Artificial Neural Networks and Deep Learning

1. Machine Learning vs Deep Learning

- **Machine Learning paradigms**: supervised learning, unsupervised learning and reinforcement learning
  Machine Learning is a category of research focused on finding patterns in data and using those patterns to make predictions. "A machine is said to learn from experience E w.r.t. a task T and a performance measure P if its performance at task T, measured by P, improves because of experience E." Suppose we have a certain experience E, i.e. data: $D = x_1, x_2, .., x_N$. We can have 3 main classes of problems:

  - ***Supervised learning***. *Given a set of data D and their desired outputs $t_1, t_2, .., t_N$, we want to learn a model which will be able to correctly assign labels to new data.*
  - ***Unsupervised learning***. *Given a set of data D (only data, without labels) we want to find regularities and similarities in order to classify the data into groups.*
  - ***Reinforcement learning***. *Given a set of data D, a set of actions $a_1, a_2, .., a_n$ which affects the enviroment and a set of rewards $r_1, r_2, .., r_N$, we want a model to learn how to act in order to maximize the rewards in the long term.*

Supervised learning. *The field of supervised learning divides in 2 tasks: classification, in which the output are labels, and regression, in which the output are numbers. In both cases we have a source of informations of the data (e.g. an image, a sound, a text) and we extract some features, which we put in a vector $\underline{x}$. Let's focus on classification (regression is its continuous counterpart and so it follows the same principles). A classifier is just a criterion to separate one class from the other, and so, learning a classifier is just learning a separating boundary. Formalized: given a set of labeled data, the goal is to find a separating hyperplane in the space of the extracted features such that different groups are separated. In this way, the space of features is partitioned and it'll be easy to determine the label of a new element based on its features. The main problem is that the existence of this separating hyperplane depends on which features we extract from raw data, which means that if we extract the wrong features it'll be impossible to achieve a good classifier. A way to prevent this is through the semi-supervised learning, where we obtain a set of relevant features thanks to an initial massive extraction of all the possible features and a successive unsupervised classification process. However this is not the best way to procede. An example of supervised learning is: given a set of images of cars and motorcycles we want to learn a model that will be able to classify an image as car or motorcycle. Otherwise, given the same set of data and an addictional information, the price, we want to learn a model that will predict the price given an image.*

Unsupervised learning. *The field of unsupervised learning focuses on clustering. Given a set of data D we want to learn some clusters, i.e. we want to be able to say that a group of elements is more similar among each other than with other elements. To procede we have to extract some features, put them in a vector $\underline{x}$ and then work with distances among points in the space of the extracted features. Again, we have some problems due to the hand-crafted feature extraction: if we extract features which are not relevant, we'll obtain unefficient clusters. An example of unsupervised learning is: given a set of images of cars and motorcycles (we know that they're cars and motorcycles, but there are no official lables) we want to learn a model which is able to cluster the data in two groups and which is able to determine which one of these groups is closer to a new image.*

- **Deep Learning**: difference with Machine Learning
  The main problem in Machine Learning is the hand-crafted feature extraction process. The performance of any task is strongly conditioned ~~to~~ on how good (relevant for the task) the extracted features are. Even the semi-supervised learning improvement deals with hand-crafted features extraction, which is too subjective. Hence we introduce Deep Learning. The focus of Deep Learning is to learn about the representation of the data (if we extract the right features it'll be straightforward to partition the features space). Deep Learning takes care of the whole features extraction - selection process, there is no feature engineering part. The aim is to find the best space to put the points (representing data) so that they can be easily separated. Once we have an optimal space, obtaining a good classifier is easy: we can procede with some Machine Learning methods to find the separating hyperplane. How can Deep Learning learn itself the features (the optimal space for the data representation)? Through a hierarchical representation. However, a drawback is that we need a huge amount of data. Another drawback is that, this way of extracting features may lead to features which are completely un-interpretable.
  Note. *"Deep" means that we push learning down in the hierarchy between raw data and clasification through all the layers of abstraction (implemented by feature extractor). We implement this with Neural Networks.*

