NEURAL NETWORKS AND DEEP LEARNING NOTES

a cura di:
MARCO ZANETTI

Indice

1 Neuroscience basics

Out of all creatures, we can sort them in 2 categories:

- **Automatic (reflexive) systems:** those which only act after a particular actions has occurred (*ex. fly or plants*);
- **Controlled responses systems:** those which can adapt to environment by learning and memorizing (*ex. fishes or humans*).

2 The human brain

The majority of the body is connected to the brain, as can be seen in picture [1]. The peripheral nervous system deal with how we interact with the environments. The spinal cord then convey the information from peripheral nervous system to the brainstem and finally the brain.

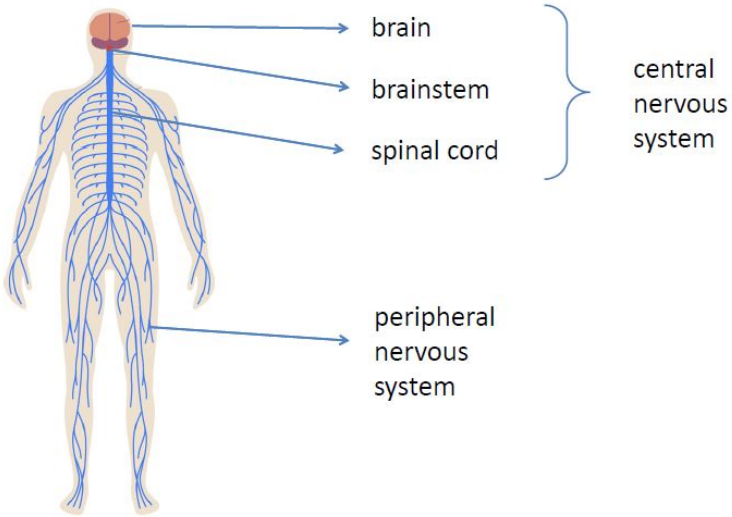


Figura 1: Human body nervous system

The brain is formed by different parts:

- **Brainstem:** control of vital functions (such as breathing, body temperature and pressure, blood circulation);
- **Cerebellum:** motor control, balancing, coordination, schemas, memory;
- **Diencephalon:** transmission of sensory and motor information to/from cerebral cortex, wake and sleep rhythm, alerting, appetite, regulation of endocrinal system (hypophysis and pineal gland);
- **Telencephalon:** divided into two hemispheres, it is mostly constituted by the cerebral cortex , supporting high level cognitive functions.

The human brain is formed by over 100 billions of neurons, plus many other auxiliary cells. On the inside, we can find the gray matter, which represents the various cells bodies, and the white matter, which is the various connections between the cells. The white matter is located on the center of our brain, while the gray matter create the laminar external structure, with up to 6 layers and different type of neuron in each layer. This can be seen in picture [2]. The impressive

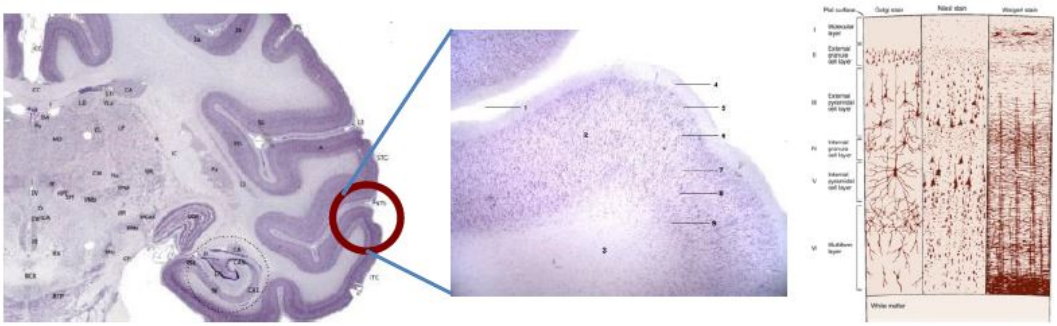


Figura 2: Layer of the human brain

flexibility of our brain can be guaranteed by brain plasticity: an effect called **neurogenesis** occurs by which the neurons are produced by neural stem cells. This effect mainly occurs mostly during embryonic development, but can continue at lower levels through lifetime. Furthermore, new connections between neurons are constantly created while useless ones get pruned. Real time brain dynamics is also modulated by a variety of neurotransmitters. At the purpose of this class we won't cover the limbic system (located at the center of our brain), which mainly is about emotion regulation and episodic memory.

3 The neuron

The goal of neuron is spreading information. Neurons are highly specialized cells that can efficiently and quickly propagate information encoded as electric signals. The branches of the dendrites (neuron input) allows for many different form of information. The numbers of neurons usually stop growing with the adulthood, but the synaptic connections occurs always. The way neurons are developing axons and dendrites follows some minimal wiring principle: the formation of axons and dendrites effectively minimizes resource allocation while maintaining maximal information storage. This doesn't get considered when working with neural networks.

3.1 Neuron morphology

The nucleus of the neuron reside inside the **soma**. The **dendrites** are receptors of the information forwarded from other neurons. Through the **axon** information get propagate by the neuron. The axon can sometimes have a shell surrounding the axon in order to isolate it electrically for efficiency. The axon terminal then propagate the information to other neuron's dendrites trough the axon terminal, where synapses resides. The shell surrounding the axon is called **Myelin**. Trough Myelin, information speed can go up to 30 times faster than usual (150m/s).

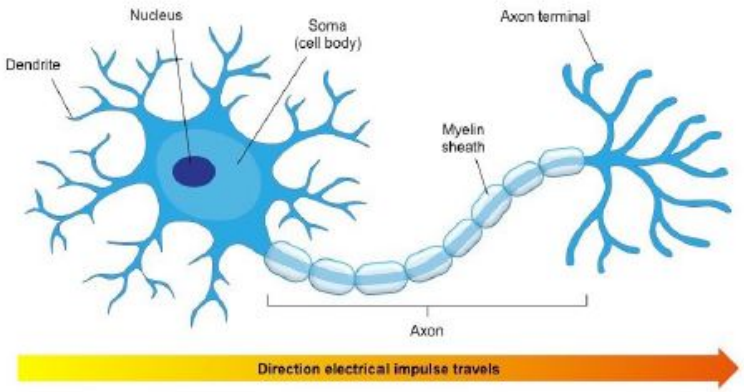


Figura 3: neuron

3.2 Membrane polarization

A voltage difference is maintained between the exterior and interior of the neuron, so that the cell is electrically polarized. Ion channels are regularly placed across the membrane and can open as a result of chemical or voltage change.

3.3 Electric signal propagation

To propagate the electric signal along the axon, the neuron uses the membrane polarization as follows:

1 The first part of the axon (close to the soma) depolarizes and an action potential is initiated by the opening of Na channels;

2 Shortly after, K^+ channels also open allowing potassium ions to exit;

3 For small voltage increases from rest, the K^+ current exceeds the Na+ current and the voltage returns to its normal resting value (-70 mV);

4 If the voltage surpasses a critical threshold (typically 15 mV higher than the resting value) the sodium current dominates. This causes even more channels to open, producing a greater depolarization of the cell membrane;

5 The process proceeds explosively until all of the available Na^+ ion channels are open. Consequentially, the neuron "fires".

6 The Na^+ influx spreads to the adjacent part of the axon, while Na channels in the current location close.

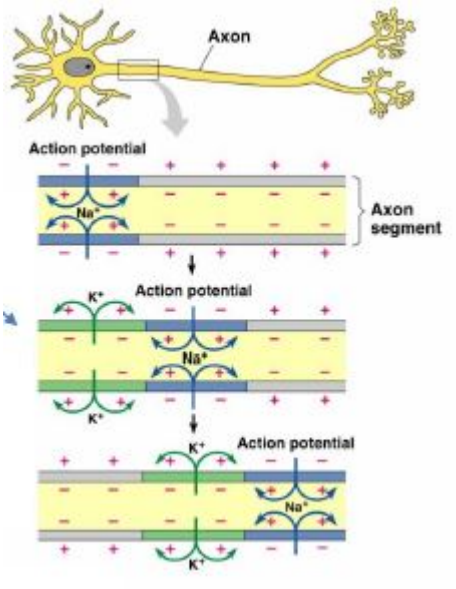


Figura 4: Polariztion of the axon

7 Now K^+ current dominates and re-polarization begins, returning the electrochemical gradient to the resting state (or even below).

3.4 Synapses

There are 2 different type of synapses: **electrical** synapses and **chemical** synapses. The first one are very fast but wired, while the chemical use a biochemical transmission process. While slower, chemical synapses are much more flexible. Synaptic transmission is usually asymmetric (directional). However, there are cases where bi-directional communication is also observed, such as in retrograde signaling. The biochemical transmission process occurs as follows:

1 The electric signal arrives at the synapse, causing a voltage transient of the pre-synaptic cell membrane;

2 Channels permeable to calcium ions open, thereby increasing the internal concentration of Ca^{2+};

3 Calcium sensitive proteins are activated, causing synaptic vesicles to fuse with cell membrane and to release specific molecules (neurotransmitters) in the synaptic cleft;

4 (Some) neurotransmitters binds to the corresponding chemical receptors;

5 Ions enter (or exit) through the receptors, polarizing the membrane of the post synaptic cell;

6 Neurotransmitters are either deactivated or reabsorbed in the pre-synaptic cell.

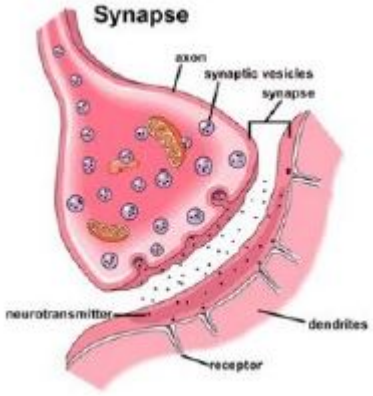


Figura 5: Synaptic transmission

We can modulate how much information is transmitted by changing the number of neurotransmitters transferred. When learning is occurring, we have a much larger number of neurotransmitters. If neurons are no longer communicate, the number gets gradually reduced. This allows flexibility. The synaptic efficiency get represented as a value in Neural Networks.

3.5 The Hodgkin Huxley biophysical model

Hodgkin and Huxley demonstrated that macroscopic ionic currents in the axon could be understood in terms of changes in Na^+ and K^+ conductance in the axon membrane. Given the membrane electric charge (expressed in Coulomb) Q, the membrane capacitance (expressed in Farad) C_m and the membrane tension V, we can say express the following equation:

$$Q = C_m V \tag{1}$$

Taking the time derivative of that formula we can find how much current is needed for change the membrane potential at a given rate:

$$C_m \frac{dV}{dt} = \frac{dQ}{dt} \tag{2}$$

We can see that the rate of change of the potential is proportional to the rate at which charge builds inside the cell. Also, the rate of charge buildup is equal to the total amount of current entering the neuron.

3.6 Integrate and fire model

The idea is that the neuron membrane is acting as a capacitor, which try to accumulate charge. Depending on the capacitance of the membrane, we can tell how much current is needed to change the potential of the membrane, as seen in the formula before. An action potential occurs whenever the input current causes the membrane potential to reach a critical (and constant threshold V_{th}. The action potential is modeled as a Dirac delta function (single pulse). After the neuron has fired, the membrane potential is set back to the resting value. After the critical point, the firing frequency of the model thus increases linearly (and without bound) as input current increases. The equivalent circuit consists of a resistor and a capacitor in parallel. The

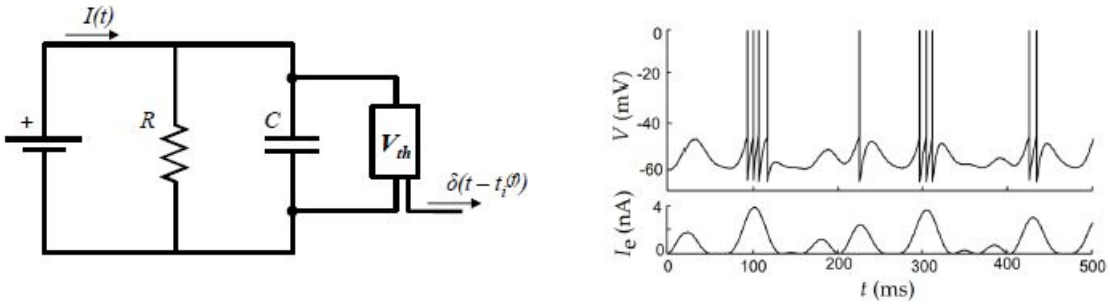


Figura 6: Integrate and fire model equivalent circuit

circuit works as followsx:

1 The input current $I(t)$ charges the RC circuit;

2 If the voltage across the capacitance points exceeds the threshold V_{th} an output pulse $\delta(t - t_i^{(f)})$ is generated;

3 After the action potential is generated, the charge is reset to the initial value.

The linearity following the critical threshold makes it easier to analytically describe the neuron dynamics, at the same time preserving its overall non linear nature (there is still a discontinuity around V_{th}). This type of activation function is extensively used to study emergent properties at the network level, when multiple neurons are interconnected. One example is the Rectified Linear Units (ReLU) used in deep learning. A neuron with a ReLU activation can be described trough the inputs x_i who gets in the synapses. We then sum all the currents from the neurons. If the signal surpass the threshold, the neuron start to fire the signal linearly. The output of the function $g(.)$ is

$$out = g(\sum w_i x_i - v_{th}) \tag{3}$$

Action potentials do not just carry binary information (fire/not fire): they are rather organized into coherent temporal patterns. We can characterize this code by computing the spike count

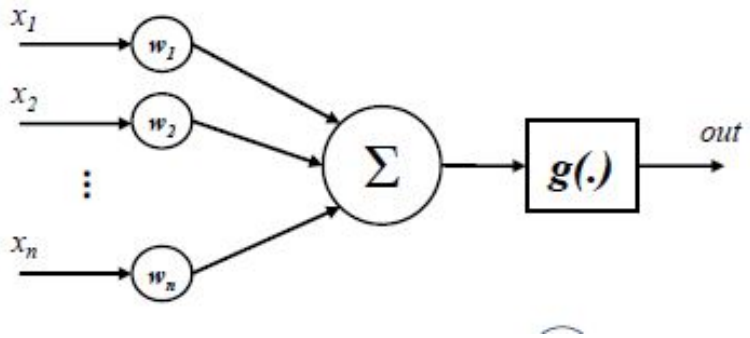


Figura 7: Rectified Linear Units (ReLU)

rate of the neuron, which corresponds to the number of action potentials (spikes) generated in a given time window T corresponding to a "trial".

$$r = \frac{n}{T} = \frac{1}{T}\int_0^T dt\rho(\tau) \tag{4}$$

Due to the high variability of neuronal responses (noise, intrinsic state of the neuron, neighborhood activity, ...), usually spike count rates are measured by averaging the number of spikes over many trials. We can also more precisely characterize the temporal pattern by computing a time dependent firing rate , obtained by counting (and averaging across trials) the number of spikes within much shorter intervals (between t and $t + \Delta t$):

$$r(t) = \frac{1}{\Delta t}\int_t^{t+\Delta t} d\tau \left\langle \rho(\tau) \right\rangle \tag{5}$$

By measuring firing rates, we can try to establish which are the stimulus properties that are mostly responsible for the activation of the neuron. We thus define the neural response tuning curve as the average firing rate conditioned on the presence of the stimulus property under investigation. We can also proceed in the opposite direction, and ask which stimulus was present when the neuron was firing. We thus define the spike triggered average stimulus as the average value of the stimulus conditioned on the activation of the neuron.

4 The neuronal code

Neural computation might serve several purposes like:

- **Signal amplification:** reduction of noise;
- **Signal Signal compression:** : reduction of redundancy to make the signal more efficient;
- **Signal recoding:** non linear transformation to highlight (make explicit) internal structure for later use or to combine signals from different sources;
- **Signal storing:** memorization (as an internal state) to constrain subsequent processing.

There are two complementary perspectives on sensory processing. The first is the **bottom-up** perspective. The idea is that neurons are filters. Neurons are tuned to particular characteristics

(features) of the sensory signal that are trying to match, and fire when they are shown a signal with such features. Characteristics than are not part of the feature set are discarded during processing. Characteristics that are successfully detected are instead further processed. The second perspective is the **top-down** view. The idea is that neurons are hypothesis for the data: they can predict the stimulus before the stimulation is occurring. Neurons create hypotheses about how to interpret the upcoming sensory signal (evidence), which are then tested during sensory processing. If the hypotheses do not perfectly match the new sensory information the neuron either cast the evidence to fit the hypotheses or adapt the current hypotheses to better fit the evidence.

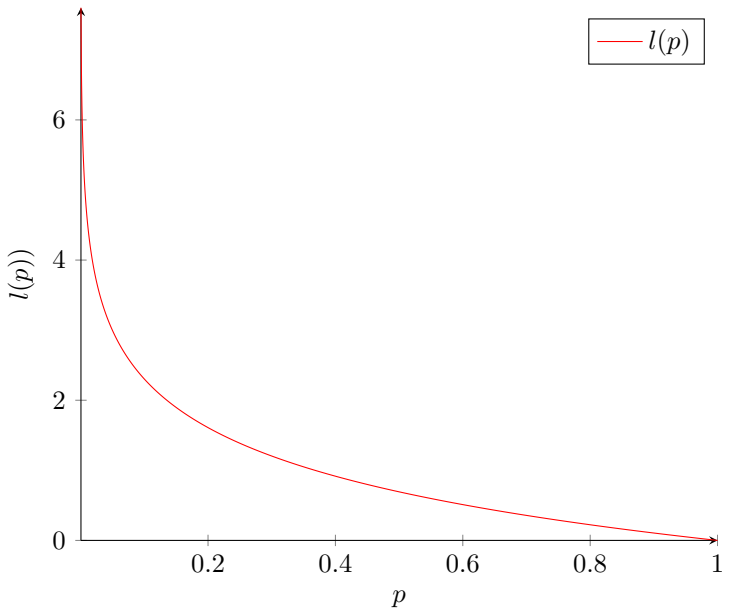
4.1 Efficient coding principles

Information theory was originally invented to study information transmission in communication systems. For design an optimal transmission code we start with calculating the expected length of messages sampled from specific probability distributions using various coding schemes. Then, the more frequent messages will be assigned shorter codes. The cost for sending frequent codes is then minimized, while codes which occurs rarely has greater cost. The basic idea is that learning an unlikely event is more informative than learning something that has likely occurred. According to this idea:

- Likely events should be assigned low information content;
- Certain events should be considered as having no information content at all;
- Unlikely events should be assigned high information content;
- Independent events should have additive information.

We can describe this mathematically by considering the self-information of a single event:

$$I(x) = -\log P(x) \tag{6}$$



We can adapt this idea to a distribution of events trough entropy. The entropy of a distribution is the expected value of the self-information of each event. The Shannon entropy of a distribution of events is the expected amount of information in an event in such distribution. It gives a lower-bound on the number of bits required on average to encode events. We describe it as follows:

$$H(x) = E_{x\sim P}[I(x)] = -E_{x\sim P}\log P(x) \tag{7}$$

The idea is that neuron are trying to encode efficiently minimizing the time they are active. So then the communication channel is just a sensory pathway, and the number of spike represent the efficiency of the code for representing sensory signals. The solution is to assigning cheap codes to more frequent sensory signals. Dependency on frequency information implies that the nervous system should adapt its internal code depending on past experience, that is, it should be able to model the statistical structure of the environment.

4.2 Predictive coding

Sensory stimulation does not occur statically: sensory input is conveyed as a continuous stream , which raises the possibility that statistical information might (and should) be collected over time. For example, the brain might extract the statistical structure of a language to more efficiently process linguistic input. Some investigators have thus proposed that sensory circuits might be transmitting prediction errors rather than input signals: if the sensory prediction is correct, nothing needs to be transmitted, otherwise if the sensory prediction is wrong, only the mismatch (error) is propagated. The underlying assumptions are that the environment has limited entropy, and the brain is able to learn such statistical regularities.

4.3 Measure neuronal activity

We can empirically measure neuronal activity at either a cellular level or at a macroscopic level.

Cellular level Even at a cellular level, we divide the process between various techniques such as:

Single cell recording In a *single cell recording*, we use thin electrode that target a single cell. It get inserted into the brain whereas it can detect the electric activity (trough the rate of change in voltage) of the brain. Also, electrode recording can be done either intracellularly or extracellularly. Single cell recording can get quite noisy, so it's usually better to use

Multielectrode recording Instead of just studying a single neuron, here we simultaneously record from a population of neurons. Even if we obtain a lower spatial resolution, we can visualize and study better local neuronal circuits. Here, electrical activity corresponds to *Local Field Potential*. 1D, 2D and 3D recordings are possible.

Optical imaging recording This type of recording is only allowed on species whereas we can control freely the genes (for example the zebra fish).

Macroscopic level At a macroscopic level, we can utilize different ways to get the neuronal activity of the brain, each with it's pros and cons.

Electroencephalography (EEG) Electroencephalography is a non-invasive technique, since the electrode grid is placed on top of the scalp. It mostly collects signals coming from the superficial layers of the cerebral cortex. EEG measures electrical activity at much coarser spatial and temporal levels. However, the signal can get noisy since we can't record directly from the brain.

Magneto-encephalography (MEG) Rather than recording electrical activity, MEG records magnetic fields produced by electrical currents. Some pros rely on the fact that we obtain a greater spatial resolution than EEG, and it's not dependent on head geometry.

4.4 Functional Magnetic Resonance Imaging (fMRI)

fMRI not directly measure electrical activity, but rather change in blood flow. As You think, You activate some areas of the brain. This areas will therefore need more blood (oxygenated hemoglobin). fMRI can be a very precise technique, but it's slow temporally and it's an indirect measure.

4.5 Functional Near-Infrared Spectroscopy (fNIRS)

Similar technique to fMRI, fNIRS emit a light flash (trough laser emitters). We'll obtain different reflection properties studying the oxygenated hemoglobin. The cons resides in the fact that it can record only from superficial layers of the cerebral cortex.

5 Coding principles at the population level

We can divide the way the neuron population works as:

- **Localist representations:** The idea is that each entity gets encoded by a single neuron. Through this representation, neurons encode orthogonal features. Pros can be found in it's metabolic efficiency (only few units are active) and the fact that there is no interference between different neurons. The issues are that it requires many units, it's hard to generalize and it's quite sensitive to signal variability.
- **Distributed representations:** each entity is encoded by a pattern of activity distributed over many neurons, and each neuron is in turn involved in representing many different entities. The pros are the processing efficiency , the generalization properties, and the resilience to noise. However, it consume more energy, and interference can occurs if shared among task.
- **Sparse representations:** Sparse representation is a compromise of the two above. Here We have a distributions over many neurons but not in all of them. The coding strategy and the degree of sparseness in neuronal activation seem to depend on the brain area. This is also the representation used by the brain. To measure sparseness, we can use for example the kurtosis coefficient:

$$k = \frac{1}{n}\sum_{i=1}^{n}\frac{(r_i - r^*)^4}{\sigma^4} - 3 \tag{8}$$

  where r represent the firing rate, σ represent the standard deviation for the average, and n is the number of neurons.

