithm. The idea is that the fluctuations at higher temperatures will stochastically "move the model away" from *weak* spurious minima towards the "distinct" correct local minima imposed by the training procedure, as one can see from Fig. 4.8.

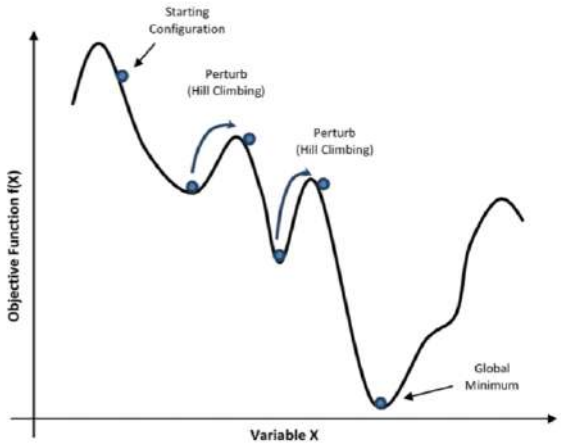


Figure (4.8) – Randomness allows the model to *explore* more the energy landscape and "jump out" local minima.

[8] $\wedge$ but depending on the specific problem $\tanh h$ can be used, too

Instead of using *expensive sampling* procedures one can think of *approximations* to retrieve a deterministic model. One of such approach is the **mean-field** one, where the *true* fluctuating activation value of a spin is replaced by an *average activation values* of its neighbors.

4.2 Generative Models

(Lesson 11 of 17/11/20) Compiled: September 20, 2021

The main *limitation* of Hopfield networks is that *all* neurons are **visible**, i.e. they correspond to inputs. In other words, they lack of *latent* units, thus being able to capture only *direct*, *pairwise* interactions.
However, we expect that adding **stochasticity** and **hidden representations** the model's performances could be improved. The problem now can be tackled using *Bayesian statistics* framework, always including the **energy** parameter related to configurations probability (*energy-based models*).
Let us now consider the usual **Hopfield network** with *fully-connected* topology, but now *add* a new set of **latent (hidden) units** H to it, in such way that they do not exactly coincide with inputs any more: we are allowed now to build **internal representations**. (see Fig. 4.9)

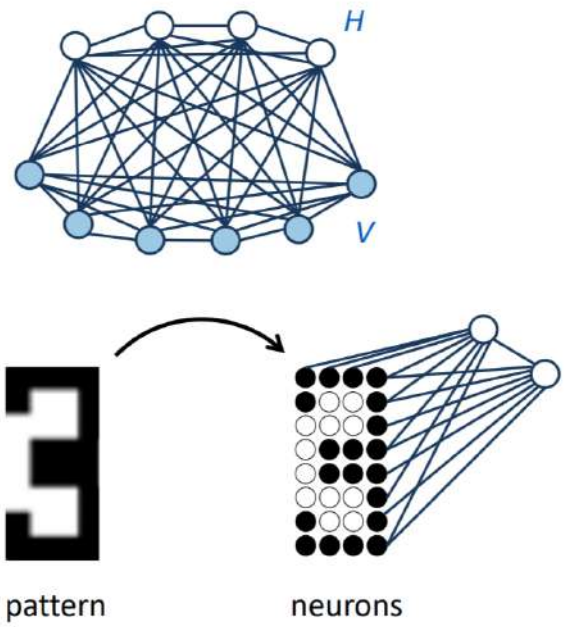


Figure (4.9) – Hopfield network + Hidden Units, in this way we are able to *build* **internal representations**. This architecture is known as **Boltzmann machine**.

The idea is strongly related to the one of *Bayesian brain*, where perception is mediated by statistical **inference**. These units act as "hypotheses" (H), which are *inferred* from the visible data (E) through Bayes' theorem:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

In other words, we have two main **tasks**: the *inference* and *learning*. During **inference**, given some observed evidence (visible units) one wants to

establish which are *the most probable* hypotheses (hidden units), able to explain it.[9] However, during **learning** we tune parameters (connection weights) of a generative model that best describes the data distribution (maximum-likelihood learning). In other words: given a hypothesis H we can *reconstruct* the evidence E supporting it, i.e. **generate data** according to the **internal representation** contained in H, namely, in the hidden units.

Let us now state what are the main features of **generative neural networks**, and how these can be exploited in Bayesian learning. Their main **goal** is to *discover* the **latent structure** of *input data* (i.e. internal model of the environment which could have produced the data), rather than simply performing input-output mapping. The *latent causes* of the sensory signals constitute an **internal representations** of the environment, and they are intrinsecally related to the hidden units, thus representing hypotheses over the data (every hypothesis being usually encoded as a a *distributed activation* across neurons). Moreover they are **updated** as *new evidence* is available. Recall that we have already seen an example of (linear) *generative model* following this same structure in the concept of **sparse coding**:

$$I(x,y) = \sum_i a_i \phi_i(x,y)$$

Indeed, we were trying to model the evidence according to our *basis* functions [10], namely our internal representation. However, our goal will be to construct more complex *non-linear* generative models.

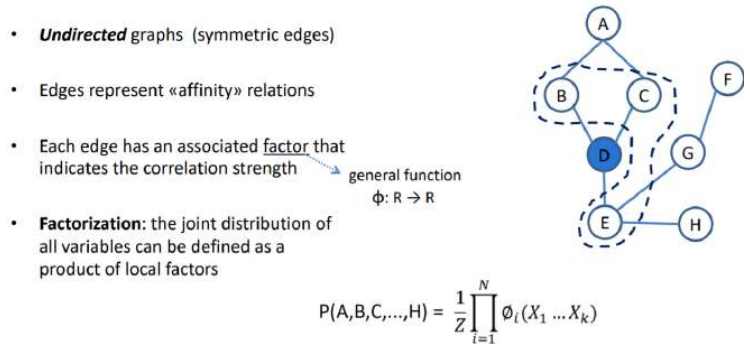


Figure (4.10) – A Markov network, which is characterised by neurons taking the role of random variables. These are connected by undirected links, which somehow "quantify" the correlations between variables.

A general network with both visible and hidden units can be described by a **Markov network** (see Fig. 4.10), i.e. a collection of nodes (neurons), representing random variables, connected by undirected links (thus symmetric) which are weighted by the *correlation strength* between different variables. Dynamically, note that every neuron can spread activation also to other neurons, even though they are not directly linked. A Markov network is just a

[9] ∧ in such networks, we refer to the "evidence" as the activation of visible units. Conversely, "hypotheses" are the activations of hidden units.

[10] ∧ in other words, we were linearly projecting the inputs onto a subspace spanned by some basis.

schematic way of showing the relation between different observables, which allows constructing a **factorized** *joint probability distribution*, encoding only the required relations. This can be estimated by *random sampling*, or by using **variational inference** (mean-field methods), in which a simpler (analytical) distribution is used to approximate a target one [11]. Recalling now the definition of **Markov blanket of a node** which is the set of immediate neighbors that, when observed, make such node *independent* from all the other nodes outside the blanket. Therefore, during inference we can exploit the *local structure* of the graph (conditional independence) to *efficiently sample* the values of our target variables that are unknown.

We want now to **reduce computational complexity**, and this can be done by exploiting *approximate inference* methods. For now, let us start by considering **sampling-based** methods (particle-based methods) [12]. The idea is to approximate marginal distributions by drawing as many instances as needed and finally computing *expectations* (discrete case):

$$s = \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f(\boldsymbol{x}) = \mathbb{E}_p[f(\boldsymbol{x})] \quad \approx \quad \hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}^{(i)})$$

Then, we can also exploit the *local structure* of the Markov network [13]: for example, conditional independent variables can be sampled separately, and finally the variables depending on them can be sampled depending on their values.

In practice, we start by generating an arbitrary (biased) sample, and then we compute new samples to gradually "fix" it by reducing the correlations. Under *ergodicity* assumption it is shown that after some time the Markov Chain will converge to a *stationary* distribution close to the target one. This is the gist of Markov Chain Monte Carlo methods. Differently, in *Gibbs sampling*, only *one variable* is sampled at a time, keeping fixed all the others:

$$T_i((x_{-i}, x_i) \to (x_{-i}, x_i')) = P(x_i' | x_{-i})$$

where x_{-i} denotes the set containing all the variables but x_i. Note as the transition probability does not depend on the current value of x_i, but only on the value of all other units x_{-i}. By iterating this process, finally we reach equilibrium, i.e., the state does not change any more. An example of the previous graph can be seen in Fig. 4.11. A possible *evolution* for Gibbs sampling could be the so called "block" Gibbs sampling: *conditionally independent* variables are sampled at the same time without affecting each other.

[11] ∧ basically, a complex distribution is approximated through a set of easy-to-handle, even analytically, functions depending of a certain set of free parameters. The goal is to minimize the distance (e.g. KL divergence) between the target and the approximated distributions, thus finding the optimal set.

[12] ∧ *variational methods* were already briefly mentioned before and will be examined in depth when discussing variational autoencoders.

[13] ∧ i.e. the fact that it is not fully connected and the idea of Markov blanket

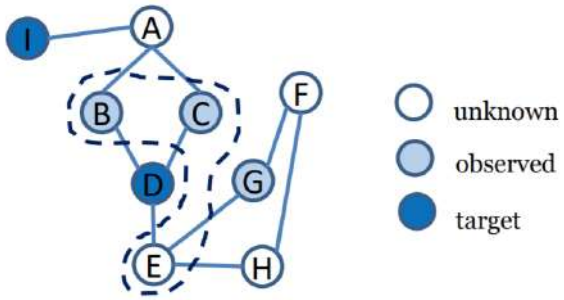


Figure (4.11) – Suppose we want to construct the probability distribution of D, and that we know B, C and G from some observations (i.e. are *constrained*). The idea is to *guess* a value for H. Depending on it (and the others) we can sample a value of E, and then one for D. From this, we can compute a new sample for E, and then also a new value for H. Repeating all these steps many times will produce distributions that *converge* to the true ones. The idea is that the initial state (which is arbitrary and biased) is quickly forgotten by this evolution process. Only the "most stable" estimates, i.e. the ones that fully respect *all* the interactions, will survive at the end of many iterations (stationary distribution). Note as everything outside the Markov blanket is ignored from D's perspective.
On the other hand, if one wants to construct the probability of I, one has to simply draw iteratively I and A until a convergence is reached. Due to topological structure, and since B and C are fixed from observations, what happens "outside" the Markov blanket does not matter.

4.2.1 Boltzmann Machines

Boltzmann machines are a *stochastic* variant of Hopfield networks that *include hidden* (latent) *units* able to learn higher-order correlations (features) from the data distribution. For some representations of such networks, see Figg. 4.12 and 4.9.

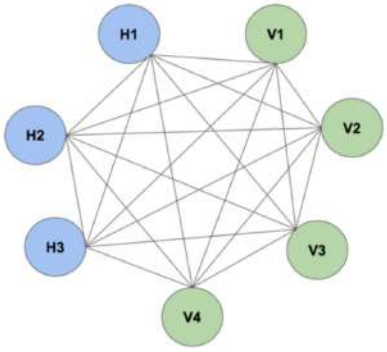


Figure (4.12) – A Boltzmann machine is stochastic variant of a Hopfield network, having a fully connected topology. Moreover, some units are latent and are devoted to learn features, while others only to receive inputs. Finally, these networks include stochasticity.

The idea is to use an **energy function** $\mathcal{E}(\boldsymbol{x})$, linear in weights and activations, to define a (Boltzmann) probability:

$$P(\boldsymbol{x}) = \frac{\exp(-\mathcal{E}(\boldsymbol{x}))}{Z} \qquad \mathcal{E}(\boldsymbol{x}) = -\underbrace{\boldsymbol{x}^T \mathrm{U} \boldsymbol{x}}_{\text{pairwise interactions}} - \underbrace{\boldsymbol{b}^T \boldsymbol{x}}_{\text{bias term}}$$

where $\boldsymbol{x}$ encodes all the (binary) units, both visible and hidden. For clarity, we can *decompose* it and explicitly show both types of units:

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^T \mathrm{R} \boldsymbol{v} - \boldsymbol{v}^T \mathrm{W} \boldsymbol{h} - \boldsymbol{h}^T \mathrm{S} \boldsymbol{h} - \boldsymbol{b}^T \boldsymbol{v} - \boldsymbol{c}^T \boldsymbol{h}$$

where the matrices R, S account respectively for interactions between *visible* and *hidden* units across themselves. In addition W accounts for interactions between visible and hidden units. An example of how an undirected graphical model with weights (though not being a BM, since it is not fully-connected) is able to *encode probabilities* can be seen in Fig. 4.13.