2. Feed-forward Neural Networks

- The **perceptron** and the **Hebbian learning**: the idea and the math behind it, the limitations (convergence, non-existence of a separating hyperplane, non-linear boundaries)
  *The perceptron was introduced to simulate the human neuron behaviour. A human neuron receives informations as charges and it accumulates them. Once the total charge in a neuron exceeds a certain treshold, the entire charge is released and divided (through weights) into the other connected neurons. We can copy this behaviour for an artificial neuron: a neuron receives weighted (ω_i) input signals from other neurons (x_i), it sums them and it compares the sum with a treshold (bias = b), having as output +1 if the total charge exceed the trashold or -1 if not. This is the perceptron. What can we do with it? With the right set of weights the perceptron can implement a logical operator AND or OR, and so, we can obtain a logic network. How can we decide the right weights? By training. The power of the perceptron is in the fact that we learn the weights, we not set them. The **Hebbian learning** is a learning method that specifies how much the connection between two neurons (a weight ω_i) should be increased/decreased in proportion to the product of their activation:* [redacted]
  In practice we set a random initialization of the weights, we go through the dataset multiple times and we modify the weights each time a sample is wrongly predicted. We use the "online" procedure: we update the weights at each sample and we use them for the next sample. An alternative to that is the batch procedure: we go through a group of data and we update the weights only at the end. A pass through all the data is called epoch.

  The math. *The perceptron computes a weighted sum and return its sign: $h(x|\omega) = sign(\omega_0 + \omega_1 x_1 + .. + \omega_I x_I)$, where $\omega_0 = -b$. This is equivalent to a linear classifier for which the decision boundary is an hyperplane $(\omega_0 + \omega_1 x_1 + .. + \omega_I x_I = 0)$ and so we can say that the perceptron is looking for the separating hyperplane that puts all the points classified as "+1" in one region and all the "-1" in the other. This procedure may not converge always to the same set of weights. In fact, the same hyperplane can be described in infinite ways. Moreover, this procedure may not converge at all, it depends on the existence of the separating hyperplane. A strong drawback is that we can't use perceptrons to learn a separating boundary which is **not** linear. To solve the problem of having complex boundaries we can introduce a multi layer perceptrons. Adding more layers leads to the ability of model complex boundaries. However, the Hebbian learning fails with multi layers. This is because we don't have the informations to update **all** the weights. The optimal way of proceding the backpropagation (being able to propagate the error so that we can touch every weight). However, it turns out that the real problem with the perceptron-Hebbian learning way is the perceptron itself: the activation function is not differentiable.*

  Note. *It can be shown that the error function the Hebbian rule is minimizing is the distance of misclassified points from the decision boundary. This error function is **not** based on assumptions of the probability distribution of the data (and so it's not about MLE). It's build on geometric reasoning.*

- **Feed-forward Neural Networks**: improvement w.r.t. the perceptron, backpropagation, activation functions (math, characteristics, usage, good/bad and derivatives of: sigmoid, hyperbolic tangent) and output layers
  The way to have backpropagation working is by changing the structure of the neural network: we don't want multi-layer perceptrons since we don't want perceptrons (the associated activation function is the step function, which is non differentiable). Instead, we talk about feed forward neural networks composed by artificial neurons ($\neq$ perceptrons because of the different (smoother) activation function). In this model we have I input neurons, K output neurons and hidden layers. How many hidden layers? How many neurons per layers? We don't know it a priori. The rules are: the activation functions must be differentiable, the signal must go forward and the output of a neuron depends only of the neurons of the previous layer.