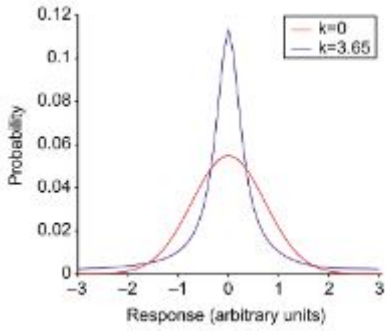


Figura 8: Example of different kurtosis coefficient output

While in local representations every item is orthogonal, in distributed representations items can be linearly dependent due to overlap of features. This allows for easier track of similarities and dissimilarities.

5.1 Visual system

Neurons in the retina, thalamus (LGN) and primary visual cortex (V1) respond to light stimuli in restricted regions of the visual field called their receptive fields. Neurons at higher levels of visual cortical processing (V2, V4, medial and ventral Inferior Temporal areas) respond to more complex configurations and receptive fields span the whole visual field. Of particular interest for Us is the ventral stream that is responsible for visual recognition. In retina and thalamus, neurons respond to basic inputs. Retinal ganglion cells and LGN neurons are mostly selectively activated by circular spots of light surrounded by darkness (on-center cells) or by dark spots surrounded by light (off-center cells). The idea is that those cells are contrast-sensitive: the neuron is active only when the core is getting stimulated. We can put together many contrast neurons (V1) representing basic features to build something more complex. We can detect now orientation, for example. The idea is that we start with simple features in retina and thalamus, then in the visual cortex we combine them to produce more complex features.

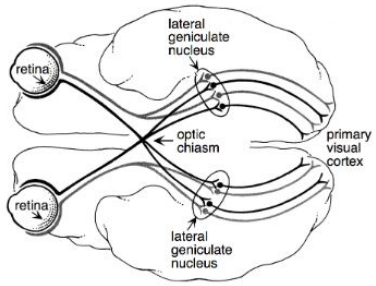
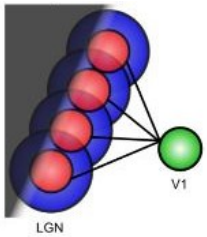


Figura 9: Representation of human visual system

Position nearby in the visual fields are encoded by nearby neurons. Therefore, the topology is preserved. The same concept is represented by nearby neurons in the cortex. This is called a topographic map. This constraint isn't used in deep learning models at the moment.



6 Computational principles for early visual processing

We aim at representing an image I in terms of a linear superposition of basis functions, where the coefficients α_i are dynamic variables that change between image:

$$I(x,y) = \sum_i \alpha_i \phi_i(x,y) \tag{9}$$

This is called **linear generative model**. $\alpha_i \phi_i(x,y)$ represents the basis functions. Those basis have the same dimension as the input, however note that usually we consider image patches. There are more efficient models exploiting smaller basis functions plus convolution. It is showed in picture [10]. We talk about **efficient coding** whenever we find a set of basis functions forming a complete code (spans the entire pixel space) and resulting in coefficient values as "independent" as possible.

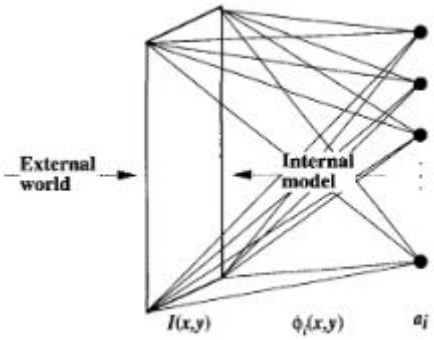


Figura 10: Linear generative model

6.1 Efficient coding of natural images

We can divide it into different types of coding:

Principal Component Analysis PCA finds uncorrelated (mutually orthogonal) basis functions due to independence. Dimensionality reduction achieved by selecting as fewer components as possible. How many should i choose depends on the variance in the dataset. PCA is computed at once over the whole dataset of images.

Independent Component Analysis The basics functions needs to be statistically independent. The underling assumption is that the complex image is generated trough superimposition of different sources. Similar results can be obtained through Non-negative Matrix Factorization (NMF), where the constraint is that signals must be added.

Sparse coding The independence we want is provided by a penalty on the total number of active units. The probability distribution of each coefficient's activity is peaked around zero. Also, the basic function code must be *over-complete*(The number of basis functions is much

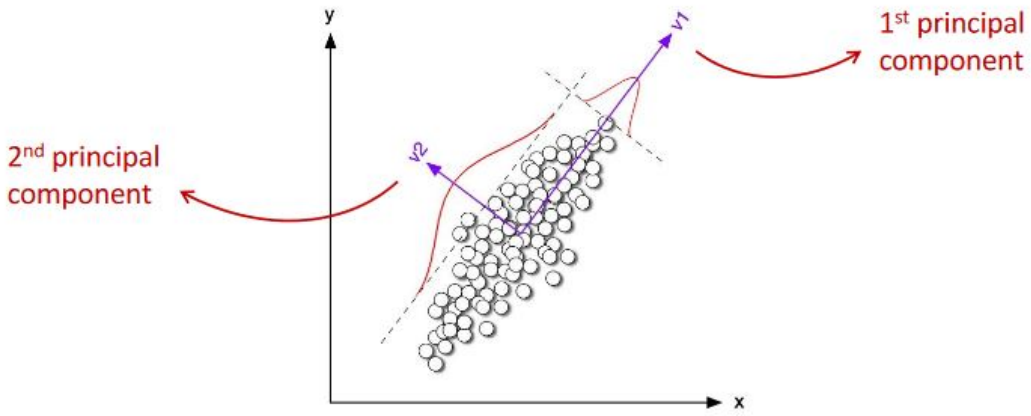


Figura 11: Principal Component Analysis choice of basis

higher than the dimensionality of the data): we select a small number of active functions from a large dictionary of possible functions. The issue with sparse coding is that it's optimized iteratively. To formalize sparse coding mathematically, keeping in mind equation [9], we write:

$$E = -\left[preserve\ information\right] - \lambda\left[sparseness\ of\ \alpha_i\right] \tag{10}$$

In particular, λ represent how much important is the sparsity contest. The higher λ is, the more accurate the sparse coding will be. Also, we define:

$$\left[preserve\ information\right] = -\sum_{x,y}\left[I(x,y) - \sum_i \alpha_i \phi_i(x,y)\right]^2$$

This represent how well the code describe the image trough mean squared error between external image and reconstructed image. Then we explicit that

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

This represent the cost on the summed values of coefficients: there might be several functions satisfying this requirement, but often a simple norm is sufficient.

7 From single neuron to neural networks

We have seen how a single neuron works as an integrate-and-fire model. We sum all the neuron inputs weighted by their synaptic strength. Then we have an activation function $g()$ that whenever the inputs sum surpass a critical threshold will allow the neuron to fire. The output will be:

$$out = g(\sum w_i x_i - b_{th})$$

Where w_i and b_{th} are model parameters (voltage critical threshold is called *bias*). This is only and idea that can be implemented in many ways. First we have to define the **architecture** of a network.

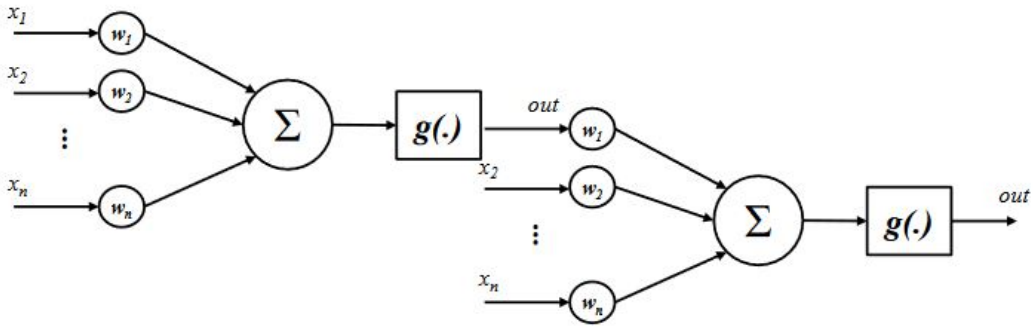


Figura 12: Connection of different neurons

7.1 Network Architecture

Network architecture is formed by its **topology** and its **directionality**. For example, in a 2D representations, we could have as topologies:

- **Fully-connected models:** every node is connected to all the other nodes;
- **Bipartite graph:** we are creating two separate sets of nodes (layers) whose cannot be connected each other. This is very efficient computationally; **Multi-layer models**;
- **Tree-like structure:** the topology is hierarchical;
- **Lattice models**.

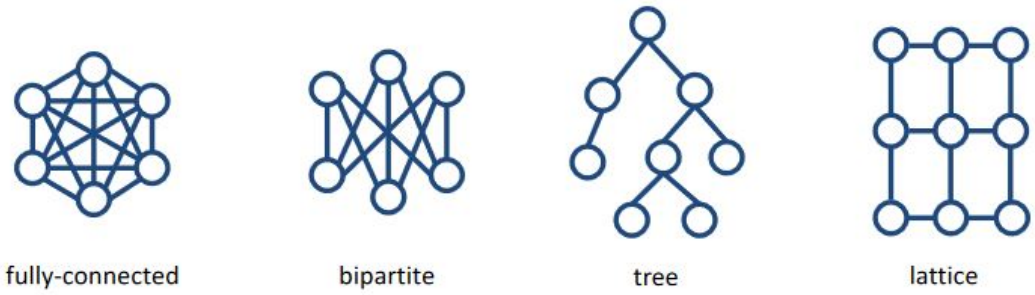


Figura 13: Examples of different topologies

Once we specify the topology, we are interested in its directionality. The idea is that we use graphs to represent and manipulate complex joint probability distributions. The *nodes* represent the random variables, while the *edges* (which form the connections) are the relations between each node. In directed models, the relation between each node is of (parent-of) relation; In undirected models, the edges describe correlation, but we don't know the cause and the effect. Then the parameters, which are the connections weight in every edge, and the structure, which is the topology of the model, can be defined in a **theory-driven way** which is based on priori knowledge. We are more interested in **data-driven** approach that regards estimation (learning). We can say that topology of a graph specify conditional independence on the distribution. The

joint probability distribution can thus be factorized only considering local interactions between variables. We can do this because we know that some variable are influenced by a defined set of variables.

Bayesian networks This works in directed(acyclic) graphical models, where edges represent *<parent of>* relations. Each node (variable) has an associated conditional probability distribution. A useful concept is that of **markov blanket of a node**, which is the set of variables that when are observed makes the node considered independent from all other variables.

Markov networks We consider undirected graphs where edges represent *<affinity>* relations. Each edge has an associated factor (the general function $\phi : R \to R$) that indicates the correlation strength between two variables. We can then factorized the joint of the probability distribution as a product of local factors:

$$P(A, B, \ldots) = \frac{1}{Z} \prod_{i=1}^{N} \phi_i(X_1, \ldots, X_k) \tag{11}$$

The markov blanket of a variable in an indirect model corresponds to the immediate neighbours.

On representing arcs on nodes When two variables interact through an undirected connection, it is like having two symmetric (same weight) directed connections. f we want to specify a bidirectional influence with different weights, then we must use two separate, directed arrows.

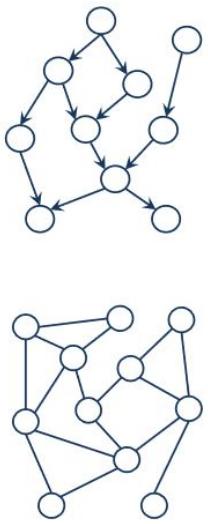


Figura 14: Network directionality

8 Machine Learning basics

Let's start discussing some biological learning principles. We already know that learning and memory are supported by synaptic plasticity. Synaptic plasticity is activity-dependent, which means that the synaptic efficacy (weight) is affected by the activation of the pre-and post-synaptic neurons. We need to repeat this activation in order to obtain an increase of synaptic connectivity (learning by iteration). In some case however we have a one-shot learning where we just observe a single event and it gets stored in our brain. A short summary of how the key biochemical processes underlying synaptic plasticity works is:

1 The overall level of neural activity on both ends of the synapse drives the influx of calcium ions (Ca^{2+}) in the post-synaptic spine via NMDA channels;

2 Synaptic weight changes are driven by the concentration of post-synaptic Ca^{2+} in the dendritic spine associated with a given synapse: low levels of Ca^{2+} cause synapses to get weaker, higher levels cause them to get stronger.

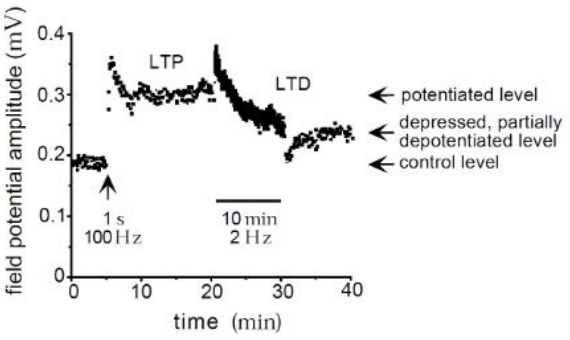
Long term potentiation and depression Usually we define two different ways for implementing synaptic plasticity:

- **Long Term Potentiation (LTP)**: we increase the efficacy of the synapse and the threshold of the efficacy;

- **Long Term Depression (LTD)**: if we not stimulate the synapse, we'll have a decrease on channel efficacy.

Both directions of synaptic plasticity are equally important for learning.

Also the relative timing of pre-and post synaptic action potentials plays a critical role in determining the sign and amplitude of the changes in synaptic efficacy. Pre-synaptic spikes that precede post synaptic action potentials produce LTP, while pre-synaptic spikes that follow post synaptic action potentials produce LTD.



Machine learning principles A learning algorithm is just an adaptive system which improves performance on a specific domain according to experience (training examples) and is able to generalize to new, unseen situations. We distinguish various framework relating the machine learning:

Supervised learning (classification, categorization, function approximation, regression). In supervised learning we assume that there's a supervisor who's helping us solving a task. This give a the learner a "teacher" guiding it and giving it feedbacks. It's very powerful, but cognitively and biologically it's often implausible.

Unsupervised learning (representation learning, feature extraction, clustering, dimensionality reduction). The learner tries to discover the underlying regularities of the world by only observing its environment. We conceive cognition as a "model building" process.

Reinforced learning (control policies, decision making). The learner/agent still tries to discover regularity of the world, but can also interact and manipulate with it. This is extremely useful, but computationally demanding.

8.1 Gradient descend

Gradient descent gets implemented trough:

- **Defining a loss function:** It defines the performance measure that we aim at optimizing (classification accuracy, reconstruction error, ...). This should be a differentiable function. It is often difficult to choose a performance measure that captures the desired behaviour of the system.
- **Experience:** We need a (possibly large) data set containing samples from the distribution we want to model. The more complex a distribution is, the more complex is the model generated. Thus we'll need larger amount of samples. Samples might need to be enriched with extra information (as labels). The whole data set is then usually split into separate sets: training, validation and test sets.

- **Minimization procedure:** Iterative optimization methods are used to find the minimum of the loss function by taking steps proportional to the negative of the (approximate) gradient of the loss function.

We usually don't do gradient descend. We talk about

Stochastic gradient descend The loss function can be measured in every point we have in the dataset. If we have (x, y) that means we are using a supervised learning case (x is the input, y is the desired output). The loss function is decomposes as a sum over m training examples:

$$J(\theta) = E_{x,y\sim\widehat{p}_{data}} L(x, y, \theta) = \frac{1}{m}\sum_{i=1}^{m}\Big(L(x^{(i)}, y^{(i)}, \theta\Big) \tag{12}$$

Where θ usually represent the models parameters. Then we compute the gradient of the loss function:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big(\nabla_\theta L(x^{(i)}, y^{(i)}, \theta\Big) \tag{13}$$

Rather than computing the gradient over the whole training set, in stochastic gradient descent we approximate it by using a subset containing m' training examples:

$$g = \frac{\nabla_\theta}{m'}\sum_{i=1}^{m'}\Big(L(x^{(i)}, y^{(i)}, \theta\Big) \tag{14}$$

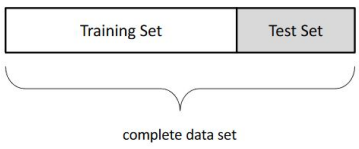
Then we change the weight θ according to this equation, where ϵ is the learning rate:

$$\theta \leftarrow \theta - \epsilon g$$

The idea is that we compute the gradient of our loss function of a subset of data, change the parameters accordingly and then repeat of a new subset of parameters.

Convex optimization For well-conditioned convex problems (only one global minimum) the stochastic gradient descend will converge to the optimum, provided that the learning rate decreases with an appropriate rate. This is a very ideal case which will never happens in Neural Networks due to the non-convexity of the optimization. SGD is not guaranteed to arrive at even a local minimum in a reasonable amount of time. However, it often finds a very low value of the cost function quickly enough to be useful.

8.2 Testing the generalization capability



Let's examine how can we evaluate the performance of our learning algorithm. The learning algorithm should be able to generalize the knowledge extracted from the training examples to novel, previously unseen test examples. This is why we usually split the data in a *training set*, used to find the optical configurations of the connections weight, and then have another *test set* used at the end to measure how well the model is performing on new data. The ratio between the size of two sets is usually in favour of the training set. The issue is that the final performance will depend on this choice. Other possible issues resides in

- **Underfitting:** we are underfitting data if we have a huge training error. Our model is not able to capture the complexity of the data.
- **Overfitting:** If we are fitting the data way too accurately, we might have issues on the test data.

The correct model should be able to balance the fitting capacity over the training points and the fitting capacity over the test points.

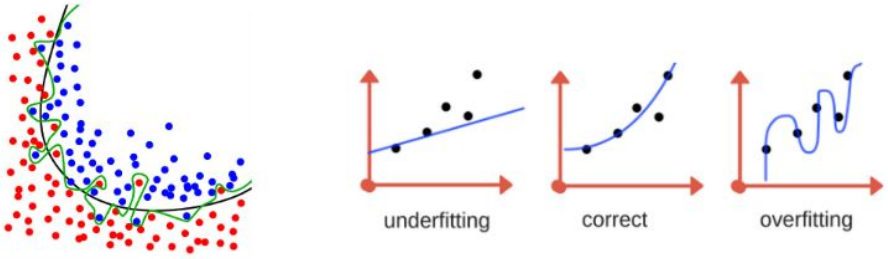


Figura 15: Examples of underfitting and overfitting data

A clearer vision of how the fitting capacity is related to the error can be seen in picture [16]. The idea is to monitor the training data, and when the error start to increase You stop before

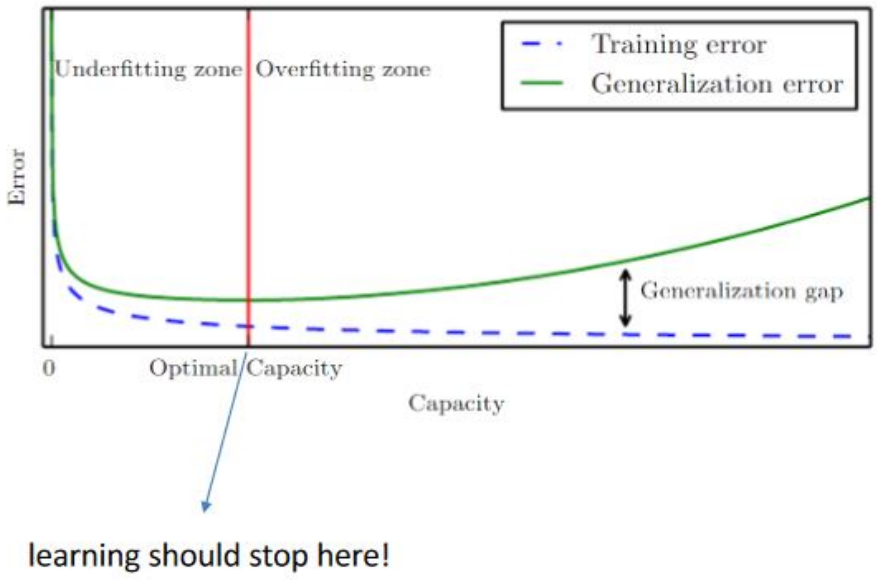


Figura 16: Monitor overfitting

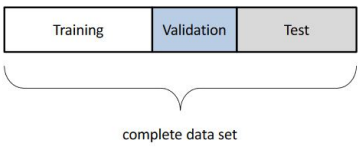
overfitting. This is called *early-stopping technique*.

Model complexity We can control the balance between underfitting and overfitting by tuning model complexity. Simpler models are less likely to overfit the data, but they must have enough capacity to describe the training set. The complexity of the model defines the **hypothesis**

space. To choose the complexity of a model, we can start wit ha simple model, and then increase the complexity of the model until we reach a satisfactory performance. This can be very expensive. In reality, this can hardly be applied. Another option is related to the *VC-dimension*. There's a way to define how complex a model is based on statistical learning theory. We try to discover the better trade-off between complexity of the model and the data we are trying to compute. However, VC-dimension is often hard to apply in practice because the capacity of deep learning models is hard to measure. In deep learning, we take a different approach called **model regularization**. We start with very complex model (very likely to overfit the data), however we impose additional constraint on the model parameters (regularizer). This forces the model to prefer one solution over another in its hypothesis space. Let's examine what's the best way to decide which model use. According **Occam's razor**, among models with similar performance we prefer the simpler ones. The idea is to start with the simpler models and increasing the complexity only if needed. A more principle way to have similar result is to use a *Vapnik-Chervonenkis(VC) dimension*: a formalization of Occam's razor according to statistical learning theory. However this strategy is hard to apply to deep learning, due to the high amount of data needed. A solution is to impose strong external constraints on the model parameters in order to limit their degree of freedom. This forces the model to prefer one solution over another in its hypothesis space. To achieve this we can include one or more penalty terms in the loss function, such as:

- **Weight decay:** We limit the weights magnitude, for example by subtracting a small fraction of the weight at each iteration.
- **Sparse coding:** we limit the overall number of units that can be activated at the same time.

There are numerous parameters that can be used that we'll see later on during this course. For tuning this parameters, we use a separate **validation set** in order to limit the risk of overfitting. The idea is to train our data in the training set, then use the validation set act as a temporary test set, where we calibrate the hyperparametres until we get a good enough result. However, dividing the data set into fixed training, validation and test sets can be problematic if we only have a limited amount of data.