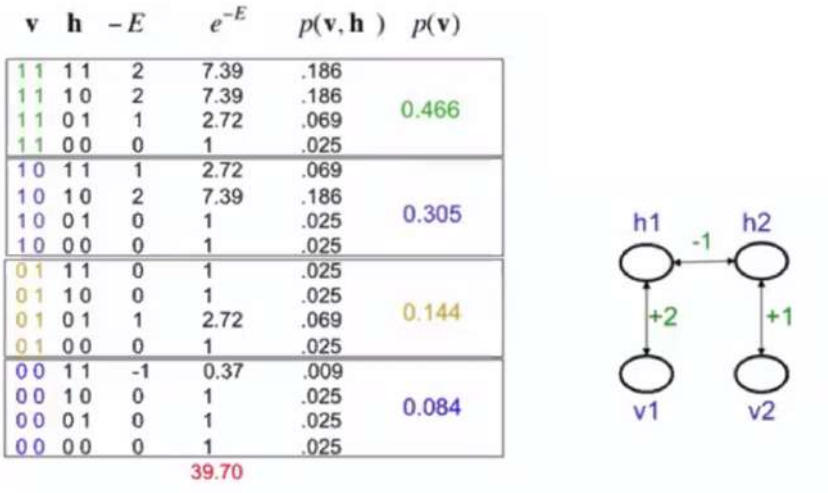


Figure (4.13) – The configurations that fully *respect* the relations encoded by the graph are the ones with lower energy (higher $-E$), and so the highest probability. For example, $\boldsymbol{v} = (1, 1)$ and $\boldsymbol{h} = (1, 1)$ has $-E = 2v_1h_1 + 1v_2h_2 - 1h_1h_2 = 2$. If one marginalizes, one obtains the probability for a given input to happen $p(\boldsymbol{v})$.

Let us study the **dynamics** of a Boltzmann machine, which is very similar to a *stochastic* Hopfield network. For **inference**, we choose to activate each neuron (binary unit) with a probability given by:

$$P(x_i = 1) = \frac{1}{1 + \exp\left(-\frac{1}{T} \sum_j w_{ij} x_j\right)}$$

If we have an observed input, and we want to *reconstruct the hidden values* (**data-driven** inference), we start with a random initial activation for the hidden units, and clamp that of the visible units on input data. Then, through Gibbs sampling, we iteratively update the activation values of *hidden* units one at a time, until equilibrium is reached. This is a very long process, also due to the fully connected topology.
An other possible approach is the so called **model-driven** sampling: we could use the model to *generate* also new inputs! The idea is to start with a random activation of *all* units, and then perform Gibbs sampling over *all* neurons, one at time, until equilibrium. Simulated annealing can be used to improve convergence towards *good* configurations. However, note that this procedure

is computationally very demanding. For this reason, Boltzmann machines are very rarely used.

Let us now understand how to theoretically model the **learning** process. We need to choose the weights so that observed samples have a high probability of being generated (i.e. low energy), and all the other ones have low probability, thus high energy (**maximum-likelihood** method). Since joint probabilities are *products* of probabilities, we can instead maximize the sum of the *log* probabilities. The gradient for that turns out to be quite simple:

$$\sum_{v \in \text{data}} \frac{\partial \log P(v)}{\partial w_{ij}} = \langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}}$$

The two contributions are, respectively, the expected value of the product of activation states (correlation) derived from the data distribution [14] (**"real" sensory perception**), whereas the second one is the expected value when the network is sampling state vectors from its equilibrium distribution and no units have been clamped to input data (**"fantasy" perception**). And then we just *tweak* the weights following the direction of this gradient, which maximizes the likelihood:

$$\Delta w = \eta \underbrace{(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}})}_{KL\ (P_{data} || P_{model})}$$

with η being the usual *learning rate*. The very last expression can be interpreted as the **Kullback-Leibler divergence** between the data distribution and the equilibrium distribution, over the visible variables produced by the *generative model*.

Restricted Boltzmann Machines

Fully-connected Boltzmann machines are very impractical due to the needed computational power: all units can influence each other. To simplify the problem we consider **restricted** Boltzmann machines, where no intra-layer connections are considered (i.e. no visible-visible links, nor hidden-hidden) is such a way we obtain a **bipartite graph**. The *energy* $\mathcal{E}(v, h)$ can be now decomposed as:

$$\mathcal{E}(v, h) = -b^T v - c^T h + h^T W v$$

The main computational advantage is that now *all* neurons of the same type (hidden/visible) can be sampled independently *at the same time* (it is possible now due to conditional independence, see Fig. 4.14), which is obviously more efficient (**block Gibbs sampling**).

[14] ∧ i.e. visible units are clamped to a training pattern

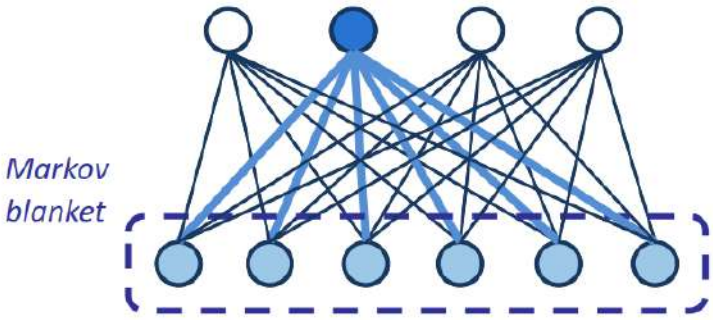


Figure (4.14) – The Markov blanket of a certain unit in the hidden/visible layer simply consists in the visible/hidden layer. Therefore, all the activations of units belonging to a unique layer can be sampled at once. (*block Gibbs sampling*)

Mathematically, this structure is equivalent to assuming that the hidden/visible conditional probabilities are separable:

$$P(\boldsymbol{h}|\boldsymbol{v}) = \prod_i P(h_i|\boldsymbol{v}) \qquad P(\boldsymbol{v}|\boldsymbol{h}) = \prod_i P(v_i|\boldsymbol{h})$$

In this case, maximum-likelihood learning can be approximated with the **contrastive divergence** algorithm, which is an *approximated* version of the ML learning by *biasing* the model driven phase.
As a first step: we observe some data in the visible units, then perform inference over hidden units which will take some values (**positive phase**, data-driven). Now, as a second step, rather than starting with hidden units initialized randomly, as the Gibbs sampling would require, we set them according to the *actual* configuration (inferred from data) and then sample data according to such configuration (**negative phase**, model-driven). This has to be done iteratively until we reach some equilibrium state, which usually happens quickly despite we started by having *constrained the model to the data*. Due to this "cheating", one can note that even after the *first* reconstruction results are really close to the ones we would obtain after *convergence* has occurred. Thus, we are allowed to use results from first reconstruction as a **proxy** for the *model* expectation values.

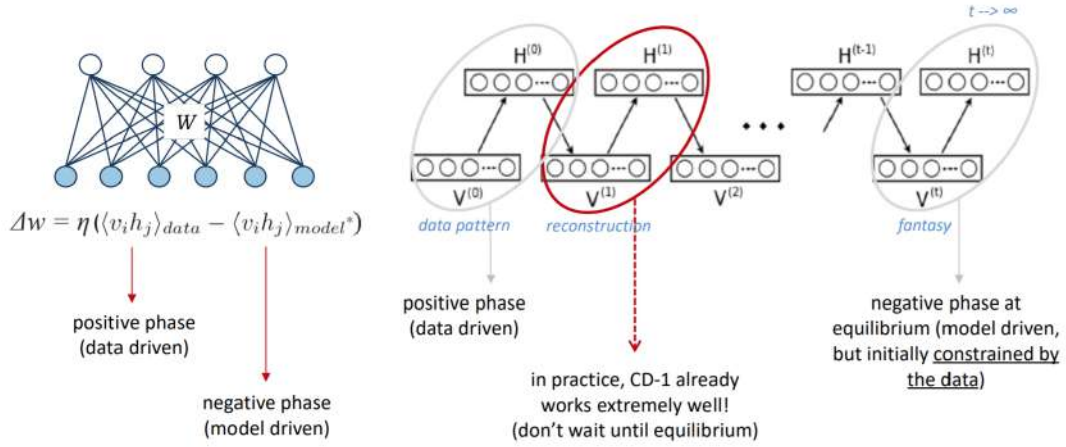


Figure (4.15) – The approximation consists of initializing the hidden units not at random, but from a *data-driven* configuration. In other words, we are *biasing* the sampling to achieve a faster convergence.

Another way to see this is that we are trying to approximate the following **Maximum-Likelihood** learning:

$$\Delta w = \eta(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}^*})$$

via the minimization of the difference of the following two KL divergences:

$$\boldsymbol{KL}(P_{data}||P_{model}) - \boldsymbol{KL}(P_{reconstruction}||P_{model})$$

since trying to optimize the following KL $\boldsymbol{KL}(P_{data}||P_{model})$ would lead to a higher computational complexity.

(Lesson 12 of 19/11/20) Compiled: September 20, 2021

Let us briefly describe how the Maximum Likelihood learning occurs via **Contrastive Divergence** (CD-1) algorithm [15]:

- **Positive phase**:
  1. The pattern is presented to the network and is clamped on visible neurons $\boldsymbol{v}$;
  2. The activations of hidden neurons $\boldsymbol{h}$ are computed in a *single* step via stochastic activation functions;
  3. The correlations $\langle v_i h_j \rangle$ between all visible and hidden units are computed, where for "correlations" we mean how units have co-activated when fed with a certain input;
- **Negative phase**:
  1. Starting from the hidden neurons activations computed previously during the *positive phase*, we generate activations (i.e. samples) on the visible layer using stochastic activations;
  2. Starting from these new visible activations (*reconstructed data*) we compute again the activations of the hidden neurons

[15] ∧ the "1" means to stop after a single reconstruction

3. We compute the correlation $\langle v_i h_j \rangle$ between all visible and hidden neurons are computed

- **Weights update**: according to the rule [16]:

$$\Delta w = \eta(\langle v_i h_j \rangle_{\text{data}} - \langle v_i h_j \rangle_{\text{model}^*})$$

Almost in all cases, the activations are *sigmoid*.
The basic idea behind *CD*-1 (see Fig. 4.16), is that the algorithm is lowering the energy of states which corresponds to training patterns, while increasing the energy of their *neighboring states*. However, differently from *Hopfield networks*, now we are able to explore also new states in the state-space, since we are trying to *reconstruct* input data thanks to hidden activations.

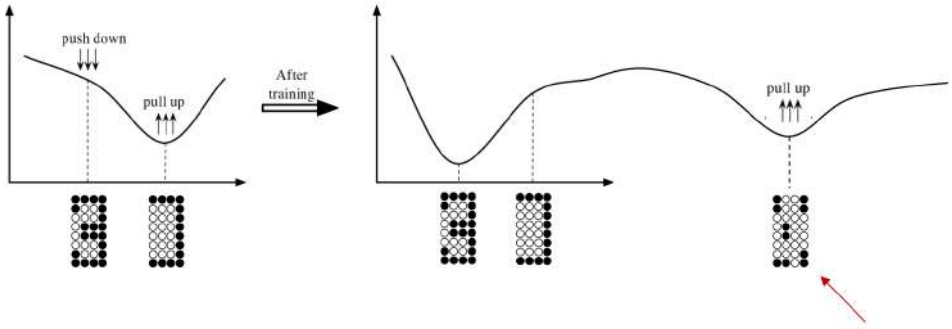


Figure (4.16) – Example of energy optimization related to $CD-1$ algorithm. We assume images/patterns to be encoded are digits. Generally, one wants to avoid spurious attractors that are low in energy, but do not correspond to any training data.

However, there may exist some configurations very different from the training data which are low in energy, thus being **spurious attractors**, and now we will see how to handle them. In order to "pull up" their energy, we need to perform more *CD* steps. Indeed, when starting with *small weights* and doing it only once (*CD*-1), we will have "optimized" only the neighborhood of training patterns by tuning weights accordingly: in such way that the energy in the immediate neighborhoods of training patterns is somehow raised. In order to handle and correct the energy landscape also in the "suburbs", we need first to be able to "visit" them. For doing so, more steps are needed (e.g., *CD*-3): the Markov Chain according to which the sampling is performed mixes quite slowly, so, to explore configurations far away from good minima, we must let the network generate samples for more iterations. After having reached them, we will proceed to update (i.e. grow) weights of the network to increase their energy, to make these states less probable. Then, once the weights have grown increasingly (and so are better and better tuned to draw a good energy landscape), we might want to increase the number of iterations, thus performing *CD*-10, *CD*-25 etc...

[16] $\wedge$ note as the second correlation term is on the "model*", and not on simply on "model". This is to take into account that now we are using some sort of trick to make converge occur faster, namely using already *biased*, and not random, parameters.

The idea underlying this approach is: when performing very *few* steps of *CD* we are able to approximate the immediate **neighborhood** of the *energy minimum* encoding the input pattern. However, the more iterations of *CD* we make, the more the system will tend to *distort* the input image, thus being able to reach states that are far (in the state space) from the input, but are less convenient in energy. There is no analytical formula to obtain the optimal number of iterations, but it is rather heuristic.
As for other *networks*, theoretically one can show the following **universal approximation properties**, and are specially related to *probability density estimation*:

- It can be proved that RBMs are universal approximators of *any* probability mass function over discrete variables.
- temporal (sequential) extensions of RBM are universal approximators of stochastic processes with finite-time dependencies.

Note as to implement these theoretical results might require a substantial number of parameters for the network, but this takes a back seat.
To understand how a RBM works in image processing, one has to take a look at Fig. 4.17. Note that as after training, hidden units specialize to detect edges at different orientations. These low-level features are visually very similar to the *basic ones* learned by the primary visual cortex of mammals, and can be obtained through different methods: sparse coding, ICA, and, as we have discovered now, RBM.