  Activation functions. *The choice of the activation function depends on the specific task. In regression, the output spans the whole $\mathbb{R}$ domain and so we use a linear activation function: $f(x) = x$, $f'(x) = 1$. In the case of classification in two classes we can distinguish the case of the classes $\{0, 1\}$, where we use the sigmoid activation function (the smooth/continuous version of the step function): $f(x) = \frac{1}{1+exp(-x)}$, $f'(x) = f(x)(1 - f(x))$, or the case of the classes $\{-1, 1\}$, where we use the hyperbolic tangent activation function: $f(x) = \frac{exp(x)-exp(-x)}{exp(x)+exp(-x)}$, $f'(x) = 1 - f(x)^2$. If we have multiple classes, we use as many neurons as classes in the output layer. Classes are encoded with one-hot-encoding and the output layer uses a softmax unit to obtain a probability interpretation of the result. Why do we choose these functions? The universal approximation theorem guarantees that any nonlinear function can be approximated, with any arbitrary accuracy, with enough of these S shaped functions.*

- **Optimization and learning**: gradient descent, backpropagation, chain rule, vanishing gradient (ReLU)

Neural networks are nonlinear functions approximators. They depend on ***hyperparameters****, which are the details of a network architecture and training approach. In a feed-forward network the hyperparameters are the number and the disposition of neurons, the activation functions, the error functions, the optimizing algorithm, batch dimension, validation set dimension. To fix them we can procede with a trial and error or with cross-validation. Once we fix the hyperparameters the optimization of the network depends only on the weights. How can we optimize them? We want to start from the output of a network, backpropagate the error and based on this update the parameters. Suppose we have a training set* $(x_1, t_1), .., (x_N, t_N)$*, where* x_i *is the* i*-th input and* t_i *is the corresponding output. Suppose that our current neural network approximate the input-output relation function with* $g(x|\omega)$*, which means that* $g(x_n|\omega)$ *is the predicted output to the input* x_n*. We want the predictions of the network to be as close as possible to the real outputs associated to the inputs. In order to minimize this distance, we minimize the sum of squared errors. We obtain the optimal weights as* $\omega = \arg\min E$*, where* E *is the error function defined as:* $E = \sum_{n=1}^{N}(t_n - g(x_n|\omega))^2$*. To find the minimum of a generic function we can use an iterative method: the* ***gradient descent****. At each iteration we define the direction in which we move as the gradient* $(\frac{\partial E}{\partial w})$ *and we set the length of the step as* η*. We obtain the updating procedure:* $\omega^{k+1} = \omega^k - \eta \frac{\partial E}{\partial \omega}|_{\omega^k}$*. We may end in a local minimum and so it's convenient to run the optimization starting from different initial points. The key point is the evaluation of* $\frac{\partial E}{\partial \omega}$*. Using the* ***chain rule*** *we can see* $\frac{\partial E}{\partial \omega}$ *as a sequence of derivatives. To avoid the long computations we can take advantage of the feed-forward neural network structure. The derivatives in which we decompose* $\frac{\partial E}{\partial \omega}$ *corresponds to values that we can retrieve from the network. With a forward-pass we compute for each neuron the value of its activation function and the value of the derivative of the activation function. With a backward-pass we implement the backpropagation and we update the weights.*

Vanishing gradient and ReLU. *Because of the universal approximation theorem we know that the sigmoid or the hyperbolic tangent activation functions are good. However, we may encounter some problems. To optimize the weights of a network we use the backpropagation approach, which in other words is the gradient descent. In practice, we have to deal with the multiplication of the derivatives of the activation functions of every neuron we meet. The problem is that the derivatives of these activation functions are* < 1*, and so, if we have huge networks we keep multiplying numbers that are* < 1*. The result might be* ~ 0*, which leads to a vanishing gradient problem. We may end up with a deep network which does not learn. That's why we introduce an activation function which derivative is not* < 1 *(or* > 1*, because in that case we would have the opposite problem: an exploding gradient): the Rectified Linear Unit (ReLU),* $f(x) = max(0, x)$*,* $f'(x) = 1_{x>0}$*. Since this activation function is not differentiable, the universal approximation theorem does not work, but we have similar results for piecewise linear functions. With this activation function we obtain an efficient gradient propagation (no vanishing/exploding gradient). However, we have a drawback: the dying neuron problem. A neuron which has a negative input is set to zero, the output is set to zero and also the gradient is zero. This means that if a neuron is working by mistake on* $(-\infty, 0)$ *and we want to move it somewhere else we cannot. During training some neurons switch off and we cannot switch them on again. (To avoid: Leaky ReLU, ELU)* ✱

- **Neural networks for regression and classification**: maximum likelihood, error function

The error functions define the task. The MLE design error functions. The gradient minimize error functions.