The solution to this problem comes in the form of the **Cross Validation** technique. Here we repeat the training, but validation and testing measurement happens on different randomly chosen splits of the original dataset, as can see in picture [17]. We split the whole training set into keyfolds. Then we rotate the folders used for validation in order to reduce the impact of the random splitting process.

One issue can be occur whereas we have different models that fit better different optimal hyperparametres. If we end up in this situation, we might keep all the models and to Ensemble model averaging. Thus we take the model that it's best for the majority of the data.

9 Supervised Learning

The goal is to learn a function that maps each possible input to a certain output. We therefore need one label(desired output) associated to each input pattern. We have two different scenario for supervised learning:

- **Classification/Discrimination:** We want to assign a label to the data points. We can have a *binary classification*, where we want to draw a line separating the two label's plains.

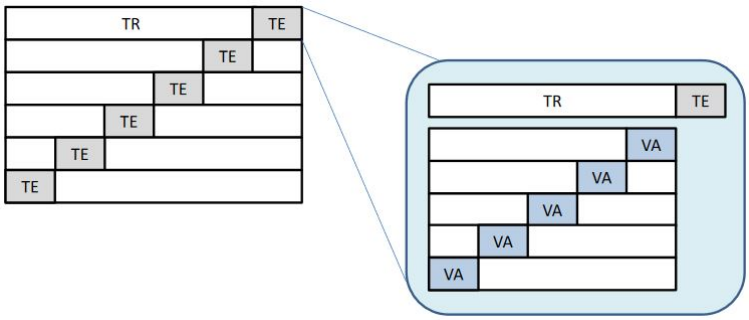


Figura 17: Example of cross validation on a set of data

More generally, we talk about *multi-class classification* with more type of labels to stick to each data point than just two.

- **Regression:** Here every label is a real value.

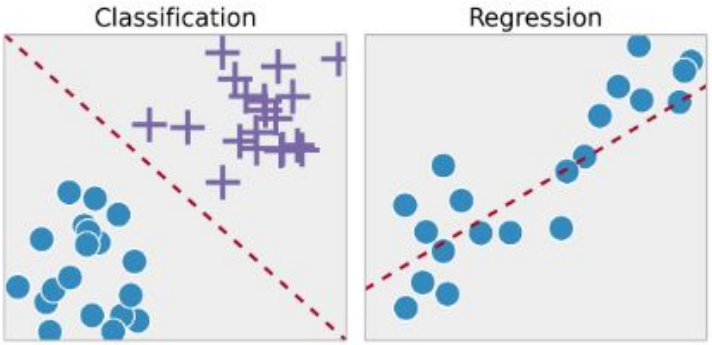
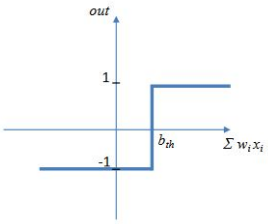


Figura 18: Classification vs Regression

How can we implement a simple binary classifier? This work like the model of neuron we already saw before, using an heavyside function as activation function.



This is called the **perceptron**, which comes with a simple learning rule. We start initializing all weight to zero (or to small random numbers). Then for each training example x:

1 We calculate the out value:

$$out = g(\sum w_i x_i - b_{th})$$

2 We compare it to the target value (the label on the dataset);

3 We update the weights:

$$\Delta w = \eta(target - out)x$$

We are in a supervised context because we have always a supervised target. Then if the predicted output matches the desired label, no changes are made:

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

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

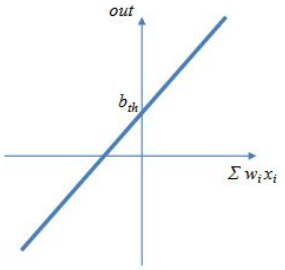
Otherwise, weights are pushed towards the direction of the positive or negative target class:

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

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

It can be shown that, if data points are *linearly separable*, the perceptron learning rule will always converge to one (among the many possible) correct solution. Some problems can occurs considering a perceptron learning rule. First, we could have ma y possible lines separating the two classes. Also, the data might not be linearly separable. Thus we use what we call the **delta rule**.

Delta Rule We are replacing the heavyside function with a linear activation function. We don't impose anymore a threshold, we are always firing and the amount of firing is linearly increasing with the weighted sum. The advantage is that this function is differentiable, thus we can create the gradient of the loss function and then we can use gradient descent to minimize loss. The error is now no more restricted to the values 0, +1 or -1, but will rather encode the actual distance between prediction and target.



Now we can define the Loss function as the mean square error.

$$J(w) = \frac{1}{2}\sum(target - out)^2 \tag{15}$$

Then we can differentiate it by considering the partial derivative divided by each weight:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i(t^{(i)} - o^{(i)}) = \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(t^{(i)} - o^{(i)}) =$$

$$\frac{1}{2}\sum_i 2(t^{(i)} - o^{(i)})\frac{\partial}{\partial w_j}(t^{(i)} - o^{(i)}) = \sum_i 2(t^{(i)} - o^{(i)})\frac{\partial}{\partial w_j}\left(t^{(i)} - \sum_j w_j x_j^{(i)}\right) =$$

$$\sum_i (t^{(i)} - o^{(i)})(-x_j^{(i)}) \tag{16}$$

Just for simplicity, we are not considering the bias in this notation. Then we can express the equation for changing the weight in the opposite direction of the gradient (**gradient descend**):

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = -\eta\sum_i (t^{(i)} - o^{(i)})(-x_j^{(i)}) = \eta\sum_i (t^{(i)} - o^{(i)})x_j^{(i)} \tag{17}$$

9.1 Non-Linear problems

Neither the perceptron or the delta rules can solve more complex problems (like the xor function for example).In this case, we use non-linear projection to solve this issue. We have an input space where the data are not linearly separable and we project every data point into a feature space by means of a general kernel non-linear function ϕ. In the feature space (which is usually of a higher dimension from the original space) data became linearly separable.

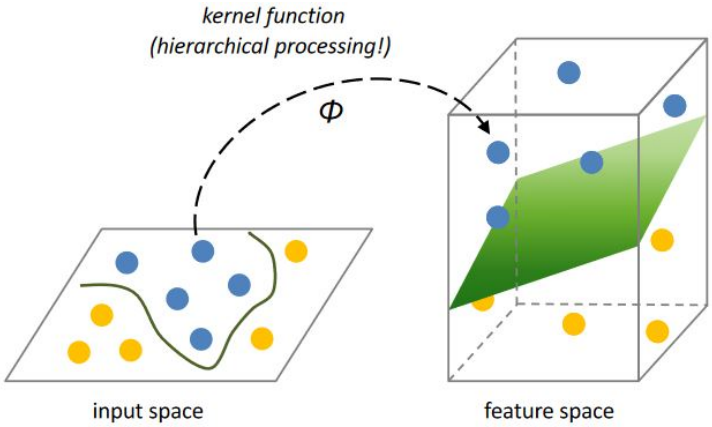


Figura 19: Mapping of a dataset in a higher dimension feature space

Multi-layer perceptrons:Feed-forward neural networks We can imagine of adding a non-linear hidden layer of neuron (all activation function needs to be differentiable). This is needed due to the fact that a combination of linear function is still a linear function. Then for learning we just compute the gradient descent. We define as **error-back propagation** a family of methods used to efficiently train artificial neural networks following a gradient-based optimization algorithm that exploits the chain rule. Once the gradient is computed, weights are updated according to standard procedures used in SGD.

Popular types of activation-function

- **Input layer**: linear activation function as identity;
- **Hidden layer**: *sigmoid function*, which has a probabilistic interpretation (co domain between (0,1)). Sometimes people use the *tanh function* due to co domain between (-1,1). Then recently people started using *ReLU*, which is a linear activation rectified. The advantage is that we can always compute the gradient of the function. Then we define the *Leaky ReLU* is similar to ReLU but with a little slope before the threshold.
- **Output layer**: linear function (for regression), Sigmoid (for binary classification), and Soft-max (multi-class).

Considering the ReLU, the gradient is always large for half of the function domain (good for limiting gradient vanishing). For leaky ReLU, the gradient is actually never at zero. However, these activation functions are not differentiable at all input points. There's a discontinuity in $x = 0$ where the left derivative is 0 and the right derivative is 1. It turns out that we could simply arbitrary choose either 0 or 1 as the derivative at this point, and this won't have a serious impact on the final result. 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.

9.2 Loss function

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

$$J(\theta) = -E_{x,y\sim\hat{p}_{data}} log p_{model}(y \mid x) \tag{18}$$

In supervised learning the aim is to estimate a conditional probability distribution. The aim of **maximum likelihood** is to minimize the dissimilarities between the empirical distribution $\hat{p}_{data}$ (defined by the examples in the training set) and the model distribution (the prediction that the model is giving), as measured by KL divergence. This corresponds to using the **cross-entropy** as loss function. However, if the output has linear units, maximizing the log-likelihood (or minimizing the cross-entropy) is equivalent to minimizing the mean squared error:

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

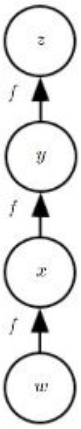
$$MSE_{train} = \frac{1}{m} \sum_{i=1}^{m} \|\hat{y}^{(i)} - y^{(i)}\|^2$$

Where $\hat{y}$ represent the model prediction.

9.3 Multi-class problems

Now we want to generalize the delta rule we have seen to multi-class problems. We imagine that we are vectorizing the input. Then we have a series of hidden layer, and at the end we have output units. THe goal is to obtain the correct output. We have two different phases:

- **Forward propagation phase:** This goes from the input to the output. Here we tried to classify the input.
- **Back-propagation:** We use the MSE to define the error on the output units. Then we adjust the weights.



Back-Propagation Back-propagation 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. In order to analytically derive the gradient, it is convenient to first represent the composite function as a computational graph:

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \tag{19}$$

Error back-propagation Let's assume that we have sigmoid neurons and the loss is mean squared error (no regularizers are applied). Also we focus on a single training pattern using MSE. So the error function is:

$$E = \frac{1}{2}(t - y)^2$$

We decompose it using the chain rule, thus obtaining the partial derivative of error function with respect to each synaptic weigh:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

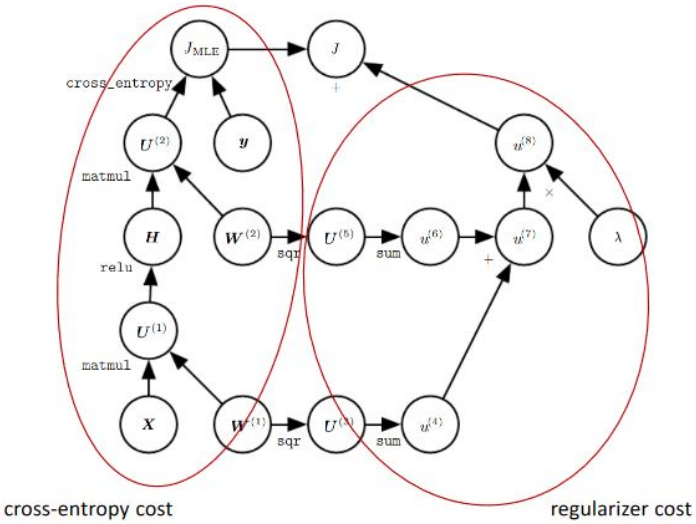


Figura 20: How PyTorch deal with backpropagation

Let's consider the singular term that form the chain rule. First we consider the partial derivative of the weights :

$$\frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj}o_k\right) = \frac{\partial}{\partial w_{ij}} w_{ij}o_j = o_i$$

We obtain the contribution of the input to that neuron (for middle level neurons). For input neurons, this is simply the data. The middle part is the activation function. We take the derivative:

$$\frac{\partial}{\partial net_j}\varphi(net_j) = \varphi(net_j)(1-\varphi(net_j))$$

Finally we consider the partial derivative of the error E over the contribution of the input to the neuron. If we are in an output layer, it's easy:

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(t-y)^2 = y - t$$

For inner neurons, we must consider all neurons in the upper layer L:

$$\frac{\partial E}{\partial o_j} = \sum_{l\in L}\left(\frac{\partial E}{\partial net_l}\frac{\partial net_l}{\partial o_j}\right) = \sum_{l\in L}\left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial net_l}w_{jl}\right)$$

We obtain overall:

$$\frac{\partial E}{\partial wij} = \delta_j o_i$$

Where o_i is the input contribute from neurons, while δ is instead:

$$\delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j)o_j(1-o_j) & \textit{if k is an output neuron} \\ \left(\sum_{l\in L}\delta_l w_{jl}\right)o_j(1-o_j) & \textit{If j is an inner neuron} \end{cases}$$

Input normalization When implement a machine learning algorithm You should do some *data pre-processing*. In order to improve convergence of gradient descent, input signals should be normalized. This way all input features are on the same scale. This only applies to independent features. For this procedure we can use techniques such as:

- **Feature scaling:** 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:** by subtracting the mean and dividing by sigma, we obtain a feature which have zero-mean and unit-variance.
$$x^{'} = \frac{x - \overline{x}}{\sigma}$$

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. However, we must take into account that:

- Being able to represent a function in principle is not equivalent of 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.

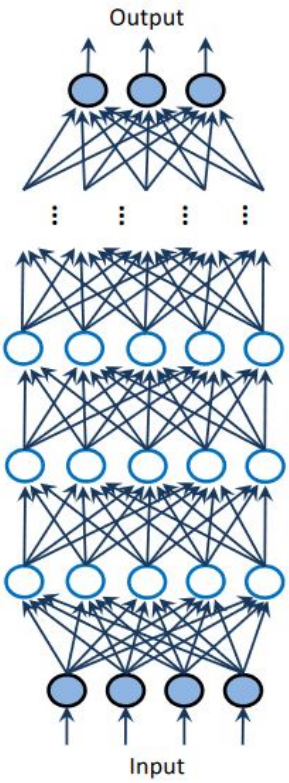
The vanishing gradient problem If we add many hidden layers in some cases the gradient will become vanishingly small, effectively preventing the weights of lower layers from changing their values. In the worst case, this may completely stop the learning process. In particular, when the gradient approaches zero it gets smaller and smaller as we back-propagate it. Some solution to this problem are:

- Unsupervised pre-training;
- ReLU activation function;
- LOT of data and LOT of computing power.

10 Deep Learning

Let's talk about the needs of using multiple layers on a network instead of just adding more neurons. By using multiple levels of representation, where simple hypotheses (basic features) are gradually combined into more complex ones (abstract concepts), we can organize a system hierarchically. This is called function decomposition. This allows to re-use features multiple times to represent many different concepts (knowledge sharing), thereby optimizing information encoding. Moreover, this work better even with limited representational resources.

10.1 Deep feed-Forward networks



We can increase the depth of a network by stacking several non-linear layers on top of each other. Theoretically, this is the same as our multi-level perceptron network. We have an input, hidden layers, and an output. Learning is still based on gradient descent and errorback-propagation:

$$\nabla_\theta J(\theta)$$

However, we must be aware of some problems. The first is **gradient vanishing**: the learning signal gets weaker as we move away from the output. The second problem is that having such a large-scale system where every weights is a parameter can lead to overfit. As we increase the capacity of the problem, we should also increase the size of the training data. To solve the gradient vanishing problem, we can:

1 Unsupervised pre-training: pre-train the network using generative models. The idea is that the signal is coming also from the input, and not only from the output;

2 To avoid saturation of activation function we replace them with ReLUs.;

3 Have a lot of data. This can be mitigated from the use of Gpus;

4 Tricks for improving the convergence Stochastic gradient descend (momentum, adaptive learning rates).

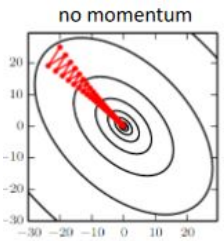
The solution to the second problem is using regularizers. Let's examine the tricks to optimize our network.

Weights initialization Back in the days the weights were assigned using a standard distribution, or sampled from a certain distribution. This is good for Good for symmetry breaking, but requires a lot of computational data for the SDG to converge. We can do better if we employ some simple heuristics that will favour SGD by avoiding numerical problems in gradient computation. One idea is that larger weights help in breaking the symmetry. However, if too large they can saturate the activation function. A good compromise is the *"xavier initialization"*. For initializing the weight matrix, we are taking a uniform distribution from this intervals:

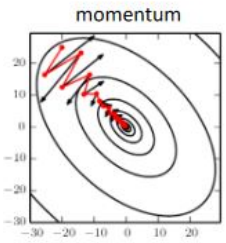
$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \tag{20}$$

The idea is that for each layer we have a certain amount of inputs, and some neurons in the output. If we initialize the weights in this range, we are taking in account the number of connections. So if we have more connection coming, we'll reduce the interval, thus avoiding saturating the function. This is the initialization used on the Homework $\#_1$. We can also use the sparse initialization. Each unit has exactly k nonzero weights. This allows to control the total amount of activation as we increase the number of units, but without making weights too small as m and n increase. Another useful trick is the use of

Momentum The idea is that adding a momentum term helps accelerating gradient vectors in the right directions, thus leading to faster convergence of SGD. This is especially useful when the Hessian matrix is poorly conditioned. In analogy with Newtonian dynamics, it takes into account a velocity term v representing the direction and speed at which the parameters are moving in the optimization space. The velocity is measured using an exponentially decaying moving average of past gradients, and added to the current gradient estimate:



$$v \longleftarrow \alpha v - \varepsilon \nabla_\theta J(\theta) \tag{21}$$



Where ε represent the learning rate. αv represent the accumulated gradient, while $\varepsilon \nabla_\theta J(\theta)$ is the current gradient. In particular, $\theta \longleftarrow \theta + v$. Note that momentum and learning rate are intimately connected!

Adaptive learning rates The final trick regards the use of learning rates. The learning rate is probably one of the most critical hyperparameters to set in SGD. It can be set with different criteria:

1 As a constant (usually small value, like 0.01). This is not particularly efficient, since it takes a lot of time for the SDG to converge;

2 Logarithmic decrease as a function of epoch number, as seen in homework $\#_1$;

3 Adaptive learning rate. This is the most efficient criteria by a large margin.

The idea behind the **Adaptive learning rate** is that its value is set accordingly to the current (and often to the past) gradient values. The main goal is either to move quickly in directions with small but consistent gradients, or to move slowly in directions with big but inconsistent gradients. An important difference from the momentum is that we user a different learning rate for each connection weight. This is good since not all the weights are impacting a certain node as much as others do. What the best adaptive learning rates schemes (like AdaDelta) do is dividing the learning rate of a certain weight by a running average of the magnitudes of recent gradients for that weight.

10.2 Model regularization

For regularize a model, we need to introduce a regularization term. The model capacity is limited by imposing a norm penalty in the parameters:

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

The hyperparameter α defines the relative contribution of the penalty term to the overall loss function. Usually only the weights (and not the bias) are regularized. We can use different penalty terms for producing different effects. For example:

- L^2 norm bring to the weight decay. This is used in Ridge regression. We are trying to maintain small weights for the parameters of the regression.

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

- L^1 norm bring to coefficient sparsity. This is used in lasso regression. The idea is to have only few active weights (or activations).

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|$$

Another very popular approach for regularize model is called

Dropout Here we are not changing the loss function, but instead adding a penalty term to the weights. We are taking the fully connected network and we are randomly removing a different subset of neurons and their connections. We end up with a much sparser connectivity, but the information still flow. This approach has a very efficient implementation (just need to multiply the output activation by zero): each time we load a training patter, we randomly sample a binary "mask" which defines which units should be dropped out according to a fixed probability p. The idea is that we are regularizing the behaviour of a single neuron. We want the neuron to extract a feature that is useful in many context.

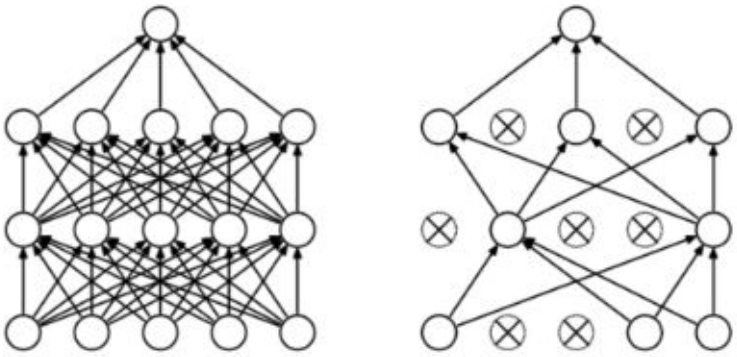


Figura 21: Example of the dropout approach

10.3 Hyperparameters tuning

A good way to search for the best value of a certain hyperparameter it's the **grid search**. Here we take all the possible combination of two hyperparameter. This is good if we have a limited number of hyperparameters. We systematically (but not exhaustively) explore the hyperparameter space. A more efficient way to find the best value is to use a **random search**. Rather than fixing certain interval values for the hyperparameters, we have a randomized layout. The advantage is that if there with grid search we could miss the best value. With random sampling instead we could pick up a better parameter by chance.

Another (not so) common technique use **second-order optimization methods** due to the fact that is computationally intensive. The idea is that second derivatives give information about the curvature of the function and can be very useful to speed-up SGD. There are 3 different scenario that are represented in figure [22]:

- Negative curvature: the cost function decreases faster than the gradient predicts;
- No curvature: the gradient predicts the decrease correctly;

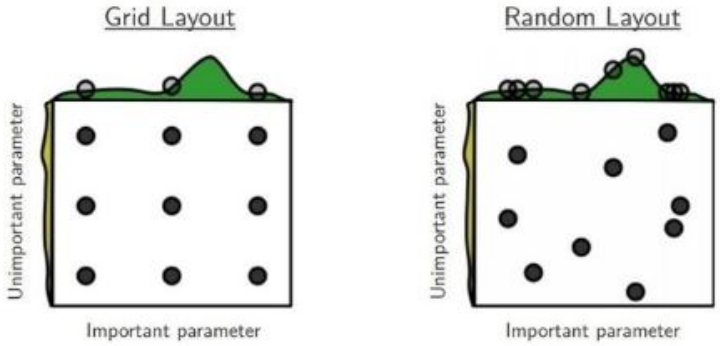


Figura 22: Example of grid search vs random search

- Positive curvature: the function decreases more slowly than expected and eventually begins to increase, so steps that are too large can actually increase the loss function.