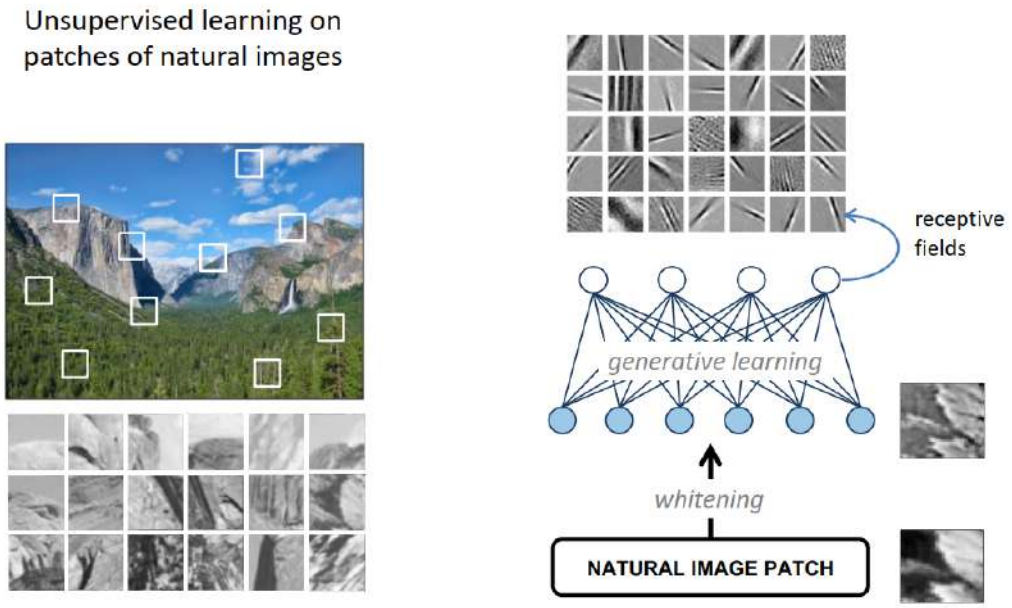


Figure (4.17) – Example RBM applied to image processing. Once some portions of the picture have been fed as input to the network (after preprocessing), the *receptive fields* will be edges at different orientations.

4.3 Deep Belief Networks

Let us proceed a little further and **stack layers** of representations, thus obtaining the so called *deep belief networks*. There are two possible ways of *how* to combine many simple (single-layer) blocks into a **multi-layer** model:

- **Hierarchical generative model**: thus building, as an example, deep Boltzmann machines or deep belief networks.
  However, **learning** is *greedy*, since every hidden layer is optimized independently, and *layer-wise*, in the sense that hidden layer are trained sequentially (starting from the bottom one and keeping the others fixed). This is a *disadvantage*, since once something has been encoded in a layer cannot be changed any more.

- **Hierarchical feed-forward model**: such as **V**ariational **A**uto**E**ncoders (VAE) and **G**enerative **A**dversarial **N**etworks (GAN).
  Here **learning** occurs as back-propagation method, thus including directionality and exploiting all the methods studied to optimize it.

Let us focus now on **Deep belief** networks. They basically consist of some *stacked RBM*s one after the other one, as one can see from Fig. 4.18. Consequently, the input data is clamped to the visible layers and the **training** of the **first** *hidden* layer is performed thanks to *contrastive divergence* algorithm, by encoding image features and then trying to reconstructing it. Later, one considers the **second** *hidden* layer, whose *input* being the previous layers, and performs again the training stacking together RBM. However, being a *greedy* architecture, it was found that after ~ 5 layers, the information starts to be lost.

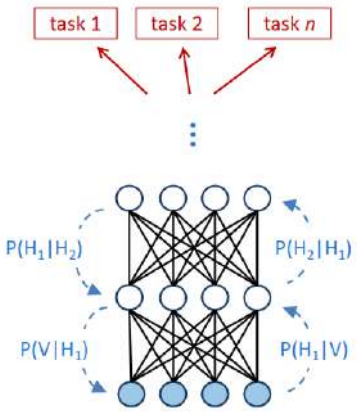


Figure (4.18) – Example of an architecture of a deep Boltzmann machine. The number of RBM-layers should be ~ 5, and eventually one could apply also supervised learning task such as classifier.

Using this approach, the *internal representations* (real-valued probabilities over visible units) of one *RBM* are used as input (visible units) for the *next* RBM. The result is that multiple levels of representations are encoded as a

hierarchical generative model (i.e. hypotheses over hypotheses), and every layer performing *non-linear* transformation of the data.
Finally, as **last** layer, *eventually* one can choose to perform **supervised-learning** at the *top-level representations*, thus implementing a decoder/classifier acting in the feature space.
Alternatively, a better approach would be to train a hierarchical generative model to discover abstract features, and at a second time perform a *fine-tuning* of the weights according to a **supervised loss** function. This proceeding of pretraining a hierarchical generative model solved back in 2006, the problem of vanishing gradient. However, it is not any more used due to the availability of more computational power, new activation functions, etc.
If one wants to inspect what features are learned by a deep belief network for different datasets *MNIST* (handwritten digits) and *Caltech*-101 (human faces), see Fig. 4.19. However, despite this learning is *totally* **unsupervised**, the network is still able to encode (i.e. learn) very abstract features, like the digits themselves, without any supervision. Moreover, even more surprisingly, there are even some units highly specialized.

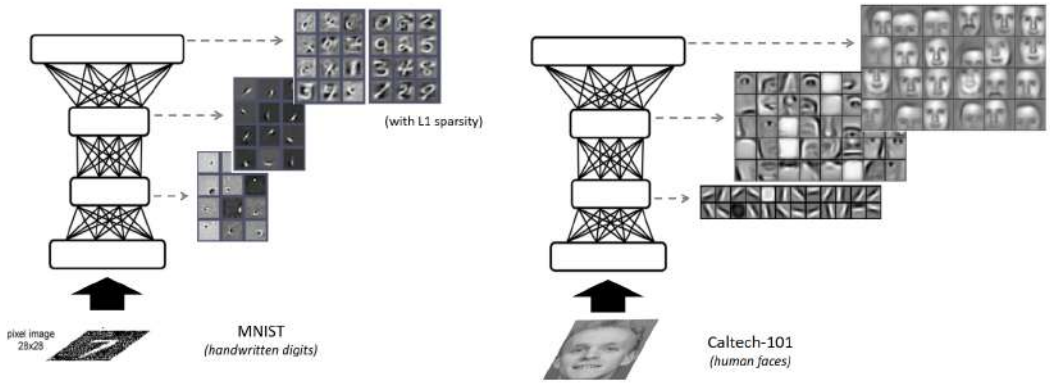


Figure (4.19) – Example of the features a hierarchical generative model can learn. Usually, the deeper we go, the more abstract (thus *global*) the features the network learns. Changing the type of data this general behavior does not vary at all.

Note as using these kinds of *generative models*, even in absence of inputs, they can be trained to **generate** *sensorial inputs* given their labels[17]. This is similar to Hopfield networks and their associative memory, and can be done since connections are *bidirectional*. Moreover, even more complex, one can train two different *Deep Boltzmann Machines* in a *multi-modal architecture* respectively on images and texts dataset, finally "joining" them with another hidden layer. In such way, it is possible to create a network that writes some description given visual inputs and, conversely, chooses the image that best fits to a certain description, thus exploiting some sort of **associative memory** able to perform information retrieval from the integration of *multi-sensorial inputs* as it is done by our brain.

[17] ∧ http://www.cs.toronto.edu/~hinton/adi/index.htm

4.4 Variational AutoEncoders

The *main difference* between an RBM and an AutoEncoder relies in the *directionality*: bi-directional the first one, while directed the second. Moreover, the first one **probabilistically** assigns low energy to "good" configurations of the system, thus finding its minima and allowing eventually the sampling of *new* data. Conversely, the **AutoEncoder** is a network that tries to **deterministically** reconstruct the input using a *feed-forward* mapping, and can be trained using back-propagation. Due to its nature, in the absence of input, the output will be null: it is not a generative model. However, one can start by randomly setting the activations of hidden units and check what is the corresponding reconstruction. Still, it is not guaranteed to be working especially for complex distributions.
This is why the need of some more powerful architecture has arisen: **Variational AutoEncoders** are today's state-of-the-art for hierarchical **unsupervised deep learning** models.
In standard AutoEncoders, the latent space can be *extremely* **irregular**: close points in the latent space (i.e., slightly changing activations) can produce very different, and often meaningless patterns over visible units. Therefore, we rarely implement **generative processes** that simply sample a vector from latent space, passing it through the decoder. However, a *possible solution* is to make that **mapping probabilistic** (see Fig. 4.20). The **encoder** now returns a **distribution over the latent space** instead of a *single* point. Moreover, the **loss** function incorporates an **additional regularisation term**, to set some constraints on the latent distribution (e.g. be multivariate Gaussian which is easy-to-handle with). The "variational" adjective for the AutoEncoder comes from this **variational inference** approach: we want to construct an *analytical approximation* to some probability distribution using a *simpler distribution* easy to work with.

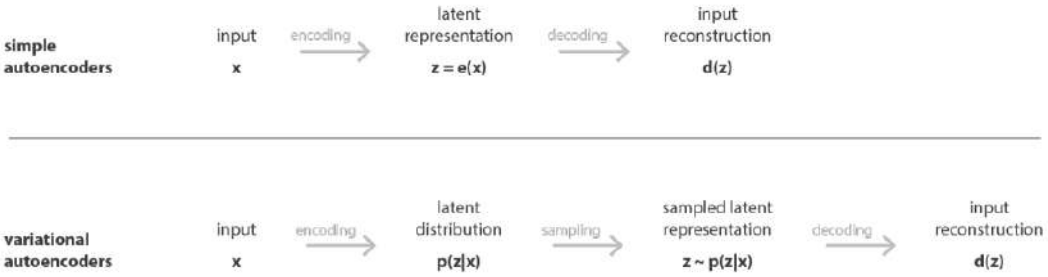
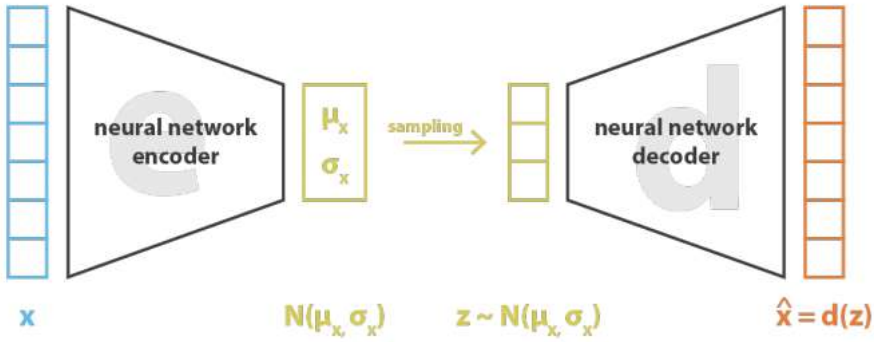


Figure (4.20) – In a *Variational AutoEncoder*, we encode the input to a *latent distribution* which we will be sample a *representation* from. Thus we will be able to *stochastically* reconstruct the input by decoding it.

One of the most popular *architecture* for Variational AutoEncoders is the one depicted in Fig. 4.21. We fix the target distribution as a **multivariate Gaussian**, so that the *encoder* is trained to produce a *vector* [18] with *means* and *covariance matrices* of the multivariate Gaussians. Once the vector has

[18] ∧ and not a *point* any more, as in the normal AutoEncoder.

been computed, one can **sample** a **representation** $z \sim \mathcal{N}(\vec{\mu_x}, \vec{\sigma_x})$. Note as the subscript x refers to a given *input* x. Finally, the *decoder* will generate the *output* $\hat{x} = d(\vec{z})$.
Using this approach one can **regularize** the **loss** function by forcing the *latent distribution* to be as close as possible to a standard Normal distribution or any other of our interest, however, one must pay attention to not sacrifice too much the reconstruction accuracy. This is a **trade-off**: the more attention is paid to reconstruction, the less the latent distribution might be approximated by a multivariate Gaussian. Conversely, the more we force the latent distribution to be close to a multivariate Gaussian, the more likely we are to decrease the reconstruction accuracy.



$$\text{loss} = ||x - \hat{x}||^2 + KL[N(\mu_x, \sigma_x), N(0, I)] = ||x - d(z)||^2 + KL[N(\mu_x, \sigma_x), N(0, I)]$$

Figure (4.21) – One of the most popular architectures for Variational AutoEncoders. The ***KL*** term in the loss refers to *how similar* the latent distribution, which we will be sampling from, is to a *multivariate Gaussian*. Ideally, in the *loss* at the end of the training both the reconstruction term and the ***KL*** one are null. Practically, it is a **trade-off**.