Regression. *Is* $E = \sum_{n=1}^{N}(t_n - g(x_n|\omega))^2$ *the right/optimal error function to work on? We assume that the data* $t_1, .., t_N$ *come from a deterministic function plus some noise. The deterministic function is the one the neural network is trying to approximate, while the noise is supposed to be gaussian:* $\epsilon \sim N(0, \sigma^2)$*. In conclusion we have:* $t_n = g(x_n|\omega) + \epsilon_n \sim N(g(x_n|\omega), \sigma^2)$*. We want to fix the parameters (weights) according to the maximum likelihood estimation, i.e. we want to find the weights for which, assumed the previous model, the observed data are the most probable. Under these assumptions, the MLE of the network weights is* $\arg\min \sum_{n=1}^{N}(t_n - g(x_n|\omega))^2$*, and hence our choice for* E *as error function is right.*

Classification. *In the case of 0/1-classification we cannot use the previous model (since we cannot think about classification as a real term output plus some noise) however, we still want to use the maximum likelihood estimation principle. Wee see the output of the classification network as a random variable with 2 possible outcomes, and so, the model we choose is a Bernoulli*(θ)*, wehere* θ *tells us how likely the outcome will be 1. The model can be written as:* $g(x_n|\omega) = p(t_n|\omega)$*, with* $t_n \sim Be(g(x_n|\omega))$*. Then we look for the weights that maximizes the likelihood, and so we obtain* $\arg\min -[\sum_{n=1}^{N} t_n \log(g(x_n|\omega)) + (1 - t_n) \log(1 - g(x_n|\omega))]$*.*
In the case of multiple-class classficiation we extend the above error function and we obtain the crossentropy. If we encode the classes with the one-hot-encoding we can write the error function as $-\sum_{n=1}^{N} t_n^T \log(g(x_n|\omega))$*.*

✱ To avoid vanishing gradient we use ReLU everywhere but the last layer (at the end). In the last layer the activation function is task-oriented (sigmoid / tanh / softmax / linear)

3. Training a Neural Network

- **Model complexity and validation**: hold-out, leave-one-out cross-validation, k-fold cross-validation

*The goal of a neural network is to approximate a nonlinear function. There are 2 situations to avoid: underfitting and overfitting. Underfitting happens when we use a model which is too simple to describe the data: we end up with high bias and we lose some relevant informations about the data. Overfitting happens when we use a model which is too adapted to the data. This leads to high variance and a missed generalization (we'll obtain two very different behaviour with the training data and new data). To avoid these problems we need a way to detect them, and so we need a way to evaluate the quality of a model (**validation metods**).*

*If we have a lot of data we can divide the original dataset in 2 parts: training data and test data. Then we train the model on the training data and we check its validity on the test data. This procedure may result biased, since it depends on how lucky we are in the split training-test. To reduce this bias, a better approach is **cross-validation**. Suppose we have a dataset of n units. In the **leave-one-out** version we train the model n times on $n-1$ data, each time we "hide" 1 datum. At the end of each training we test the trained model on the datum we left out. The model error will be the average of all the errors. However this procedure loses efficiency when the dataset is too big. In the **K-fold** version we divide the dataset in K groups and then we train the model K times, each time we leave out one of the K groups. At the end of each training we test the trained model on the group we left out. Again, the model error will be the average of all the K errors.*

At the end of the cross-validation procedure we have K or n different models. We can average them and build an ensamble (generally, through average we get a better performance than with a single model). Another usage of the cross-validation is the hyperparameters tuning or model selection. Changing the number of hidden layers or the number of neurons per layer we create different architectures and so, different models. We can perform cross-validation on each model and then choose the structure that provides the lower error.

- **Preventing overfitting**: early stopping (with cross-validation and hyperparameters tuning), weight decay (weight regularization), dropout (stochastic regularization)

We overfit when we have a discrepancy in performances between the training set and the test set.