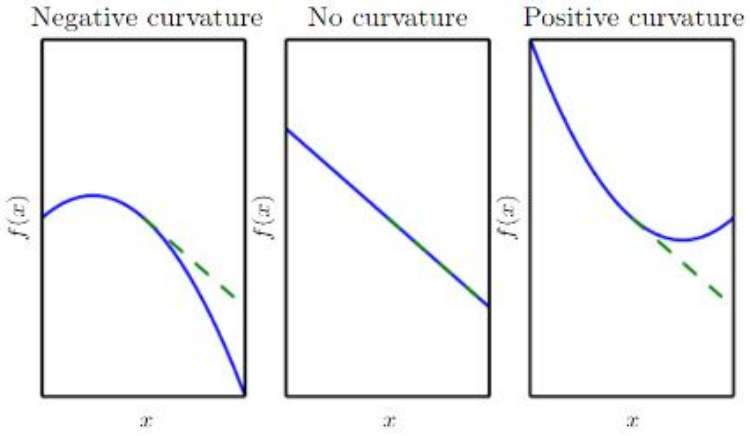


Figura 23: Example of second derivatives of SDG

While the first order derivatives are collected in a Jacobian matrix, to collect the second order information we use the Hessian matrix. This is the Jacobian of the gradient. Given a term of the Jacobian matrix $J_{i,j} = \frac{\partial}{\partial x_j} f(x)_i$, the Hessian matrix is then composed by

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

Such terms then form the Hessian matrix:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_j} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_j} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_j^2} \end{bmatrix}$$

We can then calculate the *condition number* of a matrix, that is the ratio of the largest eigenvalue to the smallest. The condition number of the Hessian measures how much the second derivatives in a certain point differ from each other. The Maximum eigenvalue express the maximum second derivative, while the minimum eigenvalue express the minimum second derivative. When the Hessian has a very large condition number, gradient descent performs poorly: in one direction the derivative increases rapidly, while in another direction it increases slowly.
To compute the Hessian matrix, we can use different functions:

- Newton's method: uses information from the Hessian matrix to guide the SGD. If the loss can be locally approximated using a quadratic function (with positive definite H), we can directly jump(no ε!) into the minimum by rescaling the gradient by H^{-1}:

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

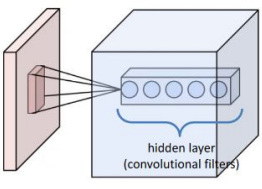
  We may occur in some problems, like the fact that the loss function surface in deep networks is usually nonconvex. In particular, there are many saddle points that makes this approach infeasible. Moreover, the size of His quadratic with respect to the number of parameters in the model, which is a serious problem for large-scale neural networks;

- Conjugate gradients/BFGS: similar to Newton's method, but implement a more efficient way to compute the inverse of the Hessian. For example, BFGS approximates H^{-1} with a low-rank matrix that is iteratively refined;

- Hessian-free optimization: relieve from the burden of computing (or even approximating) H^{-1}, and resort to finite-difference methods to approximate the product between H^{-1} and the direction vector.

11 Convolutional network

This architecture always has a visual input (like images), and 3-dimensional hidden layers. The input is a *large scale* input. We can have many input channels. The idea behind the convolutional layers is that we exploit parameter-sharing to improve the computational efficiency: the same neuron (also called filter/kernel) cover a specific portion of the input, so it's not fully connected. Then the pooling layer shrink the size for efficiency and noise reduction. Recent papers suggest that pooling could be not mandatory. Then we can replicate those procedures as many times we want. The pooling layer is then flatten in certain vector. Then we have a standard classification network. The idea is that in a fully connected architecture every neuron is taking input from every other neuron. Vice versa, in a convolutional architecture we have the assumption of local correlation. The idea is that the correlation in an image are *local*. This usually holds for sound and images.



3-Dimensional layers Let's explain why we use 3-dimensional shape to describe a convolutional network. Each hidden neuron has a local receptive field encoding a specific feature. SO we have an *activation map* over each hidden neuron. As a result of the convolution, each filter will produce a bi-dimensional map. The number of hidden neurons still defines how many features (kernels) will be represented at each processing layer. In particular, we have some parameters that define how the convolution goes:

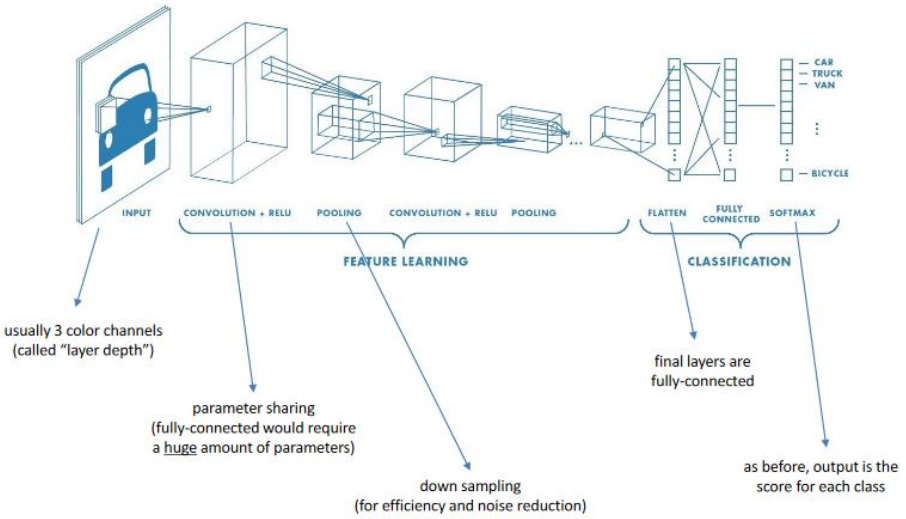


Figura 24: Convolutional network structure

- **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 zero-valued inputs surrounding the image (preserve image size).

Convolution Let's examine how we apply the convolution over one channel. WE have a matrix (filter) which is applied to another matrix. For each element in the filter we do an element by element multiplication and then we sum everything to obtain one value which is the result of the convolution. Then I move by the stride parameter and repeat the procedure until all the image is covered.
In particular, different kernels generate a different output (feature map) from the same original image. An example can be seen in picture [26]. However, usually in input that are not black-and-white we have many channels (for exampes RGB images). We say that the kernel has a parameter called **depth** dimension. Successive convolutional layers operate over stacks of feature map. In particular, every convolutional filter is applied over all the channels. Every kernel has dimension $k_1 \times k_2 \times n$.

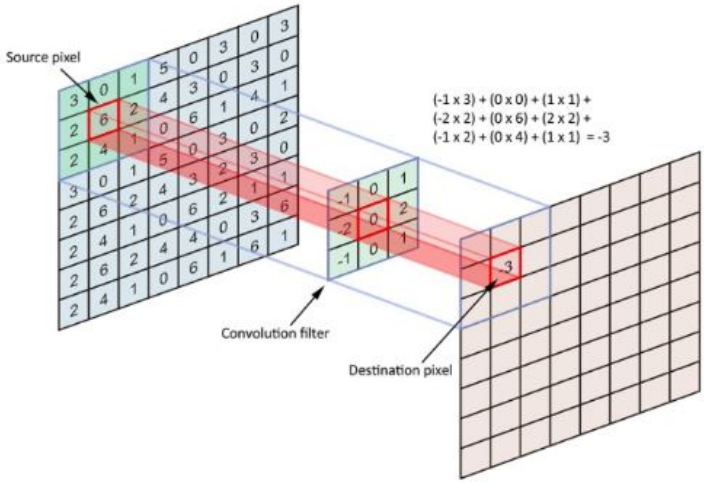


Figura 25: Example of convolution

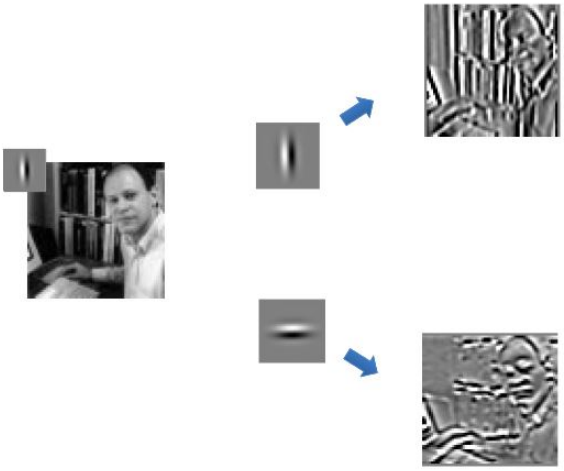


Figura 26: Use of different kernels over the same map

For example, a 3x3 kernel has different weights for each RGB channel: the final feature map for that kernel will be created by combining the convolution over each channel. This can be seen in picture [27] This will result in m feature maps, one for each kernel.

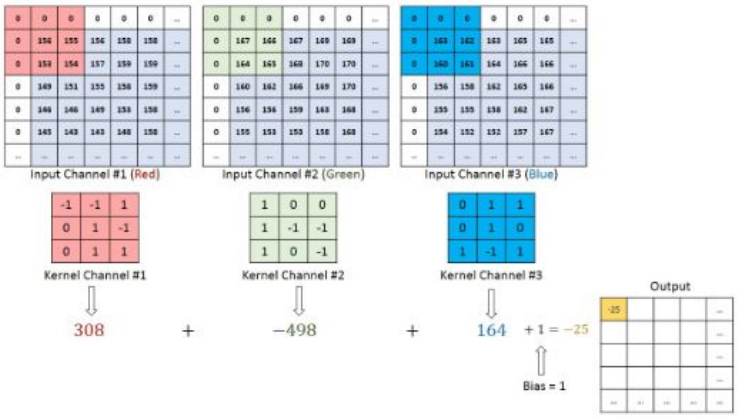


Figura 27: 3x3 kernel has different weights for each RGB channel

11.1 Max pooling

We end up with a series of bi-dimensional maps. When we do max pooling we maintain the same depth but we reduce the resolution of the map. This is useful for reducing the number of parameters (and the computation burden), controls overfitting and builds invariances. This seems like we are losing a lot of information, but in large scale image we can still the content of the image after the pooling. There are many forms of pooling(max pooling, average pooling).

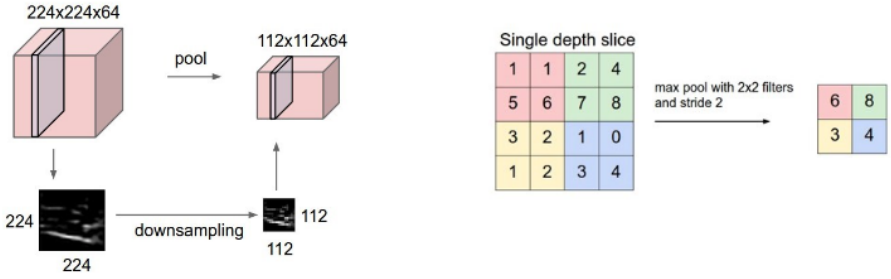


Figura 28: Example of pooling of a convolution

Note that we could avoid pooling and just use stride parameter.

11.2 Popular convolutional architectures

Let's touch some of the most popular architectures trough tout the history.

LeNet-5 Published by LeCunet al. in 1998 in the *Proc. of the IEEE*. Had 4 layers (3 convolutional, 1 fully-connected), it was the state-of-the-art digit recognition accuracy (at that time) for small grayscale images.

AlexNet Published by Krizhevskyet Alex in 2012 in *NeurIPS*. It had 8 layers (5 convolutional, 3 fully-connected); It also used several tricks to improve SGD. This was the state-of-the-art object

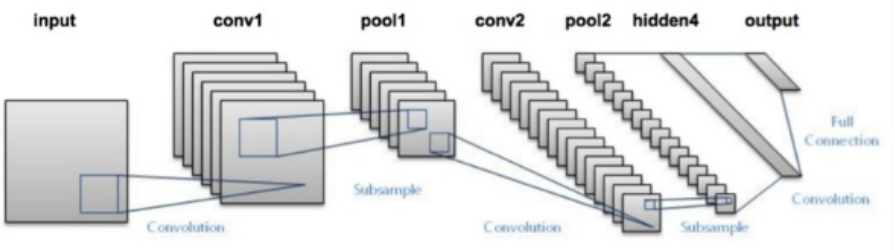


Figura 29: LeNet-5 structure

recognition accuracy (at that time) in fully-sized real images, leading to a dramatic improvement in performance of computer vision systems. The use of GPUs was also implemented here.

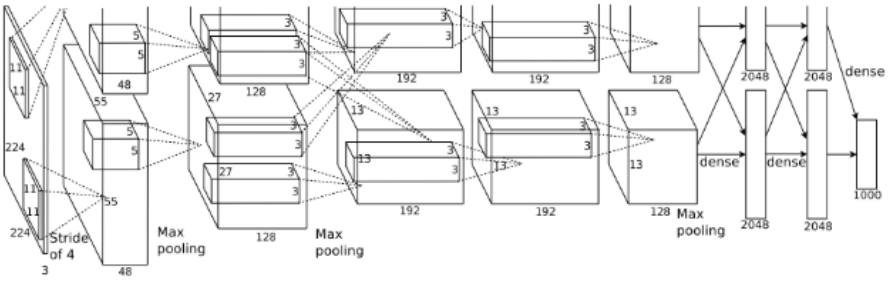


Figura 30: AlexNet structure

GoogLeNet Published by Szegedyet al. in 2015 in *CVPR*, **GoogLeNet** was the first to introduce the **inception layer**, which allowed to stack together 22 convolutional layers. It also introduce 2 auxiliary classifiers to limit gradient vanishing. You also feed the classes at intermediate step so that you can improve the gradient in early layers.

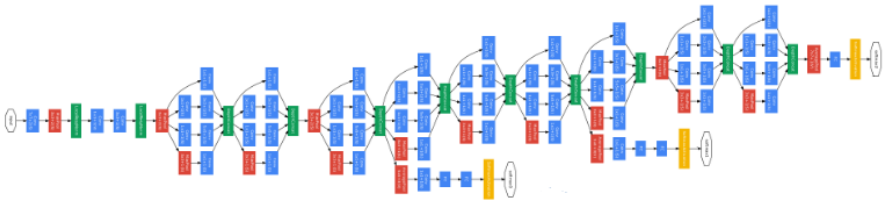


Figura 31: GoogLeNet structure

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

Residual Network (ResNet) Published by He et al. in 2015 in *CVPR* , feature 152 layers. We know that adding too many layers is very problematic due to gradient vanishing. The idea is that having more layers shouldn't have worse performance. To make the architecture more

flexible is to include*skip connections*. By summing to the result of the convolution the original input, we can evaluate if the layer is needed or not by looking at the results. The architecture of the network is much more flexible: during training, layers that are not useful for reducing the loss are skipped. This idea has been further extended for creating *Densely connected CNNs* (DenseNet).

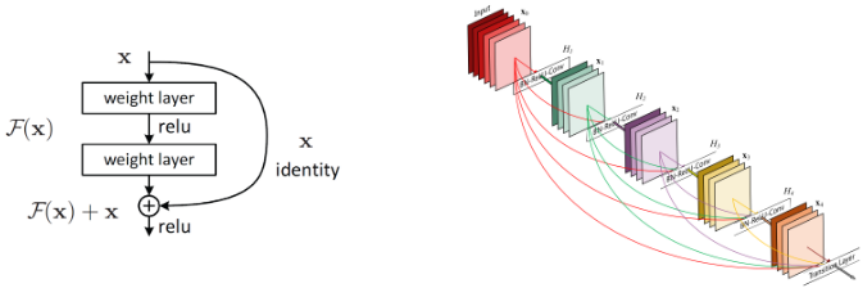


Figura 32: ResNet structure

11.3 Transfer Learning

In a **transfer learning** setting we are learning something from one distribution and we are assuming that is overlapping in some degrees with another distribution. Rather than learning each task from scratch, we simply "fine tune" the high-level features characterizing each task. This benefits both in terms of accuracy and learning time. For example, if we want to implement a machine vision system we could recycle the visual knowledge already learned by some state-of-the-art deep network, by training only the last, fully-connected layers. To implement this, we just set to zero the gradient for all the weights and change the gradient only for the layer we want to tune. One example of how many layers we decide to share/specialize is in figure [33]. This is similar to how our brain works, whereas different areas of the brain specialized on certain aspect are sharing information.

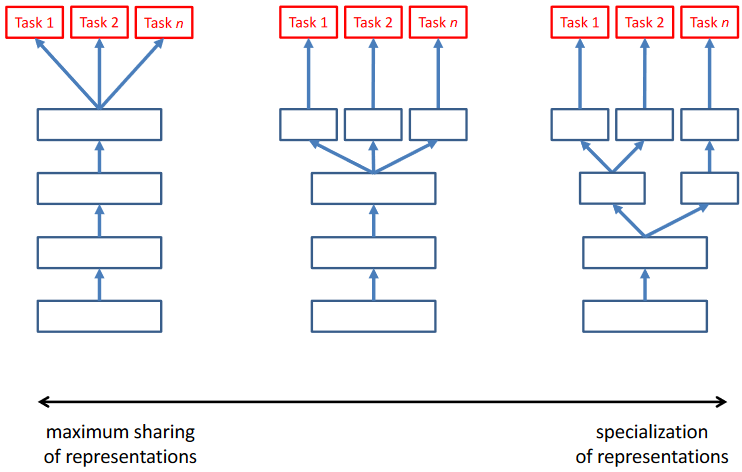
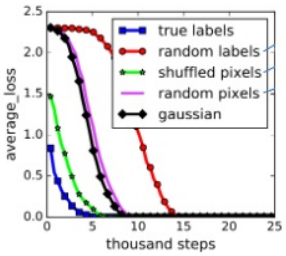


Figura 33: Transfer learning possibilities

11.4 Shallow vs Sharp minima



We are wondering why certain architecture works well and can generalize properly. Some possible answers could be due to the use of regularizers, or the distributed representation. However, recently it was found that taking a deep network and have it fit a random labelled dataset will still fit the training set really well. Instead, the test set will totally fail, even applying regularization. Thus, regularization may not be the cause of why certain neural network work well. Even replacing the true images with unstructured random noise (there's no feature on the data) the network will fit the training set well. Thus this might not be the cause of good performance either. Even taking not optimal data we'll still converge. One possible explanation might be that the number of parameters in a neural network often exceeds that of data points, allowing to accurately memorize the entire training set even if regularizers are applied. Also, optimization are still possible even if there is no relation at all between data and labels. There are no actual explanation about the generalization, since architectural properties and regularization methods are not sufficient conditions. One hypothesis is that Stochastic Gradient Descent acts as an implicit regularizer: SGD has some noise, since we are following the gradient of the "minimatch". The idea is that if we have a very sharp minimum, the SGD will not find it and will instead find a minimum that has a larger base of attraction. Flatter minima would lead to better generalization, while Sharper minima would lead to overfitting. Therefore, SGD tends to settle into flatter minima due to the noise in gradient estimation.

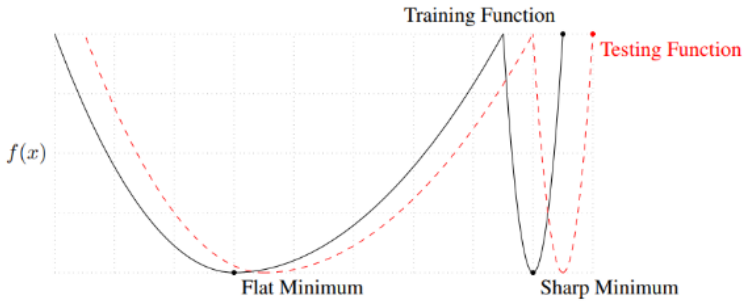


Figura 34: Sharp vs Local mimima

11.5 Adversarial samples

It was discovered that you can "fool" (misclassifying the input) a deep network by injecting some noise in the image. Noise has to be created carefully: the image should be modified in the direction of the gradient maximizing the loss function with respect to the input image. For a good adversarial sample to be good, for a human the stimulus should not distinguishable from the original one. The adversarial learning is also not tailored by the deep network we are "attacking" but can trick other network as well, and in some cases even support vector machines. Mathematically, it can be figured as

$$X^{adv} = x + \varepsilon \cdot sign(\nabla_x J(x, y_{true})) \tag{22}$$

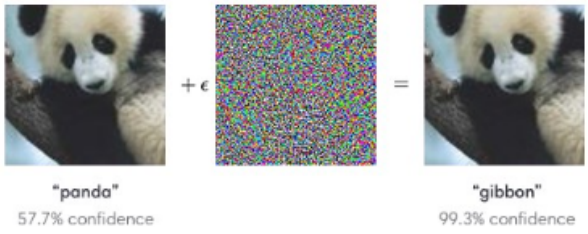
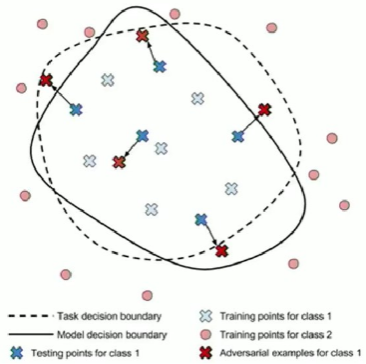


Figura 35: Example of adversarial learning



This vulnerability is challenging the use of deep learning in sensitive contexts, like self-driving cars or facial recognition. We can imagine the training and testing points in a 2-Dimensional space. What the deep network is doing is to find a decision boundary that collect all the points in it. But maybe the true model is a bit different. Even if the network classify well the test points, there could be some points near the test points that are just outside the true boundary. By adding a slight modification to the test points, we can make the new sample going outside the true boundary while remaining in the network model boundary.

Defends against adversarial attacks Ways to defend against adversarial learning are still being explored. However, some techniques that has worked so far are:

- **Adversarial training:** The neural network is optimized via a min-max objective to achieve high accuracy on adversarially-perturbed training examples. We are basically creating a new network to create adversarial attacks that we are then injecting into another network so that it learns to recognize that as well;
- **Randomized smoothing:** the neural network is smoothed by convolution with Gaussian noise

$$\hat{f}_\sigma(x) = E_{\varepsilon \sim N(0,\sigma^2 I)}[f(x+\varepsilon] \tag{23}$$

where f is the base network and $\hat{f}_\sigma$ is the smoothed network whose predictions for the input x are obtained by averaging the predictions in neighbouring points, weighted according to an isotropic Gaussian centred at x with variance σ^2.

11.6 Pruning and quantization

Now we examine how can we reduce the number of connection weights in a network without increasing its generalization error. We know that large-scale deep nets are usually required to solve difficult classification tasks. However, it was discovered that we can greatly compress (reduce the number of parameters) a deep network without losing final performance. This has to be done *during* the training. Starting with a network with low connections will fail, but using a network with a large number of connection will work. To do this, there are various options. For example, one of those works like this:

1 Train the network;

2 After each weight we remove the weights below a certain threshold;

3 We re-train the network;

4 Repeat [2] and [3].

5 Quantize (encode) weight values from 32 bits to 8 bits.

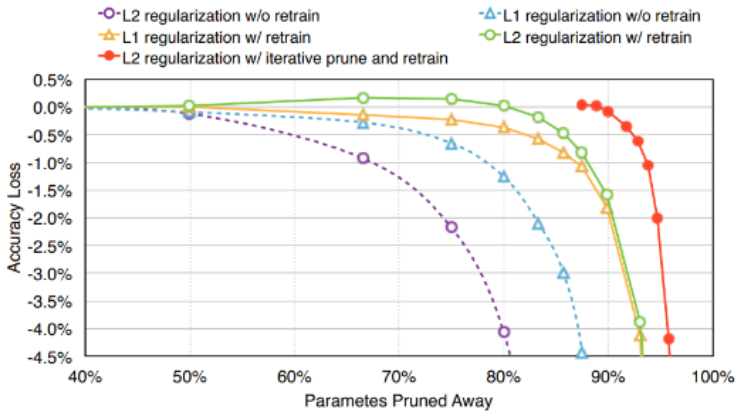
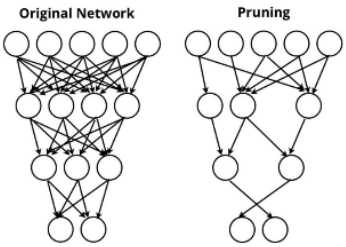


Figura 36: Pruning performance

Thus, Neural Network seems quite sensitive to initial condition. however, this is still very empirical. In figure [36] it shows how different network perform regarding certain percentages of the weight pruned. We can see that almost 90% of the weights connection can be proned before starting to lose performance.