Practically, one can implement such an architecture (see Fig. 4.22) by considering that the latent representation is constituted by two vectors $\vec{\sigma_x} = h(x)$ and $\vec{\mu_x} = g(x)$, so the encoder network has two (possibly partially overlapping) branches [19]. In the most general case, the covariance matrix is square. However, to simplify our problem and reduce the computational complexity, we can assume **latent variables independence** (as in *RBM*s), thus making the *covariance matrix* to be **diagonal**. Later we must **sample** from this latent space, but one should keep in mind that *sampling* is a **discrete process**, therefore we are not allowed to use back-propagation of the error. So, we will not be able to understand in what measure the vectors $\vec{\mu}, \vec{\sigma}$ affected the reconstruction of $\hat{x} = d(\vec{z})$. Somehow, it is needed to reparametrize $\vec{z}$ to make it *differentiable*: this is why the *dummy* variable $\vec{\zeta} \sim \mathcal{N}(\vec{0}, \vec{1})$ comes handy. In such way that the parametrization of $\vec{z}$ becomes:

$$\vec{z} = h(x)\vec{\zeta} + g(x)$$

[19] $\wedge$ h and g are two (possibly partially overlapping) portions of the encoder neural network

And we will be able to back-propagate the error once more.

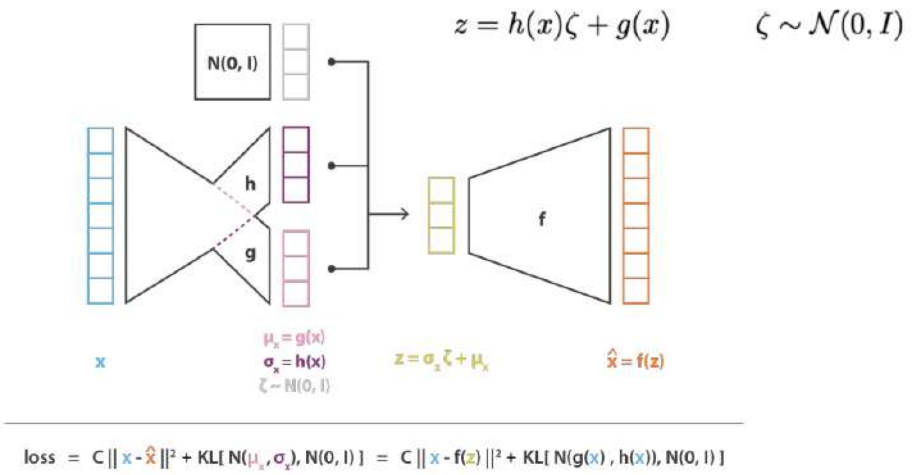


Figure (4.22) – A practical implementation of a Variational AutoEncoder.

An other important consequence of the *regularization term*, is that it makes the *latent space* **smooth**. It indeed promotes the *creation of a gradient* over the latent representations, which allows to generate samples varying smoothly! This implies that it is able to generate, as an example, faces with different orientations and emotions by respectively rotating it and making it more/less smiling in a "continuous" way [20]. In a standard AutoEncoder, however, "intermediate-featured" samples are most of the time not meaningful at all.

β-VAEs

The Variational Autoencoder can be further extended to promote learning of **more disentangled representations**, which in some cases might encode independent latent factors of variation in the data distribution. The final goal would be to have **single latent units** of z, sensitive to changes in single generative factors (e.g., hair color), while being relatively invariant to changes in other factors (e.g. skin color).
A possible way to produce it, despite it is still an *open* problem, by introducing a **penalization term** in the $\boldsymbol{KL}$-divergence using a hyperparameter $\beta > 1$ that balances latent channel capacity and independence constraints with *reconstruction accuracy*:

$$\mathcal{L}(\theta, \phi, \boldsymbol{x}, \boldsymbol{z}, \beta) = \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})] - \beta D_{KL}(\, q_\phi(\boldsymbol{z}|\boldsymbol{x}) || p(\boldsymbol{z})\,)$$

where the first term is the usual one related to *model accuracy*, while the second being related to *latent channel capacity* and *independence constraints*. The larger β, the more disentangled should be the representation.[21]

[20] ∧ a similar behavior can be reached using Boltzmann machines, but they are way less powerful in generating "smooth" data.

[21] ∧ for a better understanding, also on the loss function see https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html

4.5 Generative Adversarial Networks

(Lesson 13 of)
Compiled:
September 20, 2021

Generative **A**dversarial **N**etworks were introduced in the very last few years. They were implemented essentially to solve some *issues* arising when using **MSE**. Indeed, sometimes missing *few* input values might not result in a big loss, since *the majority* of the input values have been *reconstructed properly*. However, as one can see from Fig. 4.23, small changes are **critical**, since we might be losing the *semantic* of the data.

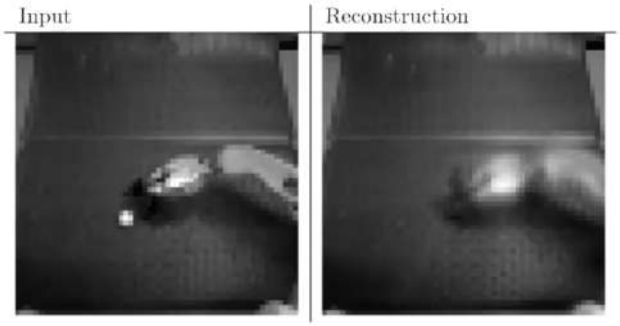


Figure (4.23) – In image reconstruction, if we "miss" the white spot the loss will not be so large (using *MSE*) since only few pixels have been changed. But, eventually, the information loss might have been significant! Here, for example, a robotic hand is trying to collect an object. If the latter one does not appear, the image loses significance.

In other words, we need a way to **specifically weight** *portions* of the input, thus defining their "saliency". In order to try to solve these *issues*, **adversarial games** come to help. In such games there are two *players* which compete against each other. The key idea is to build a good **generative model** of the data (*unsupervised learning*), whose task is to **fool a supervised classifier** (see Fig. 4.24). Thus, we are mixing unsupervised generation of data with its supervised classification: **any** single bit of information that is *discriminant* and *useful* for the classification task becomes **important**.

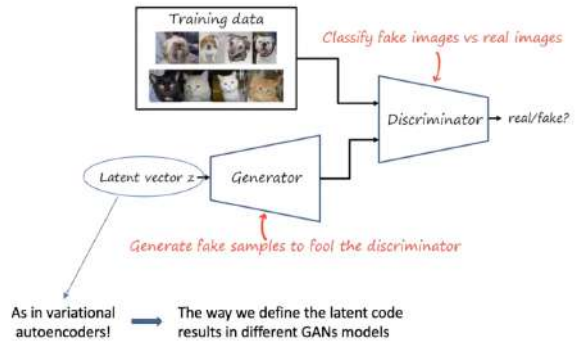


Figure (4.24) – In an adversarial games, two players compete against each other. The classifier tries to classify input information, that is *specifically* modified by the generative model in order to *fool* it. A discriminator must state whether an image is fake or not. Its input will be randomly drawn from either a training set (real), or from a generative model output (fake) created on purpose sampled according to some *latent vector* z. In this general setting, *Generator* (e.g. mixture of Gaussians, hidden Markov model, k-means) and *Discriminator* (e.g. SVM, perceptron) can be *any* algorithm.

Such a game should resemble the dynamics between *conterfeiters* vs *Police*, where the former ones try to gradually improve the quality as time passes to create fake money, with the latter one trying to spot it.
Let us now focus on *Generative Adversarial* **Networks** which have specific architectures: the one depicted in Fig. 4.25 where both *Generator* and *Discriminator* should be **deep networks**.

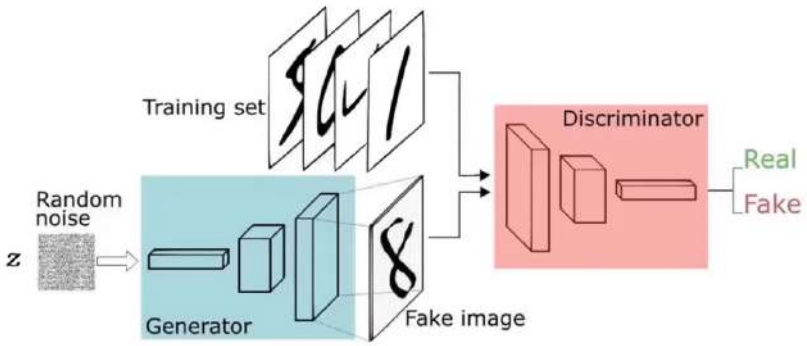


Figure (4.25) – Most general architecture for a Generative Adversarial Network.

The **Generator** can take as input a random pattern noise, thus generating an input through some stacked layers. Similarly, the **Discriminator** should be a *CNN* with arbitrary complexity, since we have been producing high-level images, thus producing a *binary* classification. Holding these assumptions, we can **train** the whole system using **back-propagation** in such way that:

- weights of **Generative** network are updated in order to *increase* the classification error
- weights of **Discriminative** network are updated in order to *decrease* the classification error

Studying the **learning dynamics** according to a *game-theoretical perspective*, we see as it is a **minimax (zero-sum) 2-player game**, with a value function that one agent seeks to *maximize*, while the other to *minimize*:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{obs}(x)}[\log D(\boldsymbol{x})] + \mathbb{E}_{z\sim p_{obs}(z)}[\log(1 - D(G(\boldsymbol{z})))]$$

where $D(x)$ represents the probability that x comes from the *true* data distribution rather than from the generator. D wants to maximize the probability of assigning the correct label to both x and $G(z)$, whereas G wants to minimize the probability of D assigning the correct label to $G(z)$. Since the **Discriminator** has to maximize the function above, its parameters are updated by **ascending** its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \Big[\log D(\boldsymbol{x}^{(i)}) + \log\Big(1 - D(G(\boldsymbol{z}^{(i)}))\Big)\Big]$$

Conversely, the **Generator** paramerers are updated by **descending** its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\Big(1 - D(G(\boldsymbol{z}^{(i)}))\Big)$$

There might be **problems** in **convergence** in such games. Indeed, they should terminate at a **saddle-point** (Nash Equilibrium state) that is a *minimum* with respect to one player's strategy and a *maximum* with respect to the other player's strategy. Applied to our problem: the **equilibrium state** is where Generator produces such high-quality data that is *indistinguishable* from the training distribution, and the Discriminator makes casual predictions with probability $1/2$ [22] This however requires a *careful synchronization* between learning in D and G to avoid instability and/or poor convergence (e.g. vanishing gradient if data generated by D is too easy to discriminate. This happens, for example, when G's parameters are randomly initialized). The ideal behavior is depicted in Fig. 4.26, and there are many algorithms for avoiding the problems mentioned above.

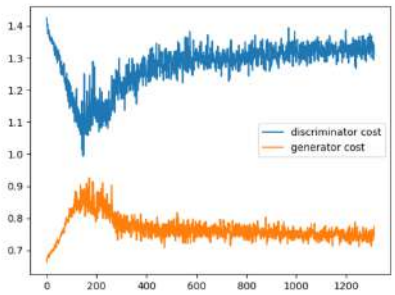


Figure (4.26) – Usual (and *ideal*) loss behaviors in GANs for the Discriminator and Generator. Initially, the Discriminator is able to tell whether some data is fake (initial behavior). However, as the Generator gradually improves itself, the task becomes harder for the Discriminator, thus leading to a different behavior.
If they are not synchronized, the discriminator cost will decrease too much thus flattening, not being able to generate a gradient to be minimized.

Another **important feature** of GANs is that, *after training*, it is simple and efficient to **sample from the generative model** using only *forward-propagation*, thus no need for approximate inference methods. Moreover, differently from VAEs, we do *not need* a *differentiable encoder* to create the latent representation z, allowing it to be *discrete* since there is no *encoder* as in VAEs.

If we want to visually inspect the different **results** obtained by *different* training algorithms, one should take a look at Fig. 4.27. As one can easily spot the best reconstruction is the one when using GANs. Indeed, using MSE the ear is not well formed: there are some little particulars which can suggest us the image to be *fake*. This occurs since the overall image is well-reconstructed except for some few (*informative*) pixels, so MSE is not that large. When one wants high quality and precision, GANs shall be used.

[22] $\wedge$ thus becoming a *random classifier*, which guesses with probability $1/2$ in the *binary* framework. This means that the Generator is winning.

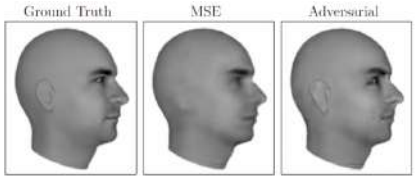


Figure (4.27) – Examples of successful trainings. GANs return better results, generally paying more attention to details.

However, there are still **open issues** in implementing a GAN. The most evident one is known as **Mode collapse**: when dealing with *bi-modal or "few"-modal* distributions and trying to generate data out of them, the GAN might end up approximating a single mode of the distribution (usually the simplest one to generate) while the others being neglected. This happens since focusing on a specific distribution is *enough* for winning the game: Classifier does not consider also the *complexity* of the target distribution, but only whether samples are true/fake. As an **example** we may want to be able to generate $[0-9]$ digits, but ending up only with generating high-quality samples of 1 and 7, whereas the remaining ones are totally neglected.
Another possible **issue** arising is that we are not immediately able to **learn internal representations** of data (i.e. encode them), but rather **generating data from them**. Indeed this architecture lacks of an encoder. The solution might be to look the direction in the latent space that makes the GAN sample as similar as possible to inputs we want to encode. Thus, inspecting the *latent space*, we might be able to understand the features that have been learned.
Let us mention some **advanced GAN architectures**, specially for image vision purposes. The way we can define the Generator latent code and the Discriminator task results in different GAN models:

- **CGAN** (Conditional GAN): we might want to include also some *class information* in the *latent space*, in order to allow generation of patterns from a specific class. As an example, using the *MNIST* dataset, one may want to generate a specific digits image, rather than sampling randomly from the distribution without any information on the actual class. The **class information** is given both to *Generator* and *Discriminator*, with the latter one trying to detect whether a sample is *actually* (real/fake) *given* it comes from a *certain* class (class information).