Complexity reduction. *A neural network is a nonlinear function approximator. The universal approximation theorem tells us that any nonlinear function can be approximated with any arbitrary accuracy if we put enough neurons with S-shaped activation functions. By increasing the model complexity, we increse the precision of the approximation. This can easily turn into overfitting. We want to obtain a generalization and not a perfect fit of the data. A possible way to fix it is reducing the number of parameters (layers or neurons per layer).*

Early stopping. *While training, the more iterations we do the more the training error will decrease. However, at some point it'll be overfitting. To prevent it, we can interrupt the training before its end. At which point? We divide the initial dataset into 3 groups: training set, validation set and test set. The test set won't be touched during the whole training process. It'll be used only at the end to evaluate the performance of the model. During the training phase we train the model on the training set. At the end of each epoch we test the obtained set of weights on the validation set. In this way we obtain 2 error curves: one for the training and one for the validation. We expect them to decrease together untill a certain point, after which the validation error curve will increase. The optimal stopping point is the moment in which the validation error curve reach its minimum. We can use early stopping also for model selection. Suppose to have 1 hidden layer with J neurons. For every value of J we have a model: we stop the training of the model when we reach the minimum of the validation error curve and we save the corresponding generalized error (which is the value of the validation error curve in the stopping point). The optimal value for J is the value for which the generalized error is minimum.*

Weight decay (weight regularization). *We can prevent overfitting by adding some constraints to the model: we force the weighs to take low values. Small weights improve the generalization of a network. We can put these constraints imposing an a-priori distribution on the weights: $\omega \sim N(0, \sigma_\omega^2)$. Following the MLE procedure to design the optimal error function (this time taking into account also the a-priori distribution of the weights) we end up with: $E = \sum_{n=1}^{N}(t_n - g(x_n|\omega))^2 + \gamma \sum_{q=1}^{Q}(\omega_q)^2$, where Q is the total number of weights and γ is the ratio between the variance of the noise and the flexibility of the model. We have a trade-off between fitting and regularization. The obtained error function is the idea of the Ridge regression. A slightly variation is the Lasso regression, which leads to: $E = \sum_{n=1}^{N}(t_n - g(x_n|\omega))^2 + \gamma \sum_{q=1}^{Q}|\omega_q|$. How can we set γ? It shouldn't be too high nor too small. The most convenient thing is to set it as an hyperparameter.*

Dropout. *Sometimes a neuron has a certain set of weights because of another neuron's. This might prevent a proper learning. We want weights to be independent. The idea of dropout is to turn off some neurons and to train the network without them. Precisely: at every epoch we switch off randomly some neurons (the probability of switching off a neuron j is p_j; we can add the switch-off probabilities to the hyperparameters) and we train the remaining network. A neuron which is off has all the weights set to 0 (in and out). Once we have a collection of models, we average them obtaining an ensamble (for which the weights are more likely to be independent).*

- **Weights initialization**

The final result of gradient descent is affected by weight initialization. Neither zeros or big numbers are good ideas. In the first case we sould have all gradient equal to zero and so no learning. In the second case, with an unlucky choice it might take very long time to converge. In the case of small networks we may initialize each weight with a gaussian distribution $N(0, \sigma^2 = 0.01)$ (higher the variance and longer to the convergence). However for large networks this approach may fail due to the vanishing gradient problem. A proposed solution in the case of large networks is the ***Xavier initialization****. Suppose we have an input x with I components, a neuron j with a linear activation function and the weights ω. The output of the neuron is $h_j = w_{j1}x_1 + .. + w_{jI}x_I$. If the inputs and weights have both mean 0 we obtain: $Var(w_{ji}x_i) = Var(w_{ji})Var(x_i)$. If we assume all the weights and inputs to be iid we obtain: $Var(h_j) = I \cdot Var(w_{ji})Var(x_i)$, which means that the variance of the output is the variance of the input but scaled by $I \cdot Var(w_{ji})$. If we don't want the variance of the weights to amplify/reduce too much the input (in order to avoid exploding/vanishing gradient) we set: $I \cdot Var(w_{ji}) = 1$. For this reason the Xavier initialization initializes each weight with a gaussian distribution $N(0, \frac{1}{I})$.*