11.7 model interpretability

State-of-the-art machine learning models, such as deep networks, are notoriously hard to interpret. This is the issues of **model interpretability**. Those are often called "black box models" due to the difficulty in explaining what's exactly happening inside. As they get more and more diffused in real-life applications, more pressure is being put in building *explainable models*. This is particularly evident for critical applications like military technology, self-driving cars, clinical diagnosis and medical support. Sometimes, we prefer a simpler (and less accurate) model that can be interpreted, rather than a high-performance (but mysterious) deep network. One way to interpret the internal code of a deep network is to visualize the features learned by it. This is mostly effective when image are input. Imagine we have a network with 2 layer where the input is a vectorized image. I want to understand what a specific neuron is doing. As in a multiple linear regression, the weights corresponds to the importance of the connection. We can simply rearrange the input and plot the weights according to some grey scale, for example (see picture [37]). We can see how the neurons are specializing to determine a specific type of features. When we stack together many layers, we just have to extend this idea. We can combine linearly (for example) the weights in

all the layer. By simply combining the strength of the connections to the unit, we obtain the features in layers below the first.

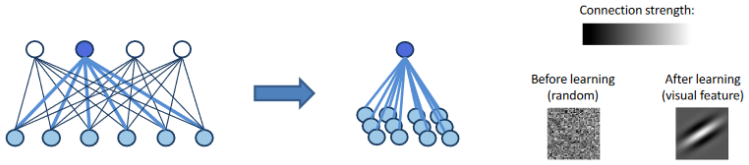


Figura 37: Visualization of neuron feature

One example on the MNIST dataset can be seen in picture [38] Here we are linearly combining

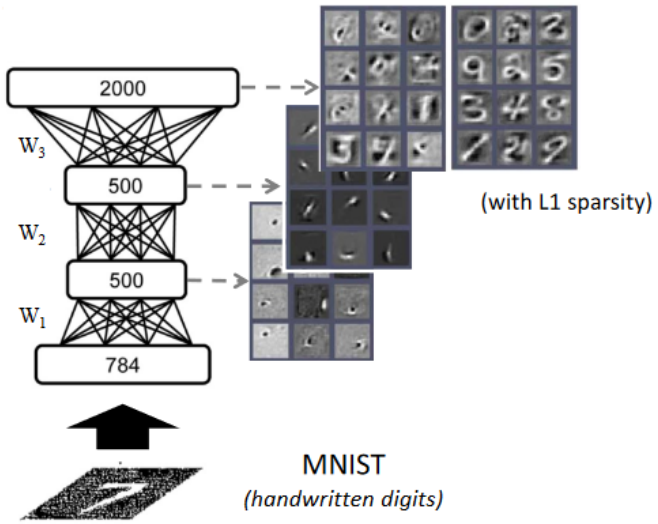


Figura 38: Visualization of MNIST feature

weights without considering that the activation function is non-linear. Thus, this approach don't scale in complex network. We know that features get increasingly more complex as we move deeper in the network. A possible solution is to generate a synthetic image that maximally activates a certain neuron. We start by a random sample in the input, and we gradually change it following the activation gradient in order to maximize the response of the neuron.

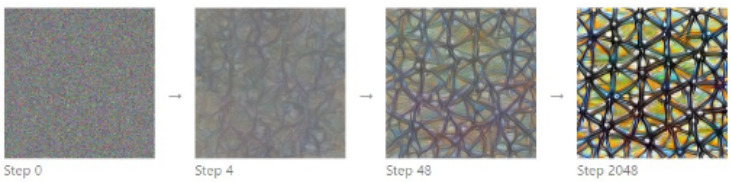


Figura 39: synthetic image that maximes the neuron

Then we can look at the examples that mostly activate the neuron to see how the resulting feature is related to the image content

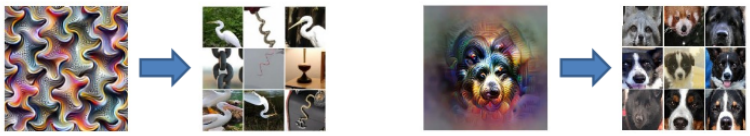


Figura 40: Correlation between synthetic images and inputs

12 Recurrent neural network

To simulate how our brain process sensory stimuli, a simple neural networks is not enough. We need to somehow introduce a correlation between past and future sensory states. This is related to the concept of **predictive processing**. THe idea is that you can somehow predict the future to optimize information encoding. This allows to consider contextual information in the temporal dimension during inference and learning (*top-down processing*). This also allows to generate plausible sensory information even when there is no stimulation at all. We can just predict the sensory based on the past before receiving any input stimulus. In general, the idea is that we can exploit some learning to incorporate the structure of the environment and predict it. All of this is made possible by the presence of feedback connections. This is similar to the back propagation connection.The difference is that those backward connections doesn't really carry the error of the classification, they can carry also other kind of informations. The basic idea behind the **Recurrent Neural Network** is that we can exploit a set of time-delayed, feedback connections to store past information in the hidden layer. Thus, we have a *context layer* besides the inner layer. The input is no more a static vector, but rather a time series of ordered vectors, while the model now behaves like a dynamical system. The activation of the hidden layer is a function that depends on previous activations, the current input and the time-step.

$$h_t = f(h_{t-1}, x_t, t)$$

The output is still a function depending on the last input and the time-step.

$$o_t = h(h_t, t)$$

Ideally, we want the final time-step to have a context that is taking into account all the previous states. We have to greatly compress the information to represent it on the last layer. The easier model of RNN is the so called **simple recurrent network**. Following each input, the activation of hidden units is copied to the context layer. The activation at the next time step is however influenced by the context layer. Processing is similar to a feed forward network, but we have a set of delayed connections which are simply copying the inner units delayed by one time-step. In the following layer, when we process it, we also incorporate the influence of the previous time-step saved on the context layer. For example, regarding the RNN in the picture [], the activation of the unit depends of the input and the previous time-step:

$$h_t = f(W_1 x_t + W3 h_{t-1})$$

The output is just depending on the current state of the last layer:

$$o_t = g(W_2 h_t)$$

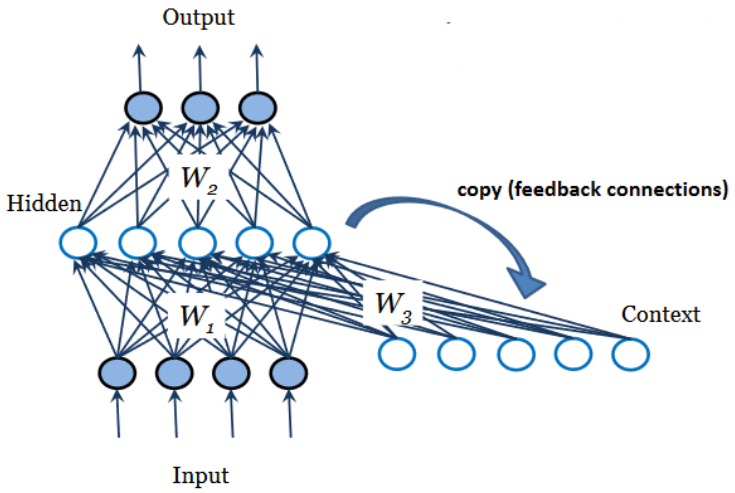


Figura 41: simple recurrent neural network with 1 hidden layer

This architecture can easily be extended to *unsupervised learning*. Rather than producing an output, we could learn to simply predict the next input.The task is to take the input at time t and produce an output at the time $t + 1$.
One of the first implementation in 1990 was trying to find a of structure in time, by Elman. The Training set was a set of letter sequences forming English sentences, with the task of learning to predict the next element in the sequence. Elman discovered that the network was able to capture the correlation between the letters. The network was also able to discover lexical categories. Hidden activations (internal representations) corresponding to each word were analyzed using hierarchical clustering. The network clustered the words according to their usage (for example based on the surrounding context), discovering several major categories (as verbs, nouns, animals, foods, ...) The statistical structure in the dataset caused the activation of the hidden units in such a way they were clustered together.
How can we scale this to make it more complex and efficient? By unrolling the computational graph over time, we can transform the recurrent architecture into a **directed acyclic graph**. We assume an initial state, and then it the process the sequence in a way where every time-step has a corresponding hidden layer activation that gets propagated. By removing the cycle, we can analyse the graph more easily.

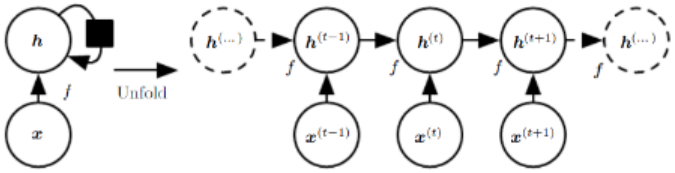


Figura 42: Unfolding the cycle

We can even represent it from bottom to top. We can see that tit's becoming a very deep neural network with a feed forward structure. A temporal network can thus be seen as a very

deep network with parameter sharing. Rather than computing the gradients only at the end of the unrolling, we can break down the sequence into shorter pieces (truncated BackPropagation Trough Time). The main difference with standard neural network is that weights are *shared*. We are constraining the weights of the network in order to use the same matrix at every time-step. A single matrix is needed to process sequences of difference lengths with the same architecture. We have transformed a sequence problem into a spacial one. We can have different sequence

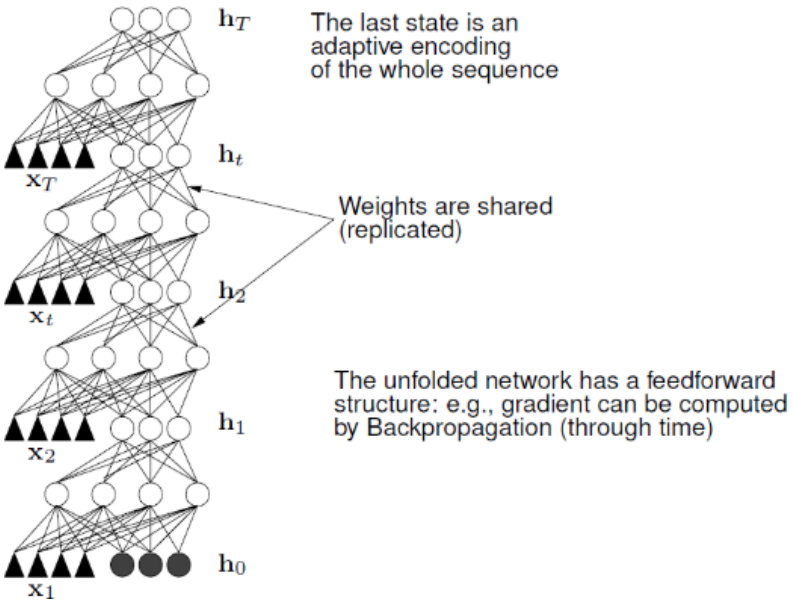


Figura 43: feed-forward structure of a RNN

problems. We ahve an input, some hidden vectors with delayed connections and an output. This is used to compute the loss with some target labels. The desider output can be different things:

- *Sequence-to-sequence*: each input is followed by a prediction(the sequence to be predicted could be the input sequence itself);
- *Encoder-decoder*: seq2seq model where the entire input sequence is first encoded into a fixed-length vector, and then the entire output sequence is generated. This tries to compress an entire sequence, and then we decode it. This is useful
- *Sequence-to-output*: one single output is produced, at the end of the sequence. We process the sequence until a final representation and then it gets reduced to one output. (i.e spam, emotions labelling)

Universal approximation properties There are some theoretical properties that follows the definition of RNN. Back in 1992, Siegelmann and Sontag demonstrates that for any computable

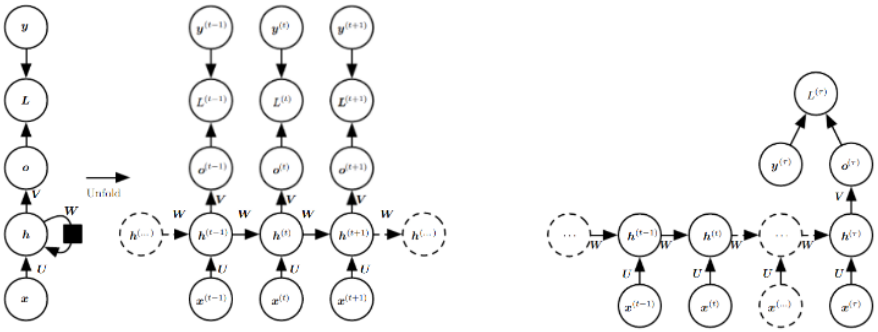
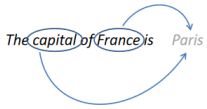
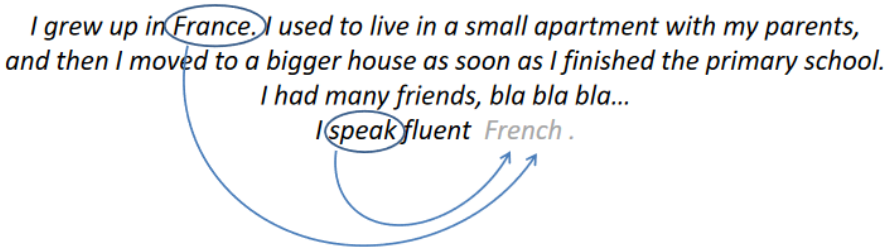


Figura 44: Variety of sequence learning regimens

function, there exists a finite recurrent neural network that can compute it. The basic idea of a the proof is that it is possible to use a recurrent network to simulate a pushdown automaton with two stacks. Therefore, our RNN could be equivalent to a Turing Machine. (can solve any problem, no time constraint) The assumption we must have is that the neuron's state must be a rational number with unbounded precision. In 1993, Funahashi and Nakamura showed that any finite time trajectory of a given n-dimensional dynamical system (a sistem evolving trough time) 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. Finally, in 2006 Schafer and Zimmermann demonstrated that any open dynamical system can be approximated with an arbitrary accuracy by a recurrent neural network defined in state space model form. However we must note that for the case of feed-forward networks, these theoretical results do **not guarantee** that recurrent networks can in practice achieve such result.

The problem of long-term dependencies Simple Recurrent Network can effectively learn short-term temporal dependencies, where all the context is in few words. The correlation between the state of a certain time point and the state of the previous time point can explain the result. However, with long term dependencies, we need to predict something that was mentioned a lot of time before. A possible solution is using a **Long-**





Short Term Memory networks(LSTM networks). This is a much more complex architecture. Here we add to the basic RNN architecture a set of gated units, which will learn how much information should be retained in temporary memory and how forward it should be propagated.
The key points is the use of gates. Those gates take an input, analyse it and then combine it in order to obtain the new cell state.

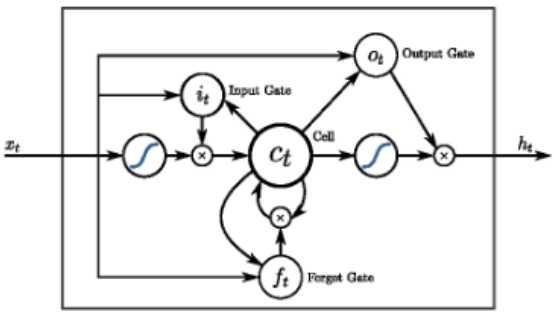


Figura 45: Scheme of a LSTM

- **Input gate**: let the input flow in the memory cell;
- **Output gate**: let the current value stored in the memory cell to be read in output;
- **Forget gate**: let the current value stored in the memory cell to be reset.

In particular, each gate has its own set of synaptic weights. A more principle representation is shown in picture [46]. Let's now analyse it step by step. we have the cell state which is simply the

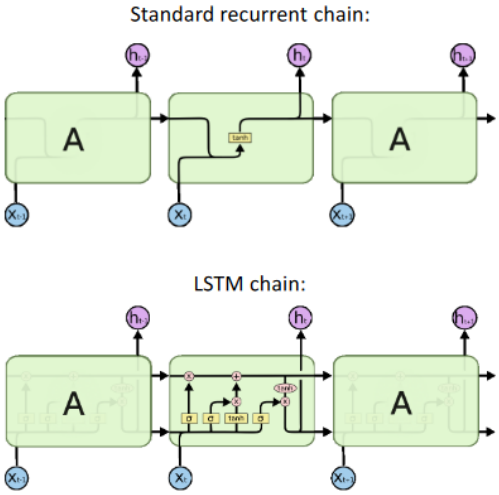


Figura 46: standard recurrent netowrk vs LSTM network

context which is being propagated into the future. We want the context to be as rich as possible. Then we have the gates, which can alter the cell state by adding or removing information through sigmoid and multiplicative connections. Those usually have a sigmoid pre-processing. There are multiple gates, as we have seen. The forget gate determine what information we're going to throw away from the cell state:

- 0: forget everything;
- 1: remember everything.

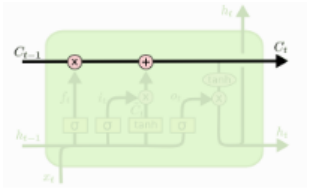


Figura 47: cell state

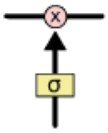


Figura 48: gate

This is defined as

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

Then we have the input gate, which determine what new information we're going to save in the

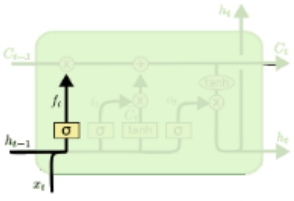


Figura 49: forget gate

cell state:

- i_t: which elements to store;
- $\hat{C}_t$: values to store.

In particular, we have

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_u)$$

and

$$hatC_t = tanh(W_C[h_{t-1}, x_t] + b_C)$$

The cell update just multiply the old state by the forget factor, and add the new input information.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t$$

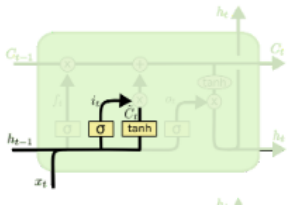


Figura 50: input gate

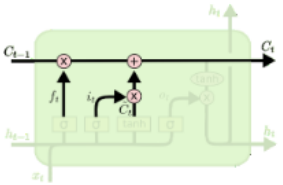


Figura 51: cell update

Finally we have an output gate which decide what propagate forward in the output. The output will be a filtered version of the cell state:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

where

$$h_t = o_t \cdot tanh(C_t)$$

This is the most typical, but there are many variance of the architecture. For example, we can

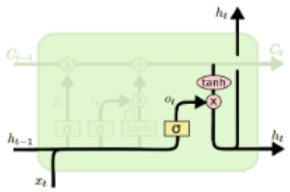


Figura 52: output gate

add peephole connections to the context layer before any manipulation. This is because knowing how the on text was before modified was useful.

Word embedding In many practical Natural Language Processing (NLP) scenarios, we are not interested in working at the level of single characters. Instead of modelling sequences by encoding every single characters, we just encode single words as fixed-length vectors using word embedding algorithms. There are many embedding algorithms, some of them sue probabilistic methods based on user-defined language model, other use dimensionality reduction methods based on statistical co-occurrences. More recently they use re-trained character-level recurrent neural networks, which can produce in the hidden activation a fixed vector for every word.

Attention / Transformer networks The issue we have is that as we increase the sentence's lenght the translation quality start to go down. This can be measured using the BLEAU score (see picture [53]) Seq2seq models have troubles in encoding long-term dependencies, because

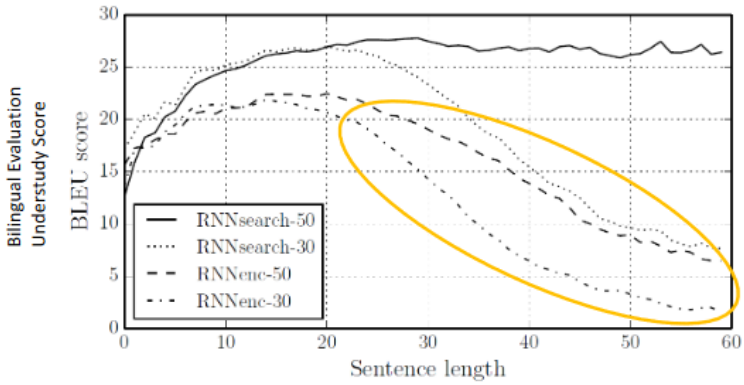


Figura 53: BLEAU score over sentence length

they map an entire sequence into a fixed-size vector. To solve this issue, we can use **attention**. With attention, we selectively use past information during decoding, by taking a weighted sum of all the encoder inputs so far and pass it into the decoder hidden state. Let's consider for example a words translating algorithm. A classical seq2seq model is just processing every input sequentially and propagating the hidden state, then propagate the hidden state and generate the output sequence. If we include attention in the model, the encoder is passing all the hidden states to the decoder. The idea is that if we can more explicitly consider the hidden states, we can also consider longer terms dependencies. At each decoding step, the decoder is giving to each hidden state a *score*. This is used to amplify the most relevant hidden state. It's like having different weights for every state. This will constitute an additional context vector, used to generate the output.