- **ACGAN** (Auxiliary Classifier GAN): we include the *class information* also in the **output space**. In this way, the *Discriminator* tries to detect whether a sample is *actually* (real/fake) **and** coming from a *certain* class (class information). In this way the Discriminator is challenged on a *finer-grained classification* task.

- **InfoGAN** (Information Maximizing GAN): we try to add **specific semantic meaning** to some variables in the *latent space*. This is achieved by adding a *regularizer* in the loss that maximizes the **mutual information** between latent variables and the *Generator distribution*. For example, when trying to generate *MNIST* digits one may want to include, besides

the digit class, also an *additional continuous variable* to represent factors of variation (e.g. angle, rotation, width, size...). Its idea is similar to the one of β-VAE, and both of them are considered state-of-art architecture. [23]

- **WGAN** (Wasserstein *GAN*): the *Discriminator* is replaced by a *"critic"* that, rather than just classifying, *scores* the realness or fakeness of a given image. The training becomes thus much stabler and faster
- **CycleGAN**: we can learn *image-to-image* translation without the need of paired training images. As an example, such architecture can change the texture of the skin of a horse to the one of a zebra, without actually containing it (i.e. the object "zebra") in the training set. Additionally, starting from a real-world image of a landscape, the CycleGAN is able to transform it to target painters' style.
- **StyleGAN** [24], [25] : the architecture of the *Generator* is significantly changed to allow for a more detail control of the generating process. A standalone "mapping network" is used to generate vectors of latent codes (i.e. styles) that are used at each level of the generator to produce the pattern.

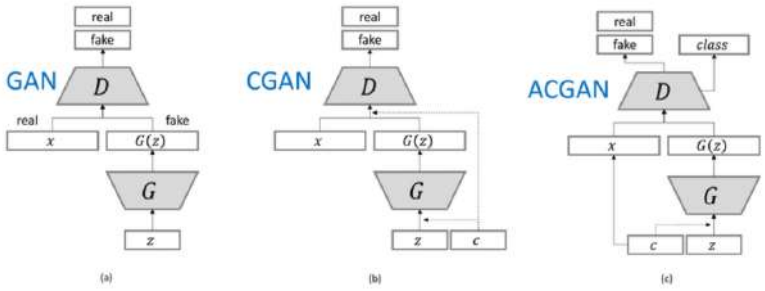


Figure (4.28) – Representations of more advanced *GAN* architectures: *CGAN* and *ACGAN*.

Finally, *GAN*s can be applied to improve video quality without increasing the traffic load during video meetings. Once a person's face has been learned (*keyframe*) the only relevant information will be its main traits (*keypoints*) movement. Thus, their trajectory will be the only data exchanged by the sender and the receivers, without the need to send the whole image. In such way the reconstructed image will have much more quality compared to what one would get using the same amount of bandwith, but continuously sending the video. [26]

[23] ∧ some faces, using also different factors of variation, generated by Info*GAN*s can be seen at https://www.youtube.com/watch?v=U2okTa0JGZg&ab_channel=JonaszPamu%C5%82a

[24] ∧ https://www.youtube.com/watch?v=kSLJriaOumA

[25] ∧ this architecture is such powerful that it can even create astonishingly quality portraits of non-existing people, which were found to be fake accounts' profile pictures in social networks. https://thispersondoesnotexist.com/
They can generate some artworks, too https://thisartworkdoesnotexist.com/

[26] ∧ https://www.youtube.com/watch?v=NqmMnjJ6GEg

CHAPTER 5

REINFORCEMENT LEARNING

(Lesson 14 of)
Compiled:
September 20, 2021

Reinforcement Learning [1] is a really powerful theory which has lot in common with *control theory* and *dynamical programming*. We will now study it, and finally bind it with **deep learning**. We will see as the latter can be exploited as a *block* encoded in a *Reinforcement Learning agent* to speed up and improve the quality of learning.
Generally, to try to understand **causality** it is *needed* to repeatedly perform *actions on the environment*: we need to distinguish between *correlation learning*, based on observations, and *causal learning*. Indeed the latter one requires a different and more complex approach. Correlation *does not* imply causation: it is *needed* to **actively manipulate** the environment to discover its *causal* structure. [2]

The *Reinforcement Learning* is strongly based on **animal conditioning** studies, and are mainly pursued by psychologists who try to model *animal learning*. In the **classical conditioning** paradigm, one tries to make the animal associate the *reward* with some *neutral stimuli*. The most classical example, i.e. *Pavlov dog* is described in Fig. 5.1, though the dog is still a **passive observer**, trying to detect *correlations* without performing any action.

[1] ∧ http://incompleteideas.net/book/RLbook2020.pdf

[2] ∧ some discussions about this, mainly, philosophical topic can be found in:
https://www.quantamagazine.org/to-build-truly-intelligent-machines-teach-them-cause-and-effect-20180515/
http://bayes.cs.ucla.edu/BOOK-2K/causality2-epilogue.pdf

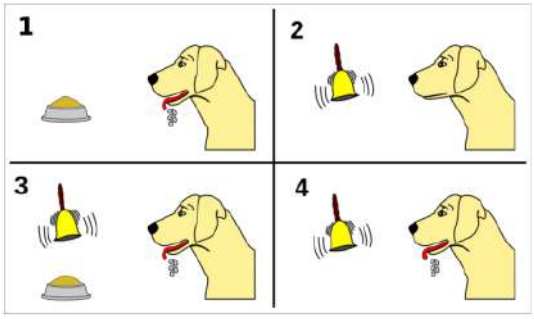


Figure (5.1) – The most paradigmatic example for associating neutral stimuli to animals reward based learning: the Pavlov's dog. A biologically potent stimulus (food, pleasure,...) is paired with a previously neutral stimulus (bell, light,...). After learning, the neutral stimulus comes to a elicit response (e.g. salivation) that is usually similar to the one elicited by the potent stimulus.

Let us now add some *more complexity*: thus introducing the concept of **Operant (or instrumental) conditioning**, where the animal needs to *select* an appropriate response to obtain the positive reward or avoid punishment. In such framework the animal must perform the correct action and actively interact with environment to get the *positive reward*. However, if also a *negative reward* is introduced, the animal learns even more carefully what must be done. Some attention must be paid in order to model the experiment in such way that negative rewards have not to be too tough, or else learning might not occur due to high stress level. This is really similar to **complex reinforcement learning settings**: one wants to act such that *rewards* are *maximized*, conversely *punishments* are *minimized*.

We are now *learning* from **trial and error**. In such framework, the fundamental problem is to understand *what actions* to be chosen in order to *maximize* one's *future rewards*. However, some **complications** might arise:

- Some actions may maximize **immediate reward**, but will turn out to be *counterproductive* in the long term. Conversely, some actions might seem useless, but will turn out to be important in the future. A possible **solution** is to learn to predict *long-term outcome* of actions, although it is a really a challenging task.

- **Environment** is usually **stochastic**: it is not sure that a certain action is always appropriate or leads to the same outcome. A possible **solution** is to use a *stochastic action policy*, where actions are sampled according to some probability.

- **Exploration** *vs.* **Exploitation** *dilemma*. one should understand whether it is better to explore new states, eventually leading to higher but eventually also to worse rewards, or stick to something it is known to be (sufficiently) safe enough. A possible **solution** is to use an *annealing-like* scheme, to encourage *early exploration*, finally settling more and more to *most rewarding* policies. Biologically, this role is actually carried out by experience.

The Markov Decision Process

The most general setup for a RL problem is the one depicted in Fig. 5.2, where essentially one has an **agent** $\mathcal{A}$ and the **environment** $\mathcal{E}$ which interact through a *loop*. The general setting for such a process is:

- $S = \{s_1, s_2, \ldots, s_n\}$ the set of **environment states**. The simplest case is the *binary* one, while in most complicated problems it might be *high-dimensional* of *continuous* values.
- $A = \{a_1, a_2, \ldots, a_n\}$ the set of possible **actions** that can be performed at every timestep. It can be *discrete* or *continuous*.
- $O = \{o_1, o_2, \ldots, o_n\}$ the set of **environment observations**, that is to say what one can perceive from the environment and is the result of the combination between the *state* and the *reward*.
- $T(s_{t+1}|s_t, a_t)$ the **rules** for **transition** between states, conditioned on the last previous state and action.
- $R(r_{t+1}|s_t, a_t)$ the **rules** that determine the **immediate reward** $(+r)$**/punishment** $(-r)$ associated to a *transition*.

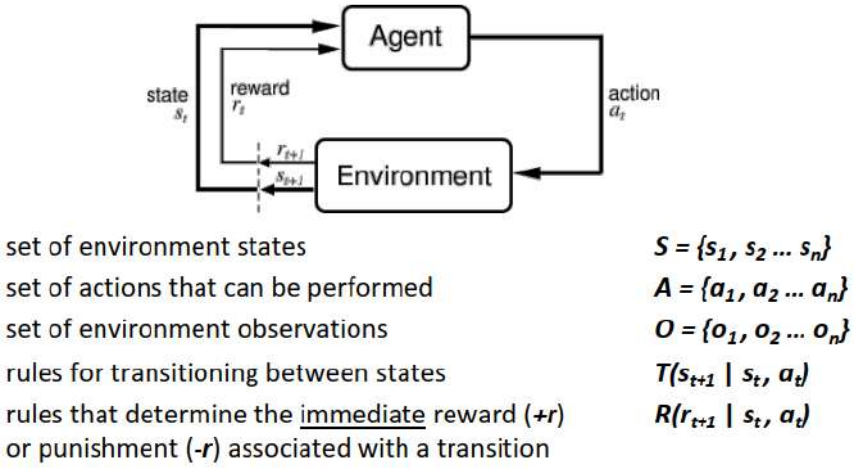


Figure (5.2) – The Markov Decision Process loop.

At every timestep t, the **agent** receives a new observation o_t which consists in the current state s_t and a *reward value* r_t. It then chooses an action a_t from the set of actions available, which causes a transition on the next state s_{t+1} in the **environment**, which is associated with a certain *reward*. The **goal** the *agent* wants to achieve is to **maximize the accumulated rewards** [3]. Such systems embed the **Markov property**: knowing the current state, the next one will be surely defined.

[3] $\wedge$ if compared to *supervised* and *unsupervised* learning goals: in the former the *classification error* had to be minimized, where in the latter one wanted to make *reconstruction* error as low as possible.

Let us now try to understand how to *practically* **maximize accumulated rewards**, also known as the "return". Introducing the *sum of accumulated rewards as*:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

We might want to proceed in two ways:

- **Finite-horizon case**: we aim at summing rewards up to a certain point in time;
- **Infinite-horizon case**: we need to introduce a **discount rate** $\gamma \in (0, 1]$ which quantifies how much we should care about *future rewards*, because we aim at maximizing rewards for the entire "agent's life". It tries to maximize the *discounted long-term expected return* defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

However, the last framework comes with the **"credit assignment" problem**: the presence of *delayed rewards* makes learning more complex. It is hard to link *actual rewards* with the *past actions* that have led to it, therefore assigning to each of them a specific "contribution" for the final reward. This can be solved, as we will see, by introducing **intermediate rewards** leading to a *finer-grained* structure.

In order to solve the *reward maximization* problem, we need to introduce the concepts of **policies** and **value functions**. We define the **value function** of a state the *expected cumulative return* from that state:

$$v(s) = \mathbb{E}[G_t | S_t = s] \tag{5.1}$$

Note as the "expectation value" operator takes into account the *stochasticity* of the environment. Computationally, one can recursively *unpack* the *long-term* reward to get the **Bellman equations**:

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] \tag{5.2}$$

In such a way, we can decompose the value function of a state as the sum of the *present* reward and the expected *future* reward, weighted by the discount factor. Both (5.1,5.2) require *full knowledge* of the **policy** π of the agent, which specifies the *actions* it will choose *for every* state:

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s] \tag{5.3}$$

Combining all three previous equations (5.1,5.2,5.3) one obtains the **Bellman equation**:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \quad \forall s \in S$$

Dynamic programming allows to solve the last equation directly. However, this problem comes with an *exponential complexity in S*: finding the optimal

policy might be computationally infeasible for large state spaces S. Moreover, the values of $\mathbb{P}(s'|s,a)$ and $R(s,a)$ need to be perfectly *declared in advance*. Hence, dynamic programming is theoretically a powerful tool, but cannot scale to real world problems and that is why *Reinforcement Learning* and *Deep Reinforcement Learning* come useful.