4. Convolutional Neural Networks

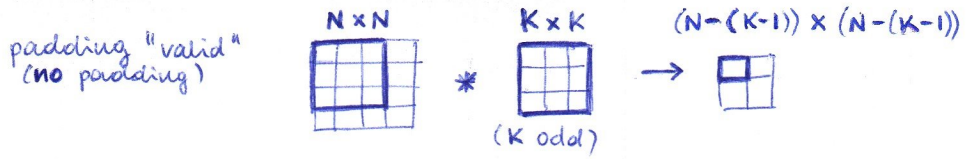
- **Image classification**: the task, nearest neighborhood classifier, linear classifier (settings, geometric interpretation, template matching interpretation), features extraction perspective (filters, convolution, padding)

Image classification *task: given an input image I, assign a label l from a fixed set of L categories. It's not straightfarward since there are some challenges: dimensionality (images are high-dimensional data), label ambiguity (a label might not uniquely identify the image), transformations (some transformations change the image radically, but not its label), inter-class variability (images of the same class may be very different).*
A first approach is the ***nearest neighborhood classifier****: assign to each test image the label of the closest image in the training set. The distance between two image is intendend as the pixelwise distance (for example the l^2 or l^1 norm). A slightly variation of this approach is the* ***K-nearest neighborhood classifier****: the idea of the distance is the same, but this time the assigned label is the most frequent among the K-closest images in the training set. These 2 approaches are very inefficient: even if there is no training time, the procedures become computationally demanding at test time. However, the main drawback is that they're not practical on images: using pixel differences to compare images is not appropriate (a translation of 1 pixel may lead to the conclusion that 2 almost identical images are very different).*

We introduce the ***linear classifier****. A classifier can be seen as a function that maps an image to a confidence score for each of the L classes. A good classifier associates to the correct class the highest score. We write an image as a n-dim vector x, the simplest classifier is given by: $S = Wx + b$, where S is the L-dim vector of the scores (S_i is the score for the class i), W is the $L \times n$-dim matrix of weights and b is the L-dim bias. How can we do it with a neural network? We consider x as the n-dim input of the network (and so we have n neurons as input). For the output we put L neurons (one per class). Connecting each input-neuron with each output-neuron we obtain that the weights of the neural network correspod to the weights of the matrix W.*
We give a ***geometric interpretation*** *to the linear classifier: at the end of the training we obtain $S_i = W_i x + b_i$ for every $i \in L$. If we consider $W_i x + b_i = 0$ we obtain an hyperplane: for every class, the classifier defines a hyperplane which divides the elements belonging to the class from all the other elements.*
Another interpretation: ***template matching****. Every row of the weights matrix has the dimension of the input image, so, every row W_i can be resambled as an image. We construct L images (one per row) and we call them templates. This means that when we perform $S_i = W_i x + b_i$ we're performing the correlation* * *between the image W_i and the input image x (then we add the bias). The network is learning templates to match the inputs.*

Features extraction perspective*. Images can not be directly fed to a classifier. We need to extract meaningful informations and to reduce the data dimension. We can decide ourselves the relevant features depending on the task (hand-crafted features), then we associate to each image a vector of features and procede with fitting a classifier. However, this procedure is strongly subjective. We want the features extraction process to be data-driven, exactly as the classifier. In fact, we want to train them together. To do so, we introduce filters (small matrices of weights (to be determined)) and convolutions (linear transformations). The goal is to build a convolutional neural network that will extract the relevant features and do the classification.*
When we do the convolution of an image against a filter we need to pay attention to the ***padding****. To make the convolution we center the filter on each pixel of the input image. Is every pixel of the input image reachable by the center of the filter? Centering a 3×3 filter (for instance) in a border pixel can create different scenarios. If we don't allow the filter to exit the input image, the output of the convolution will be an image with reduced size (no padding - "valid"). If we allow the filter to exit the input image in order to center every pixel of the original image, the output of the convolution will have the same size of the input image (half padding - "same"). If we allow the filter to exit the input image as long as there is still some intersection, the output of the convolution will have larger size than the input image (full padding - "full").*
Another variation to take into account is the ***stride****. If we make a convolution of an image against a filter, it's not mandatory to center the filter on every pixel of the input image. We can move skipping pixels (in this way the size of the output of the convolution will be smaller than the input image size). The stride denotes the way we move among pixels: stride $= k$ means that we're skipping $k - 1$ pixels.*