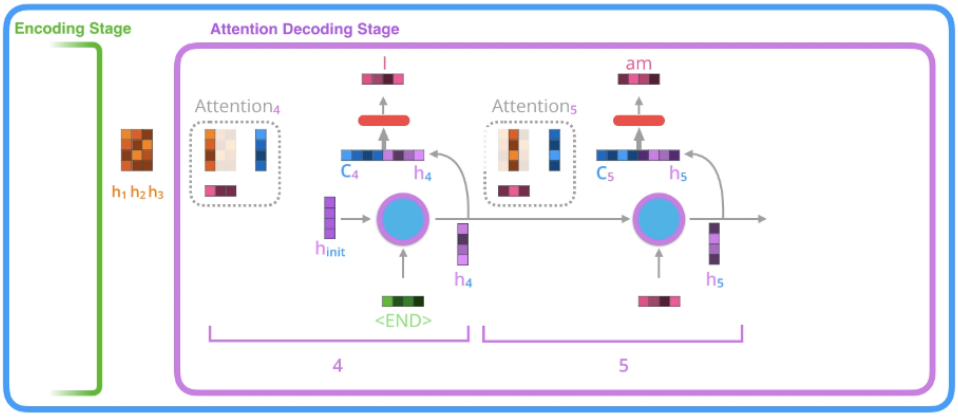


Figura 54: Example of translation using attention

We can see that the attention component of the network is able to selectively amplify re-

presentation corresponding to previous elements of the sequence. Some state-of-art examples are

- **GPT:**(generative pre-training) created by OpenAI, here the idea is to train the model on a huge set of sentences and then fine-tuning on specific discriminative tasks.
- **BERT:** (Bidirectional Encoder Representations from Transformers) created by Google, uses a transformer architecture with bidirectional encoding and decoding, with the goal of considering also the "future" into account.

13 Unsupervised learning

In **unsupervised learning**, learning does not require any labelled data. This implies that the learner can exploit the huge amount of raw information contained in its environment. Once an internal model of the environment has been learned (also known as *representation learning*) it can be effectively used to more easily learn also supervised tasks. Also, from a psychological/cognitive standpoint, it seems quite plausible that children and animals massively exploit this learning modality during development. Some of the major shortcomings of unsupervised learning are the choice of what features of the environment will be useful for solving later tasks and how to select them. Also, unsupervised learning can be computationally demanding. On top of that by mere, passive observation, we cannot infer causal relationships. Having a good representation is important: we can learn a supervised mapping directly from the data, or better we can first extract useful descriptive features and then learn a mapping from these "higher-level" internal representations.
In the sensory space, objects are not linearly separable. Visually however they might have some "patterns" in common. We have to somehow extract some meaningful information from the images. How can we make this patterns separable?

- In supervised learning we just use a lot of labelled examples;
- In unsupervised learning, we do an unsupervised feature learning and then a (linear) *read-out*.

The underling assumptions is that in high-dimensional data sets the data always lies on some lower-dimensional **smoothly curved manifolds**. A manifold can be imagined as a surface with high dimension where every point is connected to a label point. You can just transform a point to the other by using simple transformations. Basically, the idea is that we can approximate in the euclidean space the local structure of the manifold. This is similar to how our ventral stream

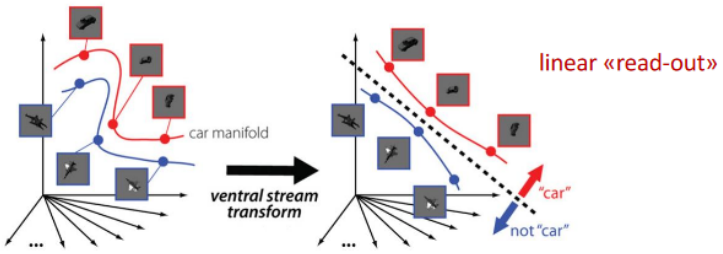


Figura 55: Manifold transformation

transform the images. If we can somehow describe this manifold, we are discovering the factor of variation of the data. When we talk about *read-out*, it's just a way to put a classifier on the data to improve the feature space where the data can be easily separated.

13.1 Complementary approaches for learning representations

When we are working on high-dimensional dataset, we have a lot of features and raw informations. Therefore, there is the need of reducing the dimensionality of this "table". There are two approaches:

1 **Clustering**: reduces the number of examples by grouping them and considering a "group prototype"

2 **Feature extraction**: reduces the number of descriptors by only considering the most informative ones or by creating more abstract features.

In both cases we reduce the dimensionality. It's also possible to combine the two approaches in some cases. To do this reduction on neural network, we start by discovering some statistical regularities (covariance/correlation) with Hebbian learning. Then, we extract principal components from the data and we find the clusters. For extracting the components, we apply the so-called

Oja Rule While the Hebb rule is a simple one, where simultaneous activation leads to pronounced increases in synaptic strength (Neurons wire together if they fire together)

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

Hoverer the **Oja rule** generalize the network by applying a *forget factor*. Here we are just subtracting every time a small fraction of the weight from the weight itself. This is basically the weight decay.

$$\tau_w \frac{dw}{dt} = vu - \alpha v^2 w$$

By including lateral connections trained using anti-Hebbian learning, hidden neurons with correlated response will develop inhibitory weights and gradually extract the k principal components of the input distribution. The lateral connections are needed because otherwise every other neuron would jsut extract the first principal component. We will not focus much on this architecture

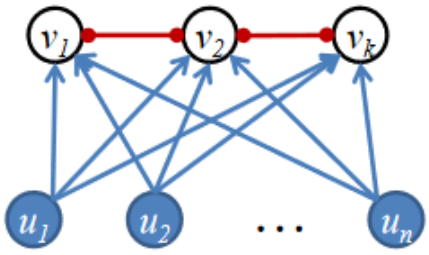


Figura 56: Lateral connection in a layer

because PCA can be computed more efficiently and it's easier to implement, and those are linear feature extraction models.

Competitive learning To implement clustering in simple neural network with a similar architecture, we can use **competitive learning mechanism**. We introduce a competitive layer, composed by a set of neurons which represent the different clusters discovered in the data (we want to decide at priori how many cluster we want to discover). Every neuron in the layer has a positive excitatory self-connection and a set of negative inhibitory connection with all the other neurons to implement competitive dynamics. Every neuron start

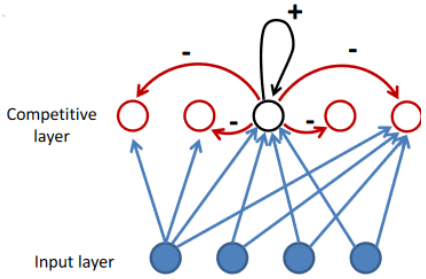
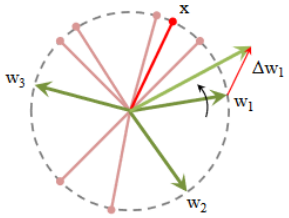


Figura 57: Example of competitive layer

with a random weight, and we activate them. This will start a "competition". The neuron which is more active tries to inhibit the other neurons and excite itself. In the end the *winner-take-all*: only one hidden neuron (the cluster prototype) will be active at the end of the competition, while all the other will be set to zero. If many neurons start with a strong activation,the competition will be harder and the dynamics to reach equilibrium state will take longer. This model is often used to implement vector quantization in signal processing. Let's see now how to implement a winner-take-all dynamics. With normalized input vectors, the activation of each hidden neuron is computed as the inner product between the neuron's weights and the input pattern

$$h_i = w_i^T x$$



The weights of the hidden neuron with the highest activation (the "winner") are then moved closer to the current input pattern, and the weight vector is re-normalized

$$\Delta w_i = \eta x$$

$$\|w_i\| = 1$$

The weights of the other neurons are left unchanged. At the end every neuron in the competitive layer will become specialized to represent a specific cluster of input data. Geometrically, this correspond to calculating the cosine distance between the data vectors and all weights vectors, and then reducing the angle between the input and the weights vector of the winner neuron. Clustering methods rely on the computation of distances to establish similarities. We can also use different measures (Euclidean distance, Manhattan distance, Mahalanobis distance) depending on the problem we are facing. A problem reside in the choice of the numbers of competitive neurons (clusters) needed. This is called *stability/plasticity dilemma:* ow can a learning system preserve its previously learned knowledge, while keeping its ability to learn new patterns? This cannot be achieved using a fixed number of cluster. Also, problems occurs when we have a non-stationary distribution, where data change over time. Therefore, learning should be **incremental**.

13.2 Topographic maps

As we have already seen, neurons in the sensory cortex are often organized in a topographic way, with nearby neurons encoding similar features. In neural networks, we have **self-impose maps**: iIf we impose a topological structure on the competitive layer, each neuron will form "coalitions" with its neighbours, which allow to more accurately map the input space into a lower-dimensional (2D) manifold. his way, each neuron will compete with distant neurons but also cooperate with its

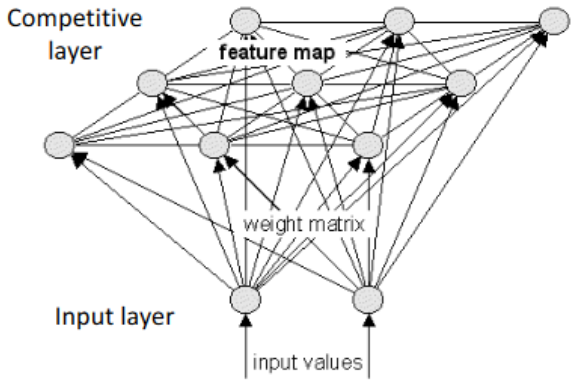
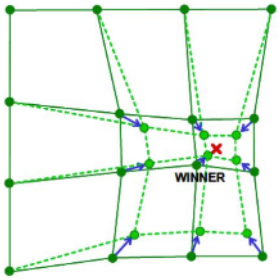


Figura 58: Topographic map

close neighbours to represent each input pattern.The idea is that the topography of the map is preserved, we are just stretching it.
Similarly to standard competitive learning, we compare the input vector to the weight vector of each hidden neuron. We establish the neuron with the closest weights the winning neuron. However, we don't just adapt the weights of the winner neuron i^*, but also the weights of its neighbours:



$$\Delta w_i = \eta \Lambda(i, i^*)(x - w_i)$$

where the neighbourhood function Λ specifies the proximity of each neuron i to the winner neuron:

- Neurons that are closer to the winner will adapt more heavily than neurons that are further; away;
- Nearby neurons receive similar updates and thus end up responding to nearby input patterns.

The magnitude of the adaptation is controlled by a learning rate, which decays over time to ensure convergence of the SOM.

The neighbourhood function The topological structure in the hidden layer can be defined in many ways. In general, each neuron will have strong excitatory connections with its immediate neighbours, which became gradually weaker as we go to longer ranges. At some point, the connections might even become inhibitory. This can be seen in the "Mexican hat" neighbour function in picture [59].

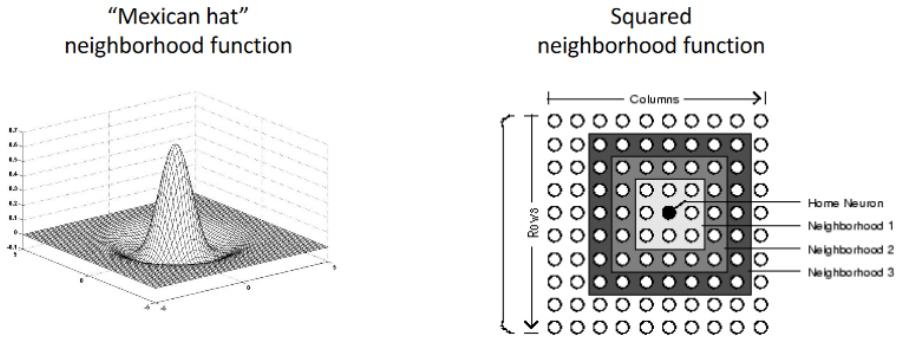
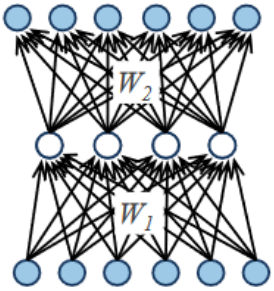


Figura 59: neighbourhood functions

Properties of SOMs Self organizing maps are mostly used to visualize high-dimensional data on two-dimensional surfaces. Those are more robust to outliers compared to standard competitive networks, and can be useful in a variety of optimization problems. One example is the "Traveling Salesman Problem", where we want to find out the shortest path connecting all points:

13.3 Auto-encoders

In an **auto-encoder** the output should just be the input, recreated/reconstructed. Deterministic models are trained using error back-propagation, like feed-forward networks. The loss function is simply the reconstruction error:



$$MSE_{train} = \frac{1}{m}\sum_{i=1}^{m} \|\hat{y^{(i)}} - y^{(i)}\|^2$$

where $\hat{y^{(i)}}$ is the training pattern and $y^{(i)}$ is the model reconstruction. Here we are just trying to measure how close is the input vector to the output vector. While in feed-forward networks there are different functions for encoding and decoding $W_1 \neq W_2$, in auto encoders we have $W_2 = W_1^T$. In this case we are regularizing a bit the model. To chose the number of hidden units, we must take into account that:

- If the number of hidden units is larger than the number of hidden units (overcomplete code) the network will just learn to copy the input;
- If the number of hidden units is smaller than the number of hidden units (undercomplete code) the network will extract relevant features from the data.

Now we are forcing the network to map the input into a lower dimensional space. If we use linear activation functions, an under complete autoencoder will learn to span the same subspace as PCA. However, if we use non-linear activation functions, we can learn more powerful mappings. We can also further improve the feature extraction process by introducing additional regularizers, which often allow to efficiently use also overcomplete codes:

- Sparse autoencoder: here the L1 penalty is applied on the hidden activations;
- Denoising autoencoders;
- Generative (stochastic) autoencoders.

In particular, in a *denoising autoencoders* the idea is that we provide a corrupted version of the input and we ask the autoencoder to reconstruct the de-noised input. By doing this the autoencoder will need to learn more general features. Choosing what kind of noise is an important part, since different types of noise might lead to different internal representations.

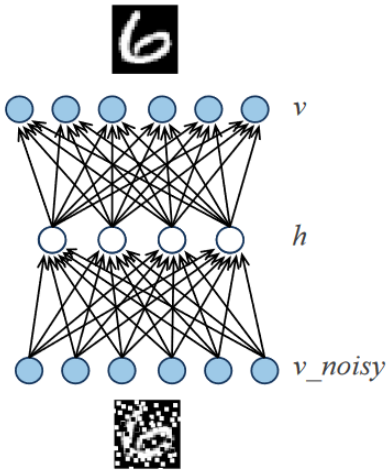


Figura 60: Denoising autoencoder

14 Associative Memories

We are moving from the deterministic framework of the feedforward network to a probabilistic framework. With associative memories, we have an activation pattern where some neurons are active and other are not active. We then want to learn that pattern (for example with Hebbian rule we try to increase the connectivity when the activation is correlating). Finally during the recall (inference/dynamics) we want to recover the most likely actiavtion from an incomplete pattern.

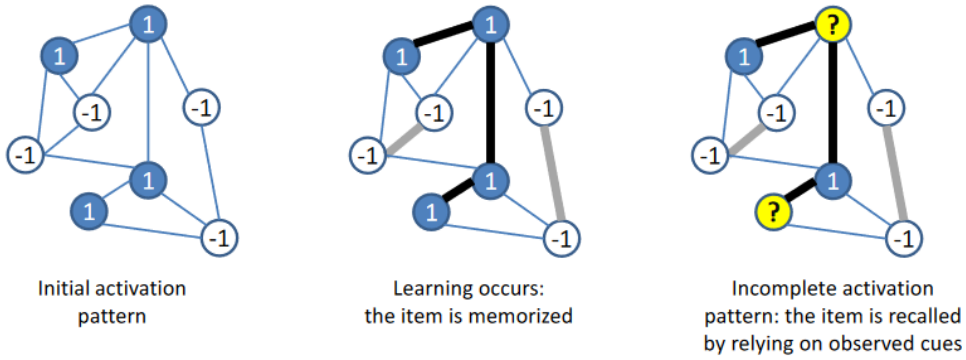


Figura 61: Associative memories using hebbian rule

For example we can imagine a face where there are 2 different representation of the pixels. By using as input a corrupted image, we want the network to reconstruct the original image. The

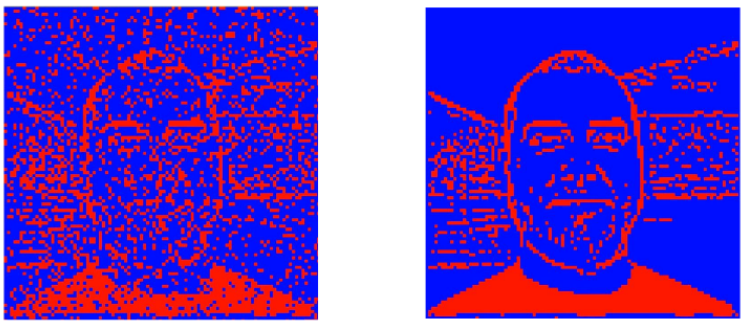
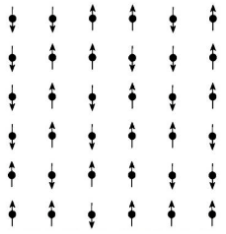


Figura 62: Associative memories using hebbian rule

key concept is how does the change of one neuron change the configuration of the energy of a system.

14.1 The Ising model



The Ising model is a model for studying phase transitions in physical systems. Here we have a bi-dimensional lattice where each component can only interact with his neighbours. Each element can be in two states, (i.e either up or down). The idea is that we can see some global properties if all the element are homogeneous. Usually the system can start from a random configuration and then it evolves. The dynamics of the system is governed by an energy function, which depends on the system temperature T, which define the most likely state of a system. The energy Hamiltonian is defined as

$$H(\sigma) = \sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j - \mu \sum_j h_j\sigma_j$$

Here we have two terms, the internal field due to *local couplings* $\sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j$ which tell us how the interaction between all the elements are impacting the energy of a certain configuration, and then we have another term which is a kind of external megnetic field we could apply, $\sum_j h_j\sigma_j$. In particular, we can define the Boltzmann distribution as

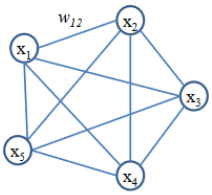
$$P_{\beta(\sigma)} = \frac{e^{-\beta H(\sigma)}}{Z_\beta}$$

Where the partition function is just summing the energy of all the possible configurations:

$$Z_\beta = \sum_\sigma e^{-\beta H(\sigma)}$$

One of the interesting property of Ising model is related to the issue of *geometric frustration*. If we try to evolve a Ising model we start from a random configuration and we try to minimizes the energy. The system will end up in a equilibrium state where the energy in not globally minimized, but rather settles into heterogeneous configurations where the energy is locally minimized. This can be translated from Ising models to Hopfield networks by considering neurons instead of atoms, extending to a fully-connected topology and make it possible to adjust the local interactions (couplings) through learning. We can define a **memory** in the system as a dynamically stable attractor, a state where the state are not really changing. When presented with a new pattern, the network will recover the most similar one by gradually settling into the closest attractor. Now the global dynamics of the network is governed by *local interactions*. Let's see how to build such associative memory. The pattern memorization is performed by gradually changing the connection weights using a simple learning rule. Pattern retrieval is performed in a dynamical way, by iteratively updating the state of the neurons until a stable state is reached. The main idea is to use an energy function to specify which states of the network are more likely to occur. The learning goal is to assign high probability to the configurations observed during training. When presented with a corrupted pattern, the network will gradually settle into the configuration corresponding to the most similar stored pattern.

14.2 Hopfield network architecture



The architecture of a Hopfield network is a fully connected, undirected graph, there are no self-connections (to ensure stability) and all the neurons are visible, that is, each one corresponds to one input variable. We define the energy function of a Hopfield network as

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

This formulation accounts for local couplings between the neurons (we are excluding the biases for simplicity). The energy landscape is usually quite complex: attractors corresponds to the local minima of the energy function, and represent stable patterns of activity that should encode the stored items. There are two mechanisms that allow moving toward energy minima:

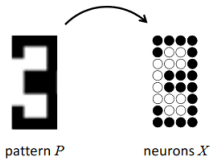
- During learning, we change the connection weights to create attractors in correspondence to training patterns (w_{ij});
- During inference, we change the neuron's activations to make them more similar to stored patterns ($x_i x_j$).

Dynamics (inference) Neurons have binary states (-1 or +1), and their activation is computed according to the hard-threshold rule:

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

In other words, each neuron x_i computes a weighted sum of the activities of all the other neurons x_j and fires if the value is above a certain threshold θ_i. The network is a dynamical systemm. Therefore, the neurons can be updated either synchronously (all together) or asynchronously (on at a time). Hopfield demonstrated that, if the weights have proper values, the network activations will always converge toward a stable state (attractor), where the energy is minimized and the mean activations will not change anymore. Therefore the network reaches the *thermal equilibrium*.

Learning To properly set the weights of the network, such that it will always recall the correct "memories", we can use Hebbian learning: that is "Neurons that fire together wire together". Assume that we have N patterns P^k to store, where k is the size of the input (e.g., pixels in the image) which corresponds to the number of neurons in the network. Then each training pattern is iteratively clamped to the network's neurons X and each connection weight is updated according to Hebbian rule: $\Delta w_{ij} = \eta x_i x_j$.



Basically, learning in the network correspond to shaping the energy function in order to create attractors corresponding to the training item. On the other hand, the energy of unlikely

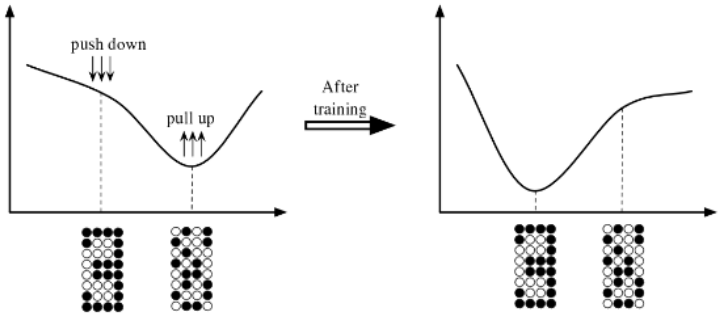
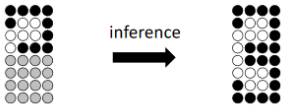


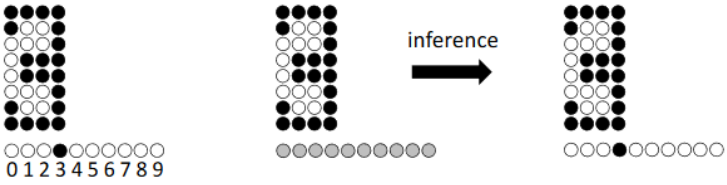
Figura 63: Carving of the energy landscape

configurations should be raised. We can see how the network is creating many local minima in the energy function corresponding to the training patterns.

Supervised learning with energy-based models Energy-based models can learn in a completely unsupervised way. They just learn how to store and retrieve data patterns However we

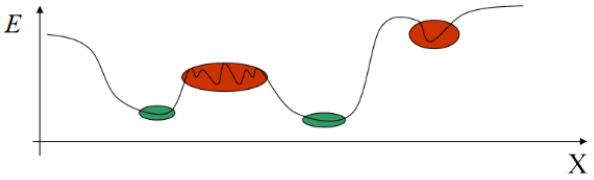


can imagine that the patterns that are encoding both the image itself and the label. Then the network can learn how to correctly retrieve it. During inference, we just show the pattern and



then ask the network to complete the missing information. Then the network will be able to recover the most likely configuration.