Temporal Difference learning (TD)

However, another possible *algorithm* to estimate the value function is the so called **Temporal Difference learning** (TD): we learn an approximation of the value function by exploiting a prediction error signal between two consecutive states:

$$V(S_t) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right] = \mathbb{E}[r_t + \gamma V(S_{t+1})] \simeq \mathbb{E}[r_t] + \gamma \hat{V}(S_{t+1})$$

After transitioning to the *next state*, the predicted value function is updated to bring it closer to the *true* value:

$$\hat{V}(S_t) \leftarrow \hat{V}(S_t) + \eta\delta(t)$$

where η is as usual the *learning rate* and $\delta(t)$ is the *prediction error* which is defined as:

$$\delta(t) = \underbrace{r_t + \gamma \hat{V}(S_{t+1})}_{\text{better estimate}} - \underbrace{\hat{V}(S_t)}_{\text{current estimate}}$$

and in the *ideal* case $\delta(t) = 0$, since the guess about the *present* and *expected future* rewards has been totally correct. In this approach, we are trying to tune the "quality" of each state, by *updating* it using repeated observations in order to find the *best trajectory* in state space for maximizing the reward.

Q-learning

An even more power way to solve this kind of *error-driven* learning is the **Q-learning**. Differently from TD-learning, here one tries to associate a value Q to every **state-action pair**:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s,\ A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s,\ A_t = a\right]$$

By relating actions to states, Q-learning allows to consider the policy (e.g., greedy policy *max*, etc.) used for *predicting future* rewards. The **updating** rule for **Q-values** is the following:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}}\right)$$

Figure (5.3)

where we usually start from a *default* estimate of the value function for each state-action pair, and then refine it over time. Note as the *update rule* is very similar to the one of TD-learning, but now also accounts for the *policy* [4]. This approach is **model-free**: we do not need any explicit knowledge about the environment: we simply use the Q-value of the next state as an estimate of future values (*bootstrap*). **Convergence** to the *correct Q-value* and thus to the *optimal policy* under some hypotheses over the learning rate α [5] is **theoretically guaranteed**, but not necessarily practical due to the limited amount of time since it might take an order of years. Due to these issues and to the considerable amount of sampling needed, Q-learning was rarely used.

To provide some hints on how to tackle the **Exploration** *vs.* **Exploitation dilemma**, in order make the *exploration* of the environment *more efficient*, usually actions are chosen **probabilistically**, according to some policies. We will briefly mention a couple of them (see Fig. 5.4), namely:

- **ε-greedy policy**: we define a relatively low probability $\epsilon \sim 0.15$ according to which we may select a *random* action. Usually, the value of ϵ decays over time, to favor exploration only at early stages.
- **Softmax policy**: actions to perform are sampled according to their relative *Q-value* distribution. Most common actions will be the ones with the largest *Q-value*, but rarely also other ones will be performed. Stochasticity might be increased as the temperature-like parameter τ becomes larger.

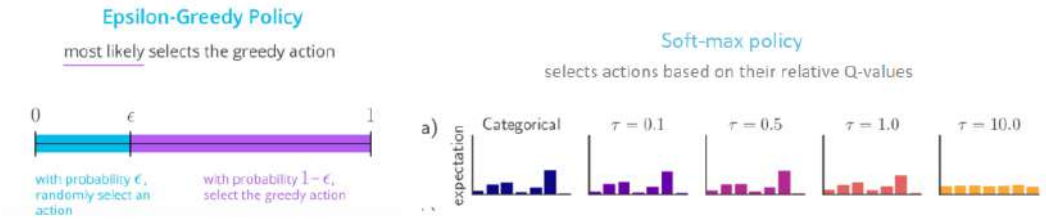


Figure (5.4) – Different policies to stochastically implement a better exploration of the environment.

Exploration is crucial, specially in the *initial phases*: the agent needs to find out as much as it can from the environment. If agent is *too greedy* during first episodes, it might get stuck in a *local* maximum. So, a possible trade-off for the dilemma is to *explore* at the beginning, while *exploiting* later on.

On/Off-policy

An other important distinction is **On-policy** and **Off-policy** methods. In the *Q-learning* example, we were assuming that **update policy** was greedy (i.e. we choose the max): the action with the highest expected value will *always* be chosen in *future states*. However, the agent has also a **behavior policy**,

[4] ∧ the "max" in $\max_a Q(s_{t+1}, a)$ term. Eventually others can be chosen.
[5] ∧ the sum diverges, but the sum of squares converges.

which is actually used to select the next **action**. If the two policies coincide, the *learning algorithm* is said to be **on-policy**.
An example of *on-policy* algorithm is **S**tate-**A**ction-**R**eward-**S**tate-**A**ction (*SARSA*), which uses a stochastic behavior policy also to update its estimates. Let us briefly discuss why some other algorithms, in addition to Q-learning, are needed: its main problem is that it has a flaw that can reduce the accuracy of the Q-values that it relies on. Indeed, using Q-learning we are basing our future reward on the score of the most likely next action, even though that is not necessarily the action that would be taken. In other words: the *update rule* assumes we are going to pick the highest-scoring action on our next move, and its calculations of the new Q-value are based on that assumption. This is not an unreasonable assumption, because both our ε-greedy and $softmax$ policies *usually* pick the most rewarding action. But the assumption is wrong when one of those policies chooses one of the other actions, that might happen with some probability different from zero. When our policy picks any action other than the one we used in the update rule, the calculation will have used the wrong data, and we end up with reduced accuracy in the new value that we compute for that action. **SARSA** has been introduced exactly to tackle this issue: it fixes the problem of choosing the wrong action from the next state by choosing *that* action in advance coherently *with our policy* [6], and finally *remembering* the choice of action [7]. Then, when it is time to make our next move, we will simply select the action previously computed and saved. In other words, we have moved the time when we apply our *action* policy: instead of choosing our action at the start of a move, we choose it during the previous move and remember our choice. That lets us use the value of the action we really will use when building the new Q-value: actions are chosen (*policy* rule) according to the same principle we were updating the Q-values (*update* rule). Since both "share" this sort of stochasticity, this algorithm is classified as **on-policy**.
Conversely, *Q-learning* is an *off-policy* algorithm: it uses a *stochastic behavior* to improve exploration therefore choosing even not convenient actions, indeed we might use either ϵ-greedy or softmax, but a *greedy update policy* (deterministic) to update the Q-values as if we were surely choosing the best move. The main difference between the two types of learning is that *off*-policy algorithms usually take more risks: they assume they will **not make stochastic mistakes** during the next steps. An example can be seen in Fig. 5.5

[6] $\wedge$ rather than just simply selecting the biggest one
[7] $\wedge$ this is where the last "A" comes from

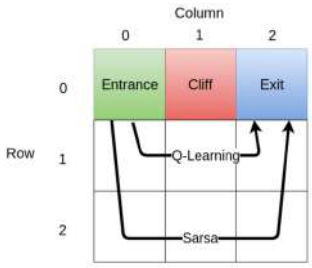


Figure (5.5) – The main goal of this game is to reach the *exit* cell, once having started from *entrance* and by moving only up/down/left/right. In addition, *cliff* cell must be avoided. *Q-learning* takes more risk and passes nearby the cliff and reaches the exit in less steps, since it knows for sure it will not take bad actions. Whereas *SARSA* has a more conservative approach since stochastic oscillations might occur, thus taking into account them.

One of the biggest problems of this theory is the so called **curse of dimensionality**. Complex tasks can have **huge** state spaces. Every time a dimension is added to the problem, the size of state space increases exponentially. As an estimation of order of magnitude: if we are dealing with a 20×20 8-bit greyscale image, the agent has 2^{1200} possible states. This was the main reason why *reinforcement learning* was not used until some years ago. Moreover, one should note that **learning** is an **incremental process**, and is not performed in two separate phases as it was in *supervised* and *unsupervised* learning algorithms.

5.1 Deep Reinforcement Learning

The basic idea underlying the *Deep Reinforcement Learning* is to use a **deep** network to **approximate the** Q **value function**. Rather than maintaining a table with estimate values, we use a function parametrized by the *connection weights* of a neural network, thus leading to a **better generalization**: similar states sharing large information return similar outputs.
More recently, deep learning has been used to directly **estimate** the **action policy** (*policy gradient methods*), in addition to the value function. In such way, training occurs using *backpropagation* over both the *reward* and the *action* sampled from the neural network.

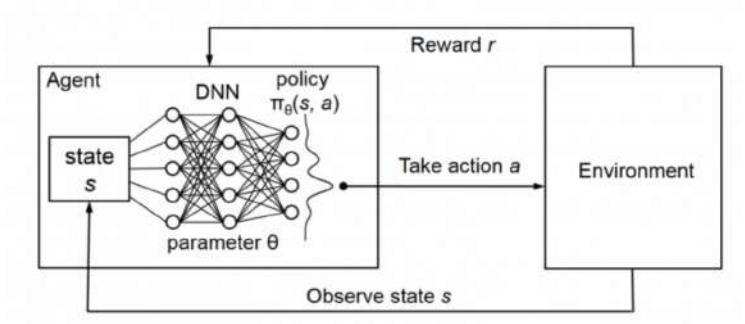


Figure (5.6) – Deep Reinforcement Learning architecture. The main task of the deep network is to better approximate the value function. However, recently, it has started to make it learn the action policy, too.

Let us focus now on the **issues** arising when performing *Q-learning* with *Neural networks*. The main problem of using them as *Q-value* estimators is **instability**. Initially the *training set* is being created **incrementally** and the agent needs a lot of time to have a good knowledge about the environment. Moreover the elements of training set are **highly correlated** and not i.i.d., which we know to be the best inputs for neural networks. Finally, the *target function* is **non-stationary**: it is constantly being updated and changing as the learning proceeds with the agent that keeps on interacting with the environment.
Two possible **workarounds are**:

- Use a **target network**, thus having two *separate* networks: a **prediction** one being trained (i.e. change weights) *every step* to predict rewards given the actual parameters, and a **target** one, used for action selection. The latter one is *periodically updated* to the update network's values (see Fig. 5.7). In this way, there are some time windows in which the network behaves coherently, i.e., keeps the same policies.

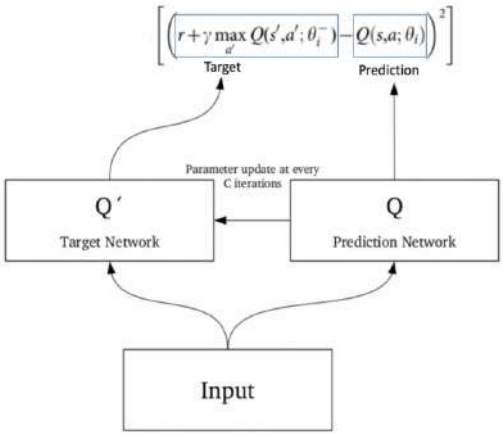


Figure (5.7) – There are two networks: one for predictions which is updated at every timestep, one related to policies and being updated every C iterations.

- **Experience replay**: use a *memory buffer* to store *previous learning episodes*, from which batches of state-action-reward-state tuples $(S_t, A_t, R_{t+1}, S_{t+1})$ are randomly sampled. In this way we can **decorrelate** the samples, thus facilitating the network learning process.

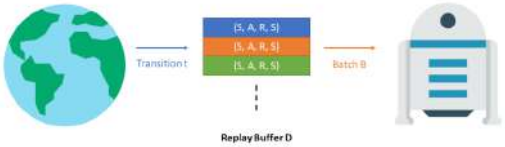


Figure (5.8) – Note as replay memory can be prioritized to show more important transitions more often.

The first working example of *Deep Q Network* (DQN) was developed in 2015 by Google DeepMind. It managed to outperform humans in several challenging

Atari games. Its *input* consisted of a screen state with memory of 4 past frames, and the reward. Conversely, the *output* was action probabilities. Surprisingly, after a lot of training, the agent shown to be able to discover *new strategies* by itself.
An improvement to the DQN architecture was later able to beat the human world champions in challenging games such as Chess or Go (AlphaGo, AlphaGo Zero). The latter was an astonishing result, due to the enormous state space.

5.2 More advanced topics

For solving such **complex problems**, RL agents might require an incredible amount of *learning experience* ("episodes") and *computational resources*. In addition to the results achieved by a model we should also look at the **learning speed**, that is to say to train a *sample-efficient RL*.

Another complication that might arise is that in many scenario **the action space is continuous** (e.g., physical control tasks for robotics): indeed we cannot simply sample actions from a limited *discrete* state space. Moreover, to maximize the state-action values (as in Q-learning), a *greedy-max* policy should need to run an optimization process at *every* step, which is clearly unfeasible. Some possible solutions are:

- **REINFORCE**
- **Actor-Critic architectures**
- **Trust Region Policy Optimization, Proximal Policy Optimization**

The basic idea behind all these methods is to represent the policy using a *dedicated function* **approximator** independent of the value function:

$$\pi(a|s, \boldsymbol{\theta}) = \mathbb{P}[A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}]$$

Note that the value function may still be used to *learn* the policy parameter, but is not required for action selection. The *policy* parameters are learned by *ascending the gradient* of some scalar performance measure $J(\boldsymbol{\theta}_t)$, whose **update rule** is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t)$$

where α is the usual *learning-rate*, and the **performance measure** takes a different form from the one we had in classification/reconstruction tasks. In this case $J(\boldsymbol{\theta}_t)$ is usually estimated by *sampling possible trajectories* for the agent ("roll-outs") that contain sequences of states and actions: in other words we would need to *guess* the possible outcomes.
An **advantage** of such methods is that we are optimizing *continuous policy parametrizations*, therefore *action probabilities* change *smoothly* as function of the learned parameters. Thus, from theory we know that **stronger convergence guarantees** are available for *policy-gradient* methods, rather than action-value ones.