* We define the **correlation** of an image against a filter as a linear transformation of the image. To perform it, we slide the center of the filter on the image, multiply each weight of the filter by the pixel in the image (pixel-wise) and then sum all these products. If instead we want to perform a **convolution** of an image against a filter we flip the filter in both directions and then we apply a correlation.



- **Architecture of a Convolutional Neural Network (CNN)**
  An image passing through a CNN is transformed in a sequence of volumes. As the depth increases, the height and width decrease. A CNN can composed by the different type of layers.

  - ***Convolutional layer.*** *The convolutional layer focuses on increasing the depth of the input volume. The output of this layer is a mix all the input components. The covolutional layer uses filters that perform convolutions against the input image. Each filter has depth equal to the depth of the input volume (that's why we don't need to specify the depths of the filters). How can this increase the depth? The output of the convolution against a filter produce a slice in the ouput: putting many filters we get a deep output. The hyperparameters of this layer are: number, size, stride and padding of the filters. The parameters are the filters' weights and biases (if consider stride* $= 1$*, padding "same" and we use* N *filters of size* $K \times K$ *on an input of depth* D*, the layer will have* $N \times (K \times K \times D) + N$ *parameters).*

  - ***Pooling layer.*** *The pooling layer focuses on the spatial reduction of the input. We want deep but thin volumes: with this layer we thin the input image (the depth remain untouched). The pooling layer operates independently on every depth slice of the input and resizes it spatially. It's a downsampling operation. In particular, we can perform max pooling or average pooling, where the maximum and the average value is taken, respectively. We can see the choice between max or average as hyperparameter. This layer doesn't have parameters. Note that convolution layers increase the depth but don't necessarily change the spatial extent (they can thanks to padding or stride), instead, the pooling layers only reduce the spatial extent, they cant't modify the depth.*

  - ***Dense/Fully connected layer.*** *The fully connected layer operates on flattened input (a flatten layer simply flattens the input* $A \times B \times D$ *into a vector of length* $A \times B \times D$*). The FC layer is a layer of arbitrary many neurons, each one is connected to every element of the (flattened) input. If the FC layer is the last one, then it has as many neurons as the number of labels. If the FC layer is not the last one, the optimal number of neurons is an hyperparameter. The parameters of this layer are the weights of the connection (if we have a (flattened) inputs of* M *elements and a FC layer of* F *elements then the total number of parameters will be* $M \times F + F$*).*

- **CNN as special case of a Feed-forward Neural Network, the receptive field**
  Can we see a ***CNN as a special case of a feed-forward neural network****? Yes. We would like to translate in the "feed-forward neural network language" a convolutional layer which takes as input a* $N \times N \times 1$ *image and transforms it through one single* $K \times K \times 1$ *filter with stride* $= 1$ *and padding option "same". In the CNN architecture the number of parameters is:* $1 \times (K \times K \times 1) + 1 = K^2 + 1$*. If we build a feed-forward neural network we obtain: input size* $= N \times N = N^2$*, output size* $N \times N = N^2$*. The total number of parameters, considering the bias is:* $N^2 \times N^2 + N^2 = N^4 + N^2$*, which is far away from the number of parameters of the CNN. This is because to represent correctly a CNN through a feed-forward NN we have to share the weights and to sparse the connectivity. A* ***CNN*** *is a feed-forward NN with sparse connectivity and shared weights.*

  Receptive field. *Due to sparse connectivity, unlike in the feed-forward networks where the value of each output depends on the entire input, in CNN an output depends on a region of the input. This region of the input is the receptive field for that output. The deeper we go, the wider the receptive field is. Layers like pooling or convolutions with stride* > 1 *or padding that reduce the output spatial extent increase the receptive field.*

- **CNN training improvements:** data augmentation, transfer learning, popular architectures
  Data augmentation. *To prevent the model to learn wrong patterns in the images we augment the data. For instance: if we have images of dogs taken always from the same perspective we don't want the model to think that the perspective influences in being a dog. Data augmentation is a collection of image transformations that helps improving the training and prevents the overfitting. Possible transformations are: flip, rotation, random crop, color shift, noise addition, information loss, constrast change.*

  Transfer learning. *The model we consider has 2 parts: a feature extractor (which has the role of extracting meaningful informations from the data) and a classifier (which depends on our task). For the features extraction we can use pre-trained models. This can help when we don't have a lot of data or when we want to save training time. It can also be useful to avoid the trial-and-error process to determine the optimal hyperparameters that define the architecture. We can procede in two ways. Either we take a pre-trained model, consider its feature extractor as it is (we freeze the weights), we append our classifier and we train only the classifier part of the*

model (to do so we train the whole model with the weights of the feature extractor freezed). This works if we have few data or when we take a pre-trained model which task was similar to ours. Or, we take the feature extractor of a pre-trained model, we append our classifier and we train the whole model freezing part of the imported weights or none. In this way we inherit a useful architecture and a good initialization of it.

Popular architectures: ***LeNet-5*** *(one of the first CNN models, developed to identify handwritten digits. it combines: convolution, pooling and non-linearity (sigmoid, tanh)),* ***AlexNet*** *(architecture similar to LeNet-5, it has larger filters but with strides > 1, it has more dense layers, it introduces ReLU for non-linearity and dropout to prevent overfitting),* ***VGG16*** *(deeper variant of AlexNet, it has smaller filters and way more convolutional layers, this leads to fewer parameters and more non-linearities),* ***GoogleNet*** *(deep network, it introduces the inception block (block which performs multiple convolutions of different size simultaneously over the same input, then the outputs are concatenated into one)),* ***ResNet*** *(first very deep network).*

- **Fully Convolutoinal Neural Networks (F-CNN) and semantic segmentation**: F-CNN, semantic segmentation, solutions (heatmaps, only convolutions, bottleneck architecture (skip connections, U-net))

Fully Convolutional Networks (F-CNN). *What happens if we feed the network with a bigger image (than the ones we trained the network with)? The convolutional layers won't create issues, the fully connected layers will. Why? Because of a dimension problem. However, if convolutionals layers do not create problems, a good solution is to convolutionalize the fully connected layers (:= look at them as convolutions instead of a feed-forward NN). We can write the output of each fully connected layer as the convolution of the input against a 1×1 filter (the depth size depends on the input). We obtain a Fully Convolutional CNN (F-CNN: CNN wehere the fully connected layers were moved to a set of 1×1 convolutional filters). With a F-CNN we can take as input also bigger images, however, if we start with bigger images as input we end with something different than $1 \times 1 \times L$. The output will preserve some spatial extent: we expect something of dimension $M_1 \times M_2 \times L$. Basically, the output corresponds to a set of L images that we call* ***heatmaps****. Each pixel of the j-th heatmap tells us the probability of the pixel's receptive field to belong to the j-th class. Thanks to these heatmaps (which are in very low resolution) we can get a hint of which region of the input belongs to a certain class.*

Semantic segmentation *task: given an input image I, associate to each pixel a label l from a fixed set of L categories. Segmentation does not separate different instances belonging to the same class (that's instance segmentation). The required training set is made of pairs (I, GT), where I is an image and GT is a pixel-wise annotated image over the categories in L. A straightforward solution can be to use directly heatmaps predictions: we can assign the predicted label in the heatmap to the whole receptive field. However this solution is very poor in precision (since heatmaps are in low resolution). We would