Storing capacity Hopfield networks can only store a *limited* number of independent patterns, which has been demonstrated to be $0.138 \cdot k$ where k is the number of neurons. Capacity is proportional to k if we are willing to accept small retrieval errors. Capacity is even lower (proportional to $\frac{k}{\log k}$) if we aim for perfect reconstruction. Moreover, as we increase the number of stored patterns (that is, the number of attractors) the network will also develop a certain number of "incorrect memories", called *spurious attractors*, which do not correspond to stable local minima but still represent equilibrium states, and can thus prevent the network to recall

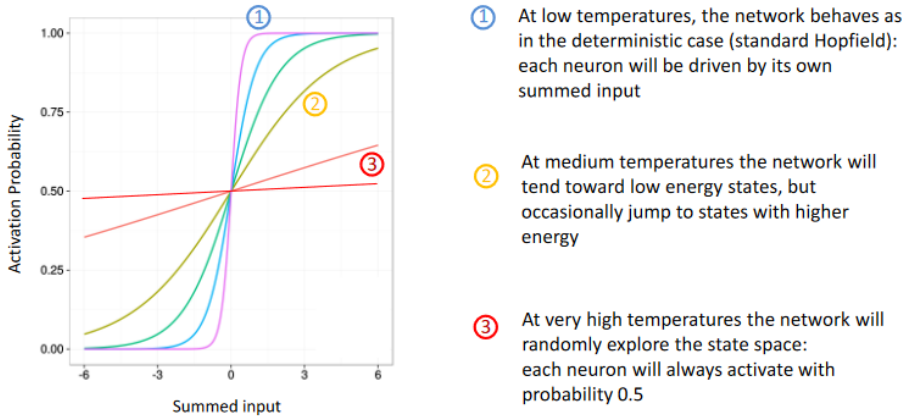


correct memories. This problem might be mitigated by introducing a *stochastic dynamics*. Here we are replacing the deterministic activation function (heavy-side) with a function that could be the sigmoid (for example). Now we are mapping the activation into a probability between 0 and 1 to being active.

$$P(x_i = 1) = \frac{1}{1 + e^{-\frac{1}{T}\sum_j w_{ij} x_j}}$$

Note that the activation is now considered as a probability value. The parameter T represents the "temperature" of the system, in analogy with physical materials.

Temperature parameter By controlling the curvature of the sigmoid, the temperature defines the degree of stochasticity of the system, that is, how much its dynamics will be *noisy*: Usually we start with high temperature, so all the neurons are activated, and then we slowly



decrease the temperature in order to reach a final deterministic behaviour. This is called **simulated annealing**. This optimization procedure has been inspired by the physical annealing of metals, which are gradually cooled in order to reach more stable atomic configurations.

15 Generative Models

In the Hopfield networks, all neurons are *visible*. In this way they can only capture direct, pairwise interactions. We can obtain something more interesting by adding a set of hidden units. In this way we can build internal representations. For example, a Boltzmann machine is composed by a set of visible units and a set of hidden units (latent units), both fully connected. Units in the hidden layer will observe the one in the visible layer and capture higher order statistical correlation of the pattern. The idea is that the environment is perceived trough the

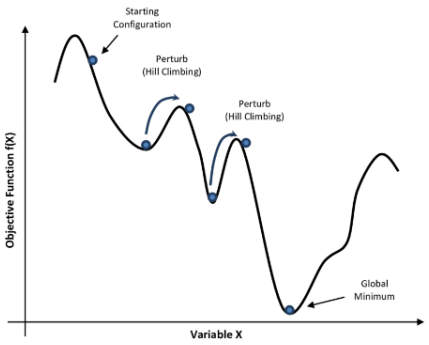


Figura 64: Search of the local minima trough temperature decrease

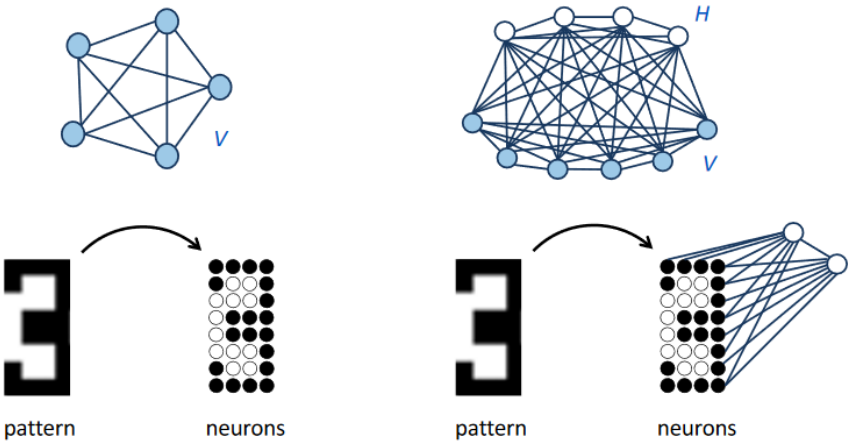


Figura 65: Hopfield network and Boltzmann machine

visible units, and then we build an internal representation of the representation by projecting it into the hidden units. Here, even if the visible units are fixed, the hidden units can adapt and change, therefore learning to encode larger correlations.

The "Bayesian brain": perception as statistical inference In **Bayesian probability** we start from the hypothesis and we ask which hypothesis could have generated some data. The Bayes theorem tells us that we can estimate the conditional probability of a certain hypothesis given some evidence (which is observed) and we try to infer the latent factor. The **Bayes rule** is the following:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)} \tag{24}$$

This tells us that we can convert the conditional probability as the product of a likelihood term multiplied by the prior of the hypothesis, and then normalized by the probability of the data itself. This is useful for *inference*. Given some observed

evidence (visible units in a Boltzmann machine), we want to establish which are the more probable hypotheses (hidden units in a Boltzmann machine) that could explain it. In particular, in Bayesian statistics the probability of every hypothesis is not fixed, it can change continuously as new evidence are collected. *Learning* in this framework means to find the parameters (connection weights in Boltzmann machine) of a generative model that best describe the data distribution (maximum-likelihood). A possible representation of a **generative neural network** is seen in picture

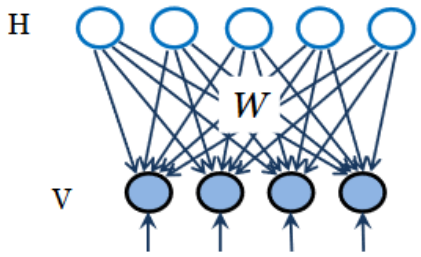
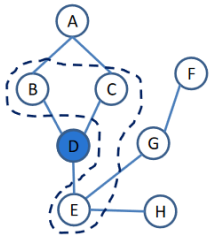


Figura 66: generative neural network

Instead of performing input-output mappings, generative models try to discover the latent structure of the input data by building a probabilistic, internal model of the environment that could have produced the data. We can imagine that latent causes of the sensory signal constitute an internal representation of the environment. In a Bayesian perspective, they represent hypotheses over the data, which are updated as new evidence is observed. However, in neural networks each hypothesis is usually encoded as a distributed activation across neurons. An example of linear generative we've saw previously is the Sparse Coding. To extend more in general this concept we use graphical representation.

15.1 Markov networks



In a **Markov Network** graph every edge is symmetric (Undirected graph), and represent the *affinity* of the correlation. Each edge has an associated factor that indicates the correlation strength. This is a general function $\phi : \Re \longleftarrow \Re^{+}$. In particular, the joint distribution of all variables can be defined as a product of local factors,normalized according to the partition function Z (this is a normalization factor that takes into account all the possible combination of the states):

$$P(A, B, C, \ldots) = \frac{1}{Z} \prod_{i=1}^{N} \theta_i(X_1, \ldots, X_k) \tag{25}$$

The **Markov blanket of a node** is defined as the set of immediate neighbours. If "observed", they make the node independent from all nodes outside the Markov blanket. By defining the Markov blanket, during inference we can exploit the local structure of the graph and consider only the variables inside the Markov blanket of the value we are interested in. Due to bidirectional interactions, we have to iteratively sample until the network converges to equilibrium distribution. This is often computationally unfeasible, so we usually form a sequence of estimators that converge toward the true distribution (Markov Chain Monte Carlo methods).

Gibbs sampling In **Gibbs sampling**, we want to estimate the value of a certain node, so we fix all the other nodes on the graphs and we use those values to constraint the inference:

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

where x_{-i} indicates the set containing all the variables except x_i. On a side-note, the transition probability does not depend on the current value of x_i but only on the value of all other units x_{-i}. This process gets repeated until equilibrium is reached. One example is shown below: If we

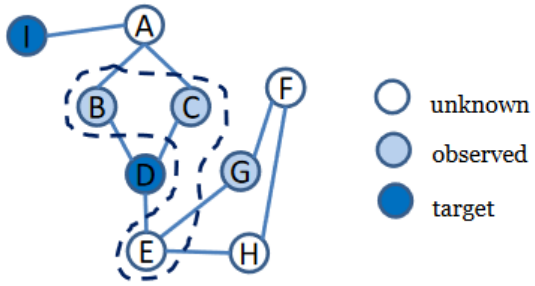


Figura 67: Example of gibbs sampling

want for example determine D, we need to consider B, C and E. While B and C are observed, E start with a random value. To obtain the probability of D, we do Gibbs sampling as the probability of the values into the Markov blanket. After sampling D, we also have to refine our estimate of E. To do this we'll sample also the unknown Markov blanket of E, and so on. In particular, conditionally independent variables can be sampled at the same time. In the example, we can do the Gibbs sampling of D and I simultaneously, since those values are not correlated.

15.2 Boltzmann Machines

Boltzmann machines are a stochastic variant of Hopfield networks that exploit hidden units to learn higher-order correlations ("features") from the data distribution. The energy function of a Boltzmann machine is written as

$$E(x) = -x^T U x - b^T x$$

we can then define the probability of a state using the energy function as follows:

$$P(x) = \frac{exp(-E(x))}{Z}$$

where $exp(-E(x))$ is still a product of local factors, but expressed as a log-linear model:

$$exp(-E(x)) = \ldots = \frac{1}{Z} \prod_{i=1}^{N} \theta_i(X_1, \ldots, X_k)$$

We can also decompose the energy function by explicitly represent the distinction between visible and hidden units:

$$E(v, h) = -v^T R v - v^T W h - h^T S h - b^T v - c^T h$$

For doing **inference** in a Boltzmann machine, we can sample the state of every unit using the activation function: neurons have binary states (0 or 1), and their activation is sampled according to the stochastic activation function.

$$P(x_i = 1) = \frac{1}{1 + e^{-\frac{1}{T}\Sigma_j w_{ij} x_j}}$$

Differently from the Hopfield model, here we have two separate phases, one for visible units and one for hidden units:

1 **Data driven inference**: Here we start with an input pattern clamped (observed) on the visible units, while the hidden units have a random initial activation. Therefore only the hidden units must be estimated. Then we iteratively update the activation value of each hidden neuron, one at a time, until the network reaches equilibrium (i.e using Gibbs sampling);

2 **Model driven sampling**: Here no variables are observed, so all units start with a random activation. Then we iteratively update the activation value of each neuron, one at a time, until the network reaches equilibrium (i.e Gibbs Sampling).

We can use simulated annealing to improve convergence towards good local minima, thereby avoiding spurious attractors. This however stay a very demanding procedure not used in practice.

Learning in Boltzmann Machine Learning in a Boltzmann machine is based on *maximum-likelihood*: we want to assign high probability (low energy) to the observed data points, and low probability to the remaining points. Maximizing the product of the probabilities that the Boltzmann machine assigns to the training patterns is equivalent to maximizing the sum of the log probabilities. To do this we consider the log-likelihood gradient:

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

In particular,

- $\langle v_i h_j \rangle_{data}$ is the expected value of the product of states (correlations) in the data distribution (we can imagine it as the "real" sensory perception);
- $\langle v_i h_j \rangle_{model}$ is the expected value when the network is sampling state vectors from its equilibrium distribution and no units have been clamped to the data (we can imagine it as the "fantasy" perception).

We should end up in a situation where the model driven component is fairly similar to the data driven one. After calculating those two components, we can change the weights in the direction that maximizes this likelihood:

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

This can also be interpreted as minimizing the Kullback-Leibler divergence between the data distribution and the equilibrium distribution over the visible variables produced by the generative model.

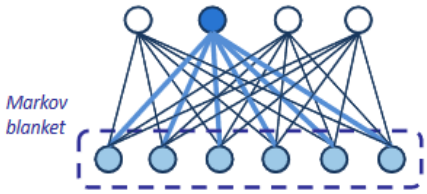


Figura 68: Example of restricted Boltzmann machine

Restricted Boltzmann machines We can restrict a Boltzmann machine by removing the intra-layer connections. Now what we have is a fully-connected *bipartite* graph. The topology of the graph now encodes conditional independences: we can infer the state of units in the same layer in one single, parallel step (**block Gibbs sampling**). Again we can find the joint probability of hidden and visible units as

$$p(v,h) = \frac{1}{Z}\prod_{i=1}^{N}\theta_i(v_k h_k) = \frac{e^{-E(v,h)}}{Z}$$

where the energy function is

$$E(v,h) = -b^T v - c^T h - h^T W v$$

which here only consider the visible interactions. To estimate the probability of the entire hidden layer given the observed values we can simply taking the product of individual joint probabilities:

$$P(h|v) = \prod_i P(h_i|v)$$

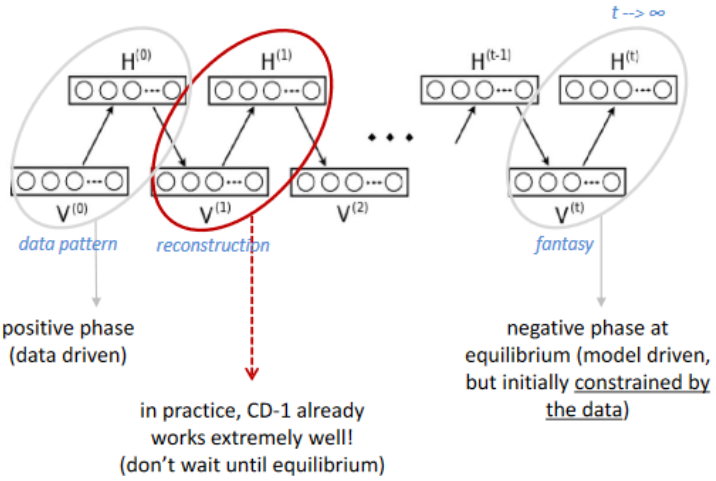
therefore we can to the sampling simultaneously for every unit. If we observe the values of the hidden units and we want to generate the sample model, we can simply write:

$$P(v|h) = \prod_i P(v_i|h)$$

Those equations show that it's possible to speed up inference dramatically by removing the intra-layer connections. In this topology the maximum-likelihood learning is performed by contrastive divergence:

$$\Delta w = \eta(\langle v_i h_j\rangle_{data} - \langle v_i h_j\rangle_{model^*}) = \eta(KL(P_{data}\|P_{model}) - KL(P_{reconstruction}\|P_{model}))$$

Contrastive divergence The original idea was to minimize the discrepancy between the empirical data distribution (training set) and the model distribution (patterns generated by the network). To lower the computational demands, we can approximate the model driven phase using some reconstructions (distortions) on the data produced by the model. To approximate the model driven phase we start by some data pattern, then we do the inference over the hidden units in one step. Instead of computing the model driven sampling, we start from a biased configuration that is driven by the data pattern we are analizing. Let's see how to implement the contrastive divergence algorithm. For each training pattern , we can divide the algorithm in 3 main phases:



1 **Positive phase**:

- The pattern is presented to the network (clamped on visible neurons v);
- The activations of hidden neurons h are computed in a single step using the stochastic activation function;
- We compute the correlations $\langle v_i, h_j \rangle$ between all visible and hidden neurons.

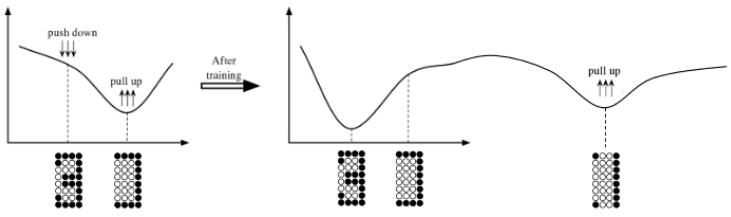
2 **Negative phase**:

- Starting from the hidden neurons activations computed during the positive phase, we generate activations on the visible layer using the stochastic activation function;
- Starting from these new visible activations ("reconstructed data"), we compute again the activations of the hidden neurons;
- We compute the correlations $\langle v_i, h_j \rangle$ between all visible and hidden neurons.

3 **Weights update**: The weights are then updated according to the rule:

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

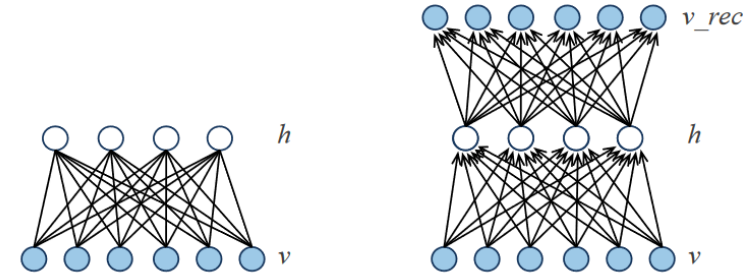
We call this algorithm CD-1 because we do only one step of reconstruction. In particular, CD-1 is lowering the energy of states corresponding to training patterns, and increasing the energy of their neighbour states(distortions produced by the model)



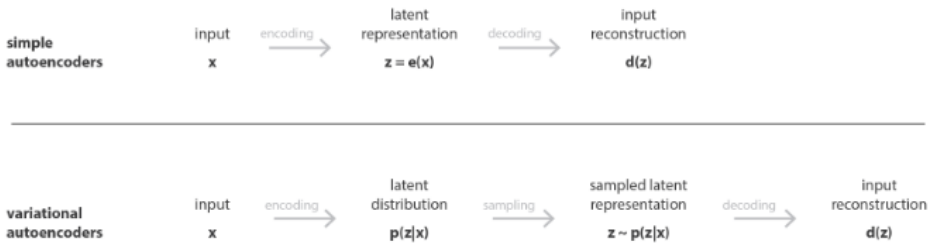
We can increase the energy also for regions of the data-space that the model likes but which are very far from any training data by starting with small weight and using CD-1. Once the weights grow, the Markov chain mixes more slowly so we use CD-3. Once the weights have grown more we use CD-10, and so on. It can be proved that Restricted Boltzmann Machines are universal approximators of probability mass functions over discrete variables. Temporal (sequential) extensions of the RBM have shown to be universal approximators of stochastic processes with finite-time dependence. Those theoretical results shows the power of those procedure in principle.

15.3 Variational Autoencoders

The idea in (single layer) Boltzmann machines is to have an indirected model. We have seen that in the fully-connected case a Boltzmann machine is very demanding. In the restricted case by removing the intra-layer connections we have obtain performances. With those architecture, we can also generate data. We already saw how auntencoders works. This apparently shows some



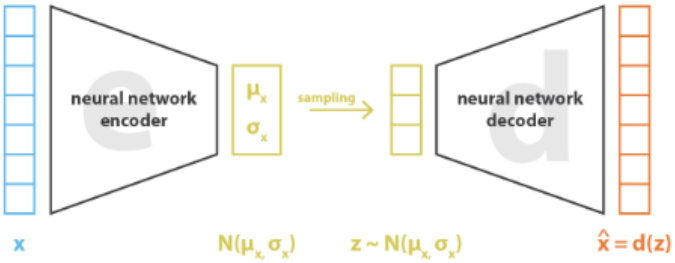
similarities with Boltzmann machines. However the main difference the issue in autoencoders is that we always have to start from a certain input, while Boltzmann machines can also start from a random configuration. An generative autoencoder is called **variational autoencoder**. With standard autoencoder there is a irregular latent space (close points in latent space can produce very different patterns over visible units) so we cannot implement a generative process that simply samples a vector from the latent space and passes it through the decoder. A possible solution to autoencoder is to make them *probabilistic*. The main idea is to have the encoder to produce a distribution over the latent space instead of just projecting the data into a single hidden unit activation. We want this distribution to have some good properties so we include a regularization term in the loss, which tries to constrain this distribution in the latent space. The loss function is now derived using variational inference techniques. The encoded distribution are



chosen to be Gaussian, so that the encoder can be trained to return the mean and covariance matrix. From this Gaussian we sample the representation from which we started to decode. This way we can more easily regularize the loss function, by forcing the latent distribution to be as close as possible to a standard normal distribution. The loss is defined as

$$loss = \|x - \hat{x}\|^2 + KL[N(\mu_x, \sigma_x), N(0,1)] = \|x - d(z)\|^2 + KL[N(\mu_x, \sigma_x), N(0,1)]$$

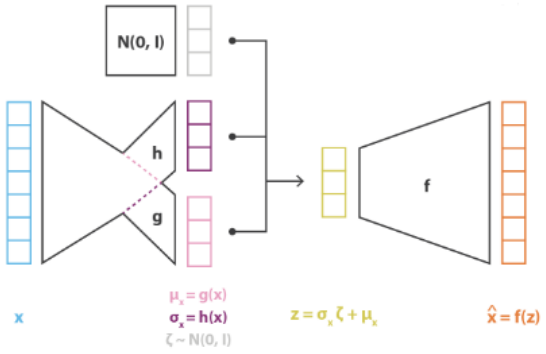
where KL is the Kullback-Leibler divergence. Note that the latent representation is now defi-



ned by two vectors (mean and covariance), so the encoder network has two (possibly partially overlapping) branches. The covariance should actually be a square matrix; however, to reduce computational complexity we assume that the multidimensional Gaussian has a diagonal covariance matrix (i.e., latent variables are independent, as in RBMs). Moreover, the sampling process is discrete, thus preventing the use of backpropagation. Thus we need to re-parametrize z as

$$z = h(x)\zeta + g(x) \quad \zeta \sim N(0,1)$$

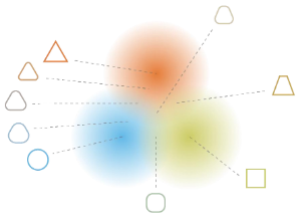
The advantage of this is that we are regularizing the latent space. Therefore now if we take



two latent vector nearby they will probably generate similar concept. The regularization term in promoting the creation of a gradient over the latent representations, which allows to generate samples varying smoothly.