However, the main **issue** is that the *performance function* $J(\boldsymbol{\theta}_t)$ depends both on *action selection* and on the *distribution of states explored* as the result of such actions. In other words, the *policy parameters* influence both quantities: given the state, it is straightforward to measure the direct effect of θ on actions, *but* the effect of parameters on the *state distribution* depends also on the *environment*, which is external and might be stochastic. However, an *important theorem* about these methods, thus providing some hints on how to tackle the last issue, is the **Policy gradient theorem**. It states that we can *analytically* derive the gradient for the policy parameters in a way that *does not* consider the state distribution.

A common architecture for policy gradient methods that can learn *approximation* to both *policy* and *value* functions is the **Actor-Critic**. The *Actor* is a reference to the learned policy, whereas *Critic* refers to the learned value function. In other words: the policy estimator is called **actor** and it decides what actions should be performed. Conversely the value function estimator is known as **critic** and its task is to estimate the best values, thus updating the policies using Temporal Difference error based on the reward provided by the environment. Knowing the value function can assist the policy update, such as by reducing the gradient variance.

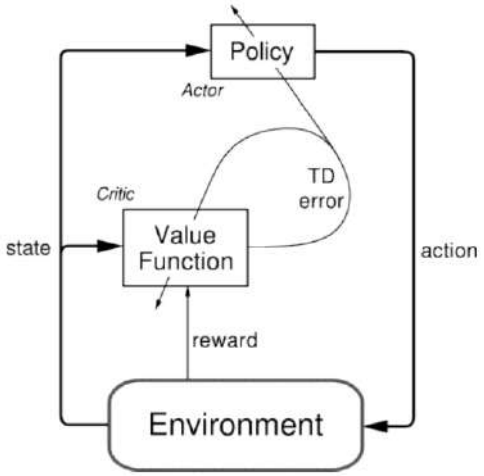


Figure (5.9) – Sketch of Actor-Critic architecture in Reinforcement Learning. It is visually similar to Markov Decision processes formulation, but policy and value functions are decoupled.

Such architecture pattern was exploited by the **Asynchronous Actor-Critic** (A3C), which allowed for efficient parallel training. The Critic learns the value function, while *multiple Actors* ($\sim 100, 1000$) are trained *in parallel* and get synced with global parameters from time to time. Its implementation it was shown that *surpasses* the current state-of-the-art on the Atari domain, while training for *half* of the time on a single multi-core CPU instead of a GPU. Indeed the **training** can be sped up, for example, by having multiple *actors* that simulate multiple trajectories in parallel. However, currently, Reinforcement Learning is far from being able to efficiently implement complex robots.

Semi-Supervised learning

There is another issue, especially evident in Atari domain, that is known as **high-level knowledge**. Deep RL can learn sophisticated control strategies, however, it often miserably fails even in *simple* scenarios but that require *high-level* knowledge, which is provided to humans by their experience and it is what machines initially lack of. As an example, in *Montezuma Revenge*, a character needs to escape a room by exiting a door. However, before being able to do it, he must obviously collect the key: this is some high-level knowledge a human surely has, but is complicated for a neural network to realize it having no *a priori* knowledge.

This problem can be tackled by **semi-supervised RL**: we should avoid sparse rewards, and **progressively guide the agent through the environment**. This can be done by assigning rewards to *intermediate step*: in the example above, a reward shall be assigned when the character collects the key, and a second one when the room has been exited. Another example: if one needs to reach a certain position to get the reward, moreover, being subject to some constraints, usually the agent is made to start closer to the final goal and progressively moves backwards by perturbing the system. Or else it would have been impossible due to the environmental complexity.
Some possible approaches for *semi-supervised RL* can be:

- **Shaping**: it is borrowed from animal training, when rewards are assigned after performing gradually more complex tasks. [8]
- **Transfer and curriculum learning** (e.g., with auxiliary subtasks). Regarding the previous example: one might take a pretrained agent in collecting keys and put it in a more complex and original framework, thus exploiting its previous knowledge.
- **Imitation learning**: especially used in robotics, the agent updates its policies according to the imitation of some supervisor's behavior, and gradually it is made independent.
- **Model-based RL**: the agent tries to learn a **internal model** of the environment, useful for predicting its dynamics (*planning*, namely predict the new state s_{t+1} given the current state s_t and action a_t). This allows to reduce the number of interactions with the real environment during the learning phase: the aim is to construct a model based on these interactions, and then use this internal model to "simulate" further episodes (see Fig. 5.10). An interesting implementation is the one proposed in Fig. 5.11, where given some input images of a 3D scene taken from different viewpoints, the goal was to predict the scene appearance from *unobserved* arbitrary viewpoints. After having reached such a goal, one is able to more efficiently move through the arena since the architecture has learned the internal representation of such environments.

[8] ^ https://www.youtube.com/watch?v=N2bt95xud8w&ab_channel=AndrewPeterson

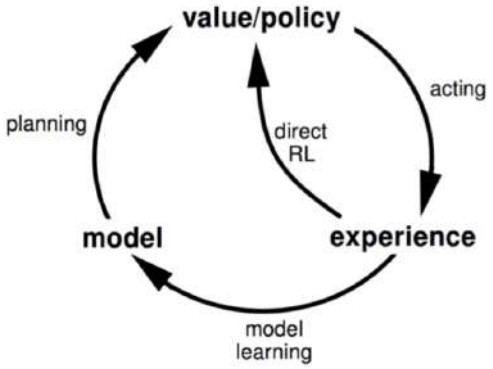


Figure (5.10) – Model-based RL algorithm description. It is similar to "policy rollouts" or Monte Carlo Tree Search, though here actions are not selected randomly but chosen according to some internal model which has been internally represented.

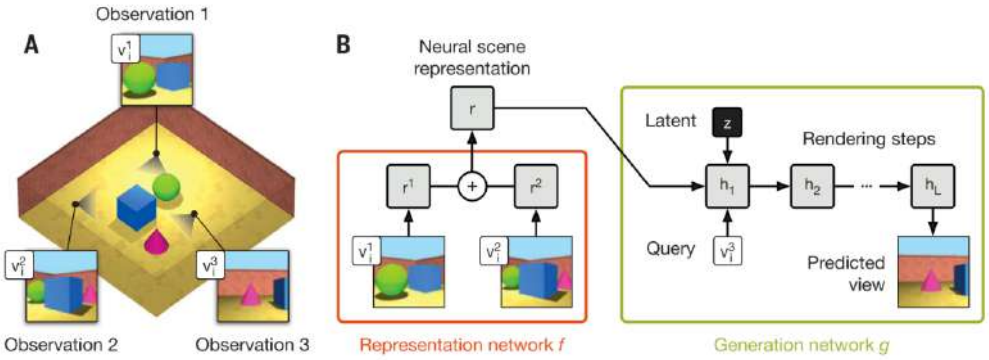


Figure (5.11) – Neural scene representation and rendering, an example of Model-based RL algorithm.

Curiosity-driven RL

Yet another recent idea in Reinforcement Learning is the **Curiosity-driven RL**. It is rooted in the *psychological theory* of "intrinsic vs. extrinsic motivation", that is to say:

- **Extrinsic motivation**: when one's behavior is motivated by an *external factor*, pushing it to do something in hopes of earning a reward or avoiding punishment.
- **Intrinsic motivation**: when one's behavior is motivated by its *internal desire* to do something for its own sake (i.e., personal enjoyment or desire to learn a skill).

Thus, being inspired by this idea, people are now trying to build a *reward function* that is **intrinsic to the agent** (i.e. generated by the agent/environment itself) rather than defining an *external reward function* given by the task. The aim is to **gain knowledge about environment dynamics** before trying to solve a problem.
An *example* of **intrinsic reward** might be: an *error* between the predicted new state s_{t+1}, given our state s_t and action a_t, and the real new state. The reward

will be small for familiar (and easy to predict) states, thus pushing the agent to seek states where it has spent less time (or with more complex dynamics), finally exploring better and more in depth the environment. Similarly to model-based RL, the agent learns an internal model of environmental dynamics, but differently from before, now there is *no explicit task* to be performed. Recently, it was shown that **without any extrinsic reward** a curiosity-driven agent was actually performing really good in Atari domain, and Super Mario Bros too. [9] The model was trying to explore new pixel-spaces never explored previously, just for the sake of curiosity, and without the task of completing levels. However, a *side-effect* is that an exploring agent gets stuck if we place a TV screen in the environment and let it play with it doing zapping (pixels keep changing).

Multi-agent RL

In most real-world scenarios, the environment includes *multiple agents* (team games, etc.) which *actively change* their policy. This setup is tackled by **Multi-Agent RL** (*MARL*) framework. Obviously, the **learning complexity is greatly increased**: environment is no more stationary since multiple agents can change it at the same time. Hence, Markov assumption is *violated*. Moreover, the **environment** is **no more fully observable** since other agents may behave in different way, according to their personalities. Indeed they can be *Cooperative* if they try to collaborate thus maximizing either a common reward function, or an individual but *compatible* one. Moreover, they can be *Competitive*, if reward functions are in contrast with respect to other players, or finally they can be *Irrelevant*, thus being simply a source of stochasticity.
In addition, **Learning** be *Centralized* if all agents share the same policy (e.g., policy of neural network is shared among nodes), or *Decentralized* if each agent learns by itself, which is useful to learn *team roles*. If the policy is shared among many players, it is important to have a good *communication*, in such way that agents learn to *coordinate*. This can be achieved by **learning** how to *communicate* and/or **understand** other's intentions.
All these scenarios are usually framed as **Game Theory** problems, but up to now successful tasks are still quite easy and simplified.

Incremental Learning

When *acquiring new knowledge* (e.g., the data distribution changes), neural networks usually exhibit **catastrophic interference** (aka *catastrophic forgetting*): previous concepts are *overwritten*, and thus forgotten. As an example, if we teach an agent how to play a game and it successes with it, when exposed to a second game it will forget about the first. The same might apply to a CNN that classifies cats and dogs, if provided with another task, it will not be able to pursue its original one. The issue is related to **distributed representations**: weights are shared among concepts, so learning new concepts alters the previous representations. A couple of solutions are:

[9] ^ https://pathak22.github.io/large-scale-curiosity/

- **Interleaved learning**: *previous experience* is re-iterated along with samples from *new* distribution. Practically, this solution works. However, the training set becomes huge if many tasks are to be learnt. In addition, the new tasks are "diluted", while they should be given more importance, with the result of a much slower learning.
- **Weights protection**: one tries to remember old tasks by *selectively slowing down* (or even preventing) learning on the weights important for those tasks. This can be pursued by selectively decreasing the learning rate. However, the main problem is that as the number and complexity of tasks increases, the number of important weights will become important.

Complementary learning systems

An actual research is on *how* to model brain *acquiring new knowledge* **without interfering** with already existing one. Moreover, one might want to find a way to **rapidly** *store new information*. For doing so, we would need *two* separate learning systems, as it *actually happens* in human brain. (see Fig. 5.12)
The **Neocortex** is the part many DL systems want to simulate, and there occurs *iterative learning with experience* of *structured knowledge*. Each learning experience represents a *single sample* from the environment distribution: a small learning rate must be used to find an accurate estimate of the domain statistics. Finally the *generalization* is promoted by *distributed representation*.
The second specific area is the **Hippocampus**, which provides a *rapid*, *one-shot memorization* of specific items and experiences. This type of learning occurs when learning specific episodes, which can be *crucial for survival*. The *representations* in the Hippocampus are much sparse and localized, in order to avoid interference. Moreover, during sleep (or mind-wandering), it can send signals to the *visual cortex* in order to "replay" them, thus consolidating the knowledge into the neocortex. In addition, this reactivation/replay can be interleaved with current sensory experiences in order to avoid catastrophic interference. These **sharp-wave ripples** were found empirically in rats, and are very specific also for the topics being replayed: if these waves are manipulated, these memories turn out to not be consolidated any more. Surprisingly, these sharp-wave ripples in place cells are **biased** to reflect trajectories through *rewarded locations in the environment*, and eventually find **shortcuts** by stitching together components of trajectories. Finally, they support **look-ahead online planning**, namely something similar to a *roll-out policy* in RL.