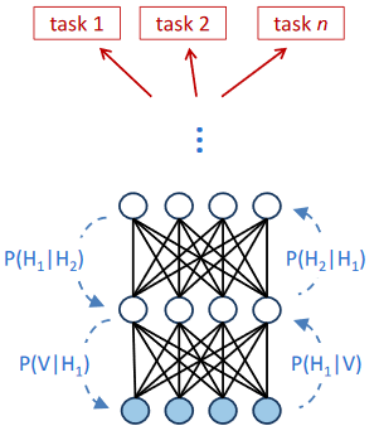
16 Unsupervised Deep Learning

To combine many simple (single-layer) blocks into a multi-layer model, there are different "routes":



- **Hierarchical undirected models**: We take a single-layer restricted Boltzmann machine and we stack it in a deep belief network or a deep Boltzmann machine. The issue is that learning is greedy (each hidden layer is optimized independently) and layer-wise (hidden layers are trained sequentially,starting from the bottom layer);
- **Hierarchical feed-forward models**: Here we use feed-forward network with back-propagation, and we can possibly have variational auto-encoders or the generative adversarial networks. Here learning is backpropagated end-to-end.

16.1 Deep belief networks



To build a **deep belief network**, we start with a single layer Boltzmann machine. This can be interpreted as a graphical model where we take the conditional probability of the hypothesis given the visible unit and vice versa when we do the top-down. This get trained like a restricted Boltzmann-machine using contrastive divergence and then we take the internal representation as input to a subsequent restricted Boltzmann machine. Now we have multiple levels of representations encoded as a hierarchical generative model (hypotheses over hypotheses). In particular, it's still possible to do some supervised learning tasks (i.e by decoding with a linear classifier the top layer representation). We can also fine-tune the entire architecture according to a supervised loss function using back-propagation. In particular we can also do generation with those kind of networks. By training the network in a unsupervised way and then adding the labels, we can propagate the evidence and reconstruct the class.

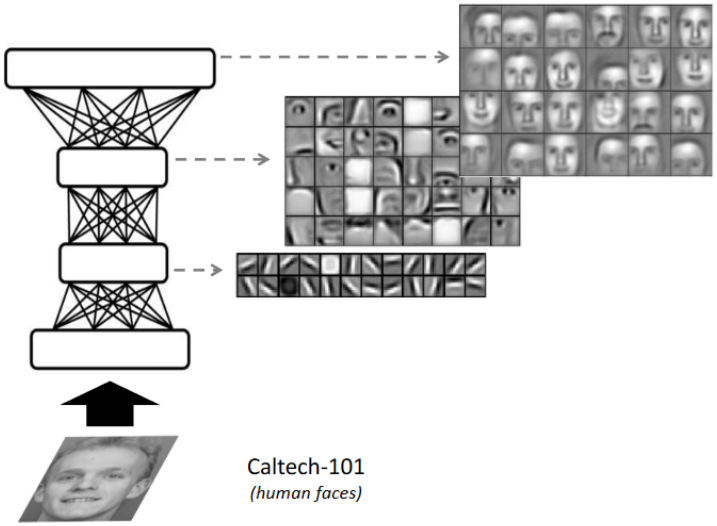


Figura 69: deep belief network trained using faces

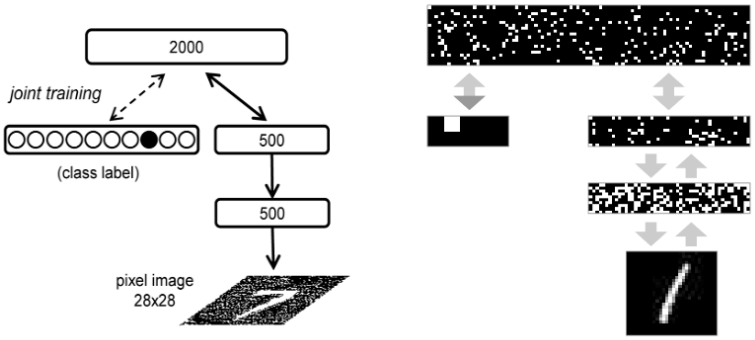


Figura 70: deep belief network generating numbers

16.2 Generative adversarial networks

In unsupervised learning we use Mean Squared Error to compute the loss. Missing few input values might not result in a big loss, since the majority of the input values have been reconstructed properly. However, sometimes small changes might be important, and losing those information might result in bad reconstructions. To solve this, we can use **Generative adversarial networks**. Here we have two main pieces, the *generator* and the *discriminator*. We have a training dataset. The generator just generate data while the discriminator act as a classifier which tries to discriminate between the data in the training set and the fake data generated by the generator. At every iteration, we decide to take either one sample from the dataset or the generated set, then the discriminator will determine if the image is real/fake. Both the generator and the discriminator can be (should be) deep neural networks. For example, if we consider the MNIST dataset, we want to learn to produce digits. We have the mnist dataset as training data, and the

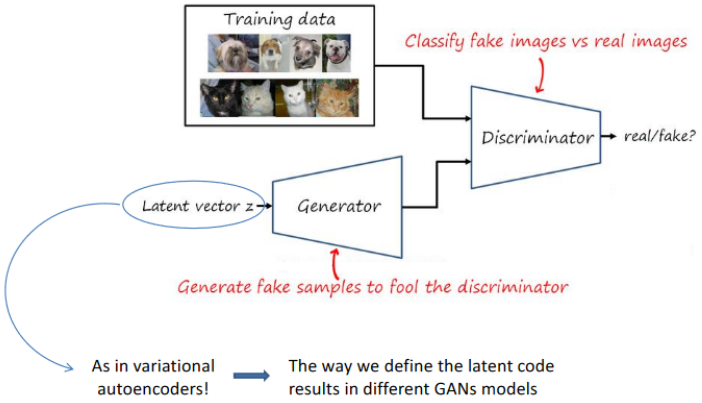


Figura 71: Example of a generative adversarial network scheme

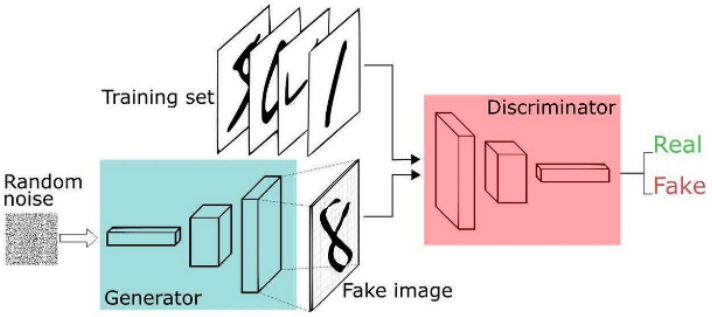


Figura 72: generative adversarial network with mnist digit

digits produced by the generator. The discriminator must learn how to discriminate between the two. At the beginning, the generator will be scramble. We'll start with random noise, and with the weights not tuned we'll just produce random noise. Thus at the beginning the discrimination will be pretty accurate. We train the whole system using back propagation:

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

As we slowly back-propagate, the generator will start to produce better fakes. In particular, the learning dynamics is a game theory perspective: learning is a minimax two-players game, with a value function that one agent seeks to maximize and the other seeks to minimize

$$min_G max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

where $D(x)$ represents the probability that x comes from the true data distribution. The Discriminator parameters are updated by ascending its stochastic gradient:

$$\Delta_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[logD(x^{(i)} + log(1 - D(G(z^{(i)}))) \right]$$

The Generator parameters are instead updated by descending its stochastic gradient:

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

The game terminates at a saddle point that is a minimum with respect to one player's strategy and a maximum with respect to the other player's strategy. The equilibrium state is hopefully achieved when the Generator produces data indistinguishable from the training distribution, and the Discriminator always predicts "true" or "fake" with probability 0.5 (i.e., the Generator wins). This can bee seen in picture [73].

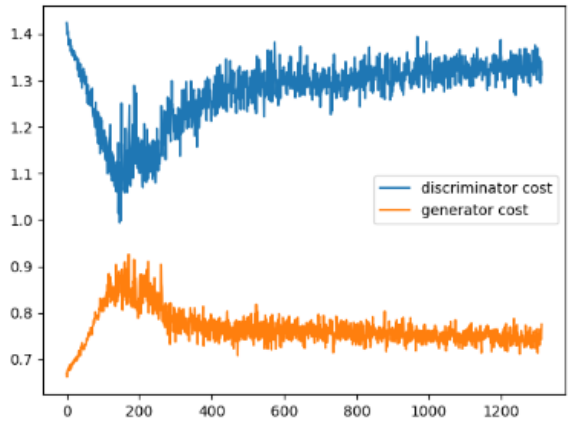


Figura 73: losses of generator and discriminator

Advanced Generative Adversarial Network architectures The way we define the initial latent code and the discriminator loss results in different GAN models. Some of the most common are

- **InfoGAN**(Information Maximizing GAN): The idea is that rather than starting from a totally random vector, we try to add specific semantic meaning to the variables in the latent space;
- **WGAN** (Wasserstein GAN): Here the discriminator is replaced by a "critic" that, rather than classifying, scores the realness or fakeness of a given image. This results in training much more stable and fast.

17 Reinforcement learning

In **reinforcement learning**. instead of learning from approximating a function/structure of the labels, in reinforcement learning the model make an action in the environment and see the consequences. Reinforcement learning is inspired by Skinner's behaviourist psychology, applying observations on animals to software agents. We can now define the concept of **operant conditioning**: specific consequence are associated with voluntary behaviour. In particular,

- *rewards* are introduced to increase a behaviour;

- *punishment* are introduced to decrease a behaviour.

This can be formalized by the Markov Decision Process.

17.1 The Markov Decision Process

In the **Markov decision process**, we have:

1 An **agent**, which can perform actions on the environment;

2 The **environment**, which has a state which respects the Markov property.

The action of the agent depends on the current state of the environment, and influence the next state. There's also a reward for each action at a certain time-step that the agent can get and see. The agent is trying to maximise the reward by taking the correct action. In particular,

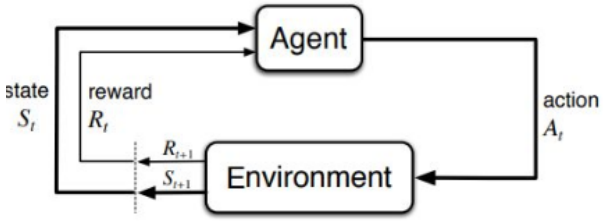


Figura 74: Markov decision process

maximizing the instantaneous reward is not always the correct action. To learn more complex behaviour, the agent needs to learn over time. Thus, the agent tries to maximize the **long-term expected reward**, which is defined as

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

where γ is a parameter between $0 \leq \gamma < 1$ used to represents the *foresightedness* of the agent. The presence of delayed rewards makes however the task more complex: it is hard to link the reward with the past actions that led to it.

value function We can define the *value function* of a state as the expected reward in the state as

$$v(s) = E[G_t | S_t = s]$$

We can now unpack the long-term reward to get the **Bellman equation**:

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

Both these equations require full knowledge of the **policy** Π of the agent. The policy is a function that maps actions into states:

$$\pi(a|s) = P[A_t = a | S_t = s]$$

And to find the optimal policy is the objective of the learner. The Bellman equation then becomes:

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

So the optimal solution to a Markov decision problem is dynamic programming: you take the bellman equation directly and you solve it for the policy. There are however two main issues:

- The complexity is exponential, thus finding the optimal policy might be computationally infeasible;
- The values of $p(s\prime|s,a)$ and $R(s,a)$ needs to be perfectly known in advance, and this is often impossible.

What we do is **temporal difference learning**: starting from a default estimate of the value function for each state-action pair, which we call **Q-value**, and improve it over time by making decisions and seeing what happens. This approach is model free (we just need the definition of the state and the knowledge of which state we're in). In particular, we use the Q-value of the next state as an estimate of future value (**bootstrap**). Therefore we define the new estimate as

$$Q(s_t,a_t) \longleftarrow Q(s_t,a_t) + \alpha\left(r_t + \gamma + max_a Q(s_{t+1},a) - Q(s_t,a_t)\right)$$

where $Q(s_t,a_t)$ is the old value, α is the learning rate, r_t is the reward, γ is the discount factor and $max_a Q(s_{t+1},a)$ is the estimate of the optimal future value. In particular, the sum $(r_t + \gamma + max_a Q(s_{t+1},a))$ express the learned value. This formula is guaranteed to converge if α respects some basic properties (the sum diverges, but the sum of squares converges).By using the maximum a in the estimate of the optimal feature value, we are using a *greedy* update policy, which is the one that tries to maximize the reward at each step. The agent also has a **behaviour policy**, which it uses to select the next action in each state. Thus, we can say that, in reinforcement learning, a learner is

- **On-policy** if the update policy and the behaviour policy coincide;
- **Off-policy** if the update policy and the behaviour policy are different;

There are several behaviour polices that are used in practice. In particular, the most common ones are the ε-*greedy* policy and the *softmax* policy. In both cases, there is some randomness to explore the state and action spaces. ε-greedy is a policy which select the greedy action (the

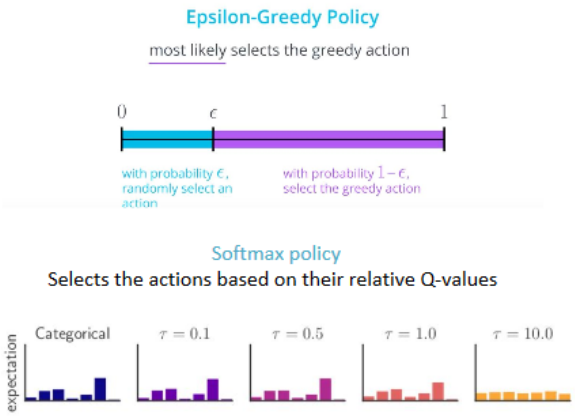


Figura 75: softmax vs ε-greedy policies

action with the highest reward) with probability $1-\varepsilon$, while with probability ε the learner select

one of the non-greedy action. On the other hand, the softmax policy select the action based on the softmax distribution of the Q-values. So there is a vector of Q-values for each action, and the learner select the action based on the temperature. High temperature will equal to a uniform distribution, while zero temperature is the greedy action. We'll need to change the temperature over time to ensure a good learning. The advantage of softmax over ε-greedy is that the learner makes smarter decision when it explore. The best algorithm for reinforcement learning at the moment are:

- **Q-learning** : off-policy algorithm which uses a stochastic behaviour policy to improve exploration and a greedy update policy;
- **State-Action-Reward-State-Action** (SARSA): on-policy algorithm which uses the stochastic behaviour policy to update its estimates.

The off-policy algorithms have an advantage, since they can take more risks, as they assume they won't make mistakes in the next step. Usually, during training stochastic policies are used.

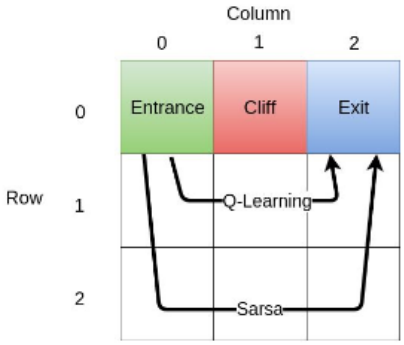
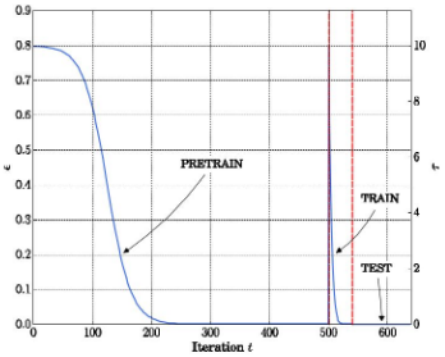


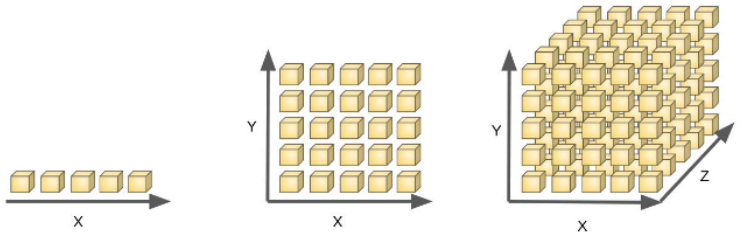
Figura 76: Q-learning vs sarsa

Then when the training is done, there's a switch to greedy policy since it's the optimal one. **Exploration** is crucial in the initial phases, since the agent needs to find out as much as it can from the environment. If the agent is too greedy in the first episodes, it can get stuck in a local maximum. The learner can't learn from a trial-error if it doesn't make errors. In general, in practical, a pre-training phase using a simplified environment can help if the real one is unavailable or computationally heavy. Foresightedness has pros but even drawbacks. In particular,



the discount factor (γ) can affect the convergence of the algorithm. A high value can improve foresightedness and make the network find a better solution, but at the same time convergence is slower, as the bootstrap makes the updates rely more on current estimates. Another problem that comes up is the so-called curse of dimensionality.

curse of dimensionality If the task is complex, then it might have a huge state space. (i.e if the state is a 20x20 8 bit grayscale image, the agent has 21200 possible states) In particular, each new dimension multiplies the size of the state space. If a problem is multi-dimensional, the quantization of each parameter needs to be very rough. The curse of dimensionality means



that linear estimators are not enough. Therefore, we need a way to generalize experience faster. Therefore, we can use a neural network. The problem of neural networks as Q-value estimators is **instability**. The issue can be solved by keeping two networks: a prediction network that is trained at each step, and a target network (used for action selection) which is periodically updated to the update network's values. The first Deep Q-learning algorithm, DQN, was developed by

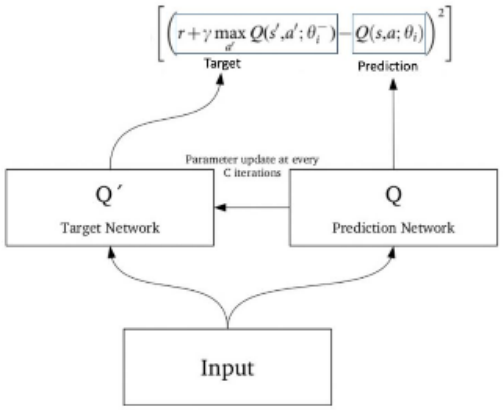


Figura 77: NN for Q-learning

DeepMind in 2015. DQN managed to outperform humans in several classic Atari games, using only the screen state as an input. An improvement on DQN (AlphaGo) beat the human world champions of chess and go.

Experience replay Neural networks need independent samples to train correctly, but actions are heavily influenced by the history of the learner. **Experience replay** solves the problem by saving samples in a replay memory and training DQN on mini-batches. In particular, the replay memory can be adjusted to show more important transitions more often to speed up training.

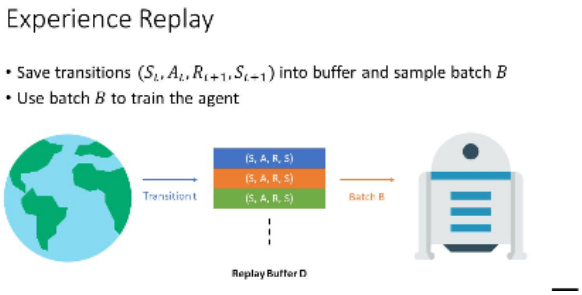
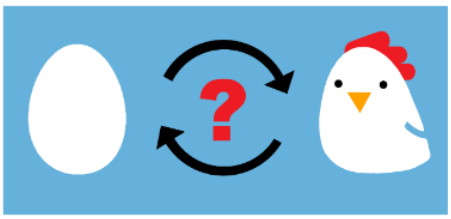


Figura 78: experience replay

18 Causality

We all know that correlation does not imply causation. There are good reasons to believe that we need to actively manipulate the environment if we want to move from simply discovering correlations to discovering causal structure. If we can actively interact with the environment, then it's useful for discover probabilistic causal models. Since reinforcement learning is the only way Artificial agent can interact with the environment: some suppose robots could learn causal relationship by acting with the environment. To solve causality issue, we can think that Y is a



cause of Z if we can change Z by manipulating Y.

19 Advanced topics in reinforcement learning

One of the biggest issue with Q-learning or TD-learning is learning speed: those algorithm takes a lot of time and experience in order to converge to some useful behaviour. We shouldn't just look at the final performance, but also at how the performance develop over time (*sample efficient RL*). Moreover, Deep RL algorithm can learn sophisticated control strategies, but might fail in simple scenarios that require high-level knowledge. To overcome this issue, we can use **semi-supervised learning**: instead of waiting for sparse rewards, thus getting rewarded only at the end, you try to progressively guide the agent trough the environment. There could be

- **shaping policies:** this is inspired by animal training. You start with smaller reward for every action that leads toward the goal, and give a biggest reward at the end;
- **transfer and curriculum learning:** you use subtasks to better explore the environment;
- **imitation learning:** you show the behaviour so that the agent may imitate it by itself later.

19.1 Model-based RL

To further improve this, we can implement a model-based approach. The idea is that it's possible to learn an internal model of the environment, which can be used for planning (by predicting the new state s_{t+1} given the current state s_t and the action a_t). This allows to reduce the number of

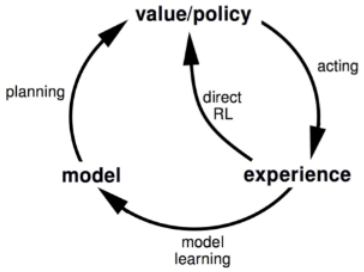


Figura 79: Model-based RL

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. This allows to reduce the number of interaction with the environment, but you need a good model of it.

19.2 Curiosity-driven RL

Usually psychologist separate the concept of *intrinsic* and *extrinsic* motivation:

- **Extrinsic motivation:** when your behaviour is motivated by an external factor pushing you to do something in hopes of earning a reward or avoiding punishment;
- **Intrinsic motivation:** when your behaviour is motivated by your own internal desire to do something for its own sake.

The main idea of curiosity-driven reinforcement learning is that rather than defining an external reward function, we build a reward function that is intrinsic to the agent(generated by the agent itself). Here is important to gain knowledge about environment dynamics before trying to solve the problem. An *intrinsic reward* can be defined as the error between the predicted new state s_{t+1} given our state stand action a_t and the real new state. Here, reward will be small in familiar (or easy-to-predict) states. This will push the agent to seek states where the agent has spent less time(or with more complex dynamics) and consequently better explore the environment. Similarly to model-based RL, the agent learns an internal model of environmental dynamics. However, here there is no explicit task to be performed.

19.3 Multi-agent RL (MARL)

In most real-world scenarios, the environment includes *multiple agents*. This greatly increases learning complexity, because environment is no more stationary (other agents can change it) therefore Markov assumption is violated. Furthermore, the environment is no more fully observable, because we need to consider about how the other agents are thinking. The other agents can be either *cooperative* (if they try to collaborate) or *competitive* (if reward functions are in contrast) or *irrelevant* (if they are simply a source of stochasticity). Learning can be *centralized*

if all agents share the same policy, or *decentralized*, if each agent learns by itself. The agents should try to coordinate trough learning to communicate and/or learning to understand other's intentions.