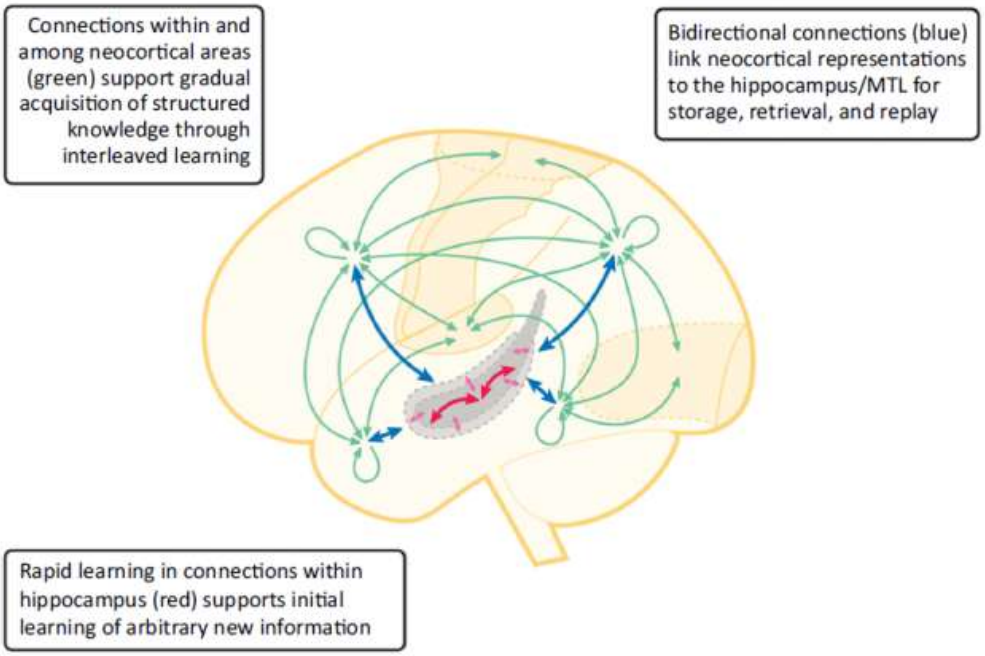


Figure (5.12) – Portions of the brain devoted to different tasks and different learning processes.

The Hippocampus, by biasing the replay towards rewarding events, may allow to circumvent the general statistics of the environment by **reweighting experiences** such that *statistically unusual*, but *significant events* may be afforder **privileged status**.
Memory systems are likely *optimized* to the **goals** of an organism, rather than to simply mirror the environment structure.

This **experience replay** mechanism, was one *crucial milestone* for successfully training the "Deep Q-Network" to be able to play Atari videogames. Learning was based on randomly chosen subsets of recent experiences stored in the **replay buffer** interleaved with ongoing game-plays. This minimized learning from *consecutive samples*, which are naturally strongly correlated. Finally, **biasing replay** towards experiences associated with *larger rewards* yield further gains ("*prioritized replay*").

CHAPTER 6

DL HARDWARE

(Lesson 16 of 22/12/20)
Compiled: September 20, 2021

Especially in Deep Learning, one should take care also of the **Hardware** being used, that may affect the performance of the models.

As one can see from Fig. 6.1, the computational resources required for training more advanced models scale up very quickly wrt time and had two different slopes, depending on the "Era". Specially, in the last few years ("*Modern Era*"), one can see as compute demands are doubling at a *very high* speed. Surprisingly, despite DL networks have been taking inspiration from the brain, they require much more energy to function. Indeed, large companies provide electricity to their datacenters building power plants on purpose: this is one of the main reasons why small companies are penalised for doing DL research.

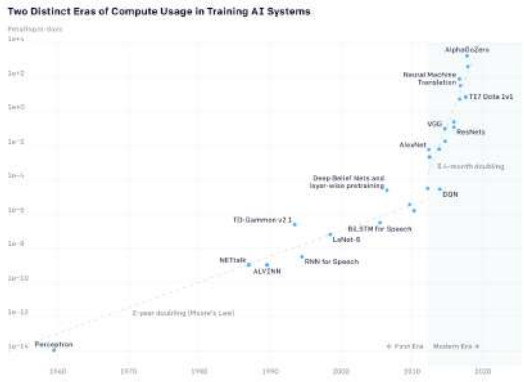


Figure (6.1) – Computational power needed for training different DL models.

Many researches now have been trying to optimize *power consumption* for DL models since in the long run it cannot be sustainable: **metabolism and efficiency**, as long with performance, are the new targets for the *AI*.

The critical **issues** for these methods are:

- **Computational complexity**: despite learning not being linear, it should be *fast* and inference should even be *real-time*
- **Power consumption**: both learning and inference should operate on *low-power* devices

Moreover, it has been shown (see Fig. 6.2) that **hardware** is now *limited* by its *clock speed* and *power consumption*. This means that further improvements, specially in these fields are not likely. However, if one relies on *Quantum Computing* for solving such issues thus starting a *new generation* of DL, much more work has to be done in that direction.

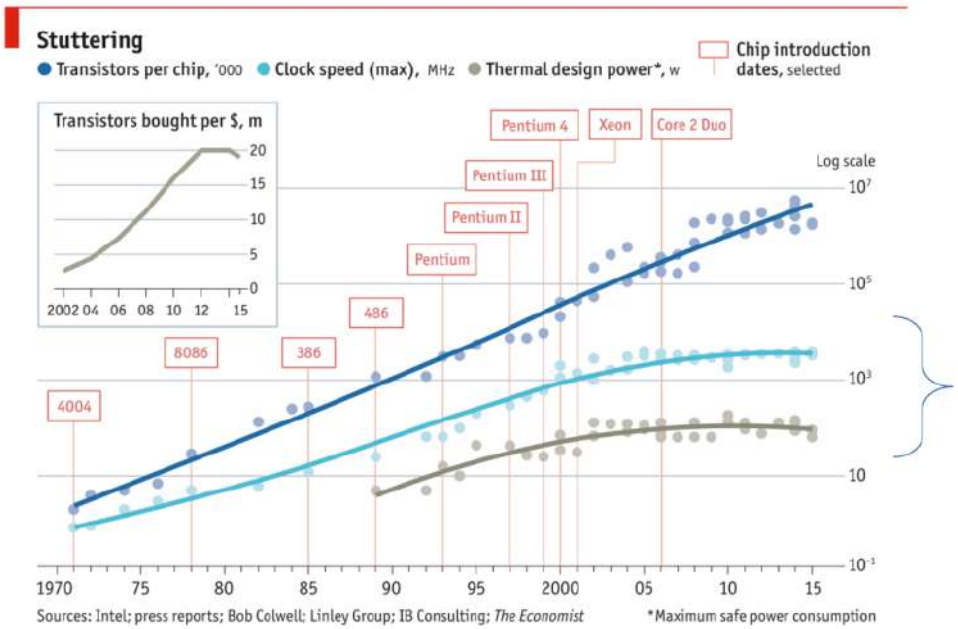


Figure (6.2) – Moore's law seems now to be slowing down, specially for clock speed and power consumption.

In order to **speed up computation**, one can exploit *parallel hardware* computations. This happens due to the Neural Networks nature, which perfectly fits **parallel distributed processing**. Moreover, the dominant computations ($\sim>$ 90%) are quite simple, namely **add** and **multiply**, which are usually performed by domain-general arithmetic logic units (*ALU*s). Due to these reasons, GPUs have taken the lead in the very last years: they consist of *many* and *simple* ALUs with fast, shared memory, and are suited for **element-wise operations**, since they were originally created for *rendering* high-quality images. Moreover, using **advanced compilers** frequent, and useless (e.g., multiply by 1, 0), operations are optimized, as it is optimized also memory access via the use of cache. Finally, some other new *libraries* have been created to increase velocity in *linear algebra* (BLAS, CuBLAS...).

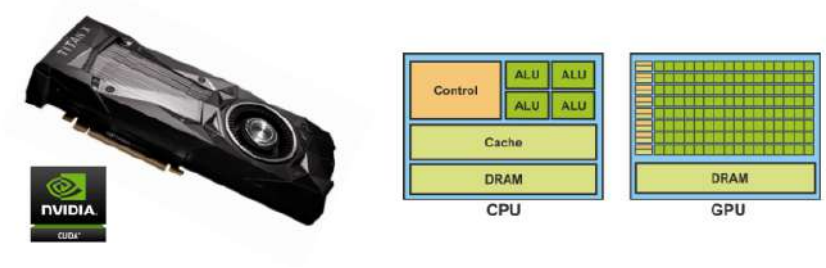


Figure (6.3) – GPUs nowadays are a perfect fit for training parallel distributed processing systems.

However, the *fully* computational power of GPUs (or any other parallel computing architectures) often cannot be fully exploited, since data *is not yet ready*. This problem is known as **data transfer bottleneck**: we should reduce the *overhead* due to data transfer from/to RAM memory to/from GPU memory. Indeed, sometimes GPU memory cannot fit the whole network and some *transfer* of data is requested eventually leading to **overhead**.
One among the many possible solutions is to exploit **in-memory-processing** architectures (see Fig. 6.4). Rather than having a unique shared memory-on-chip, every ALU has its own (very small) memory, and communicates with every other ALUs the results of calculations. The main **challenge** these architectures want to tackle is the *optimal data flow*, that is to say to reduce movement of data around the board, thus significantly reducing overhead. However, to implement such systems is really a *complex task*: one tries to *map* the *topology of the network* into the topology of the hardware, in such way that latency is significantly decreased. One the other hand, if a reconfiguration is required, this should come with high energy-efficiency. These architectures can be implemented specially on *Application-specific integrated circuits* (ASICs). [1]

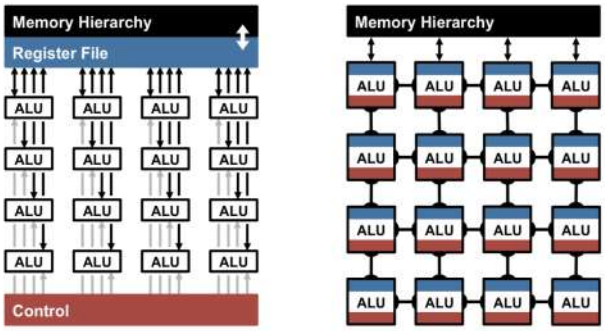


Figure (6.4) – In in-memory-processing archtectures (right), the memory along with the control unit and the ALU are distributed among different *many units*. It basically exploits the fact that many units might have always the same task, thus performing the same operations over time, and having connections in such way that *latency* is very short. (*optimal data flow* problem).

Let us briefly mention one of the research frontiers in DL are **Neuromorphic circuits**. ALUs are now designed with the purpose to replicate *neuronal dynamics* in silico, so that **inference** tasks becomes relatively *easy*. However, **learning** seems to be *very challenging* due to modelling of synaptic connections.

One of the most promising venues in DL is constituted by **Memristors** (see Fig. 6.5), namely semiconductors whose resistance varies *as a function of flux and charge*. Their **advantage** is that they have much more *limited power consumption* when compared to standard parallel computing architectures. However, these devices require to be able to build *nanoscale synapses*. Reconfiguration in memristors is driven by *internal ion redistribution*, as it happens in biological synapses. These *artificial* synapses are typically only a *few nanometers* thick:

[1] ^ http://eyeriss.mit.edu/

even *moderate* electric fields can alter the ionic configuration of the material (e.g., oxidation) and thus change its local conductivity. Current applications for these systems are, for example, efficient neural network implementations in *crossbar form* (possibly 3D) or being *embedded in memory systems*.

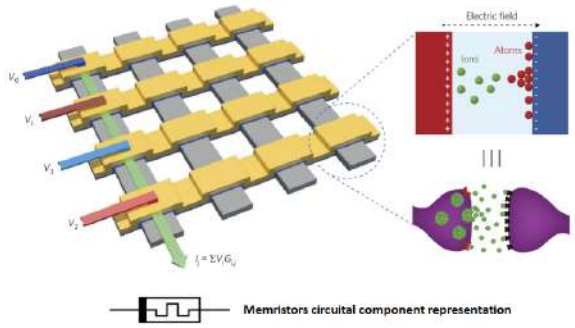


Figure (6.5) – Every neuron, designed at *nanoscale*, tries to propagate signals to other neurons through the axons. The design is made such that it mimics biological synapses, where the different concentrations of ions eventually lead to the propagation of information.

One of the latest approaches to Deep Learning is implementing networks in *optical* systems, thus leading to **Optical Neural Networks** (see Fig. 6.6). In this case, efficiency is really high, since *inference* occurs at *light speed*: the signal, travelling at the speed of light, is *modulated* through the weights (change in *phase*) related to portions of some **diffractive layers**.
Usually, the network is trained *off-line* using GPUs or any other usual hardware. Later, diffractive layers are build according to the optimal parameters previously learned, in such way that *inference* can occur at *real-time*. Even more complicate, one can perform even *learning* directly on these networks. Some proclaimed applications for them are for example to implant a chip into an animal's brain, thus manually driving it to solve tasks. [2].

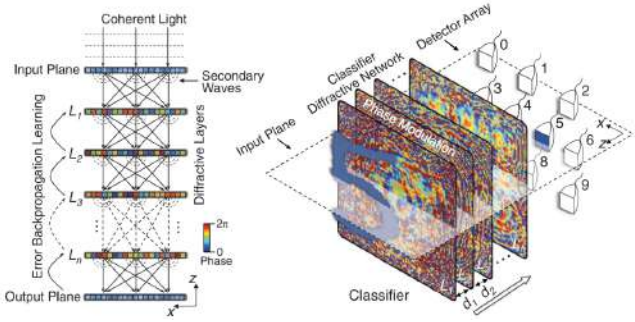


Figure (6.6) – Optical Neural Network representation: inference, due to signal travelling at speed light, can be performed at real-time. Even the learning can done on such networks, but in a more complicated way.

[2] ∧ see https://neuralink.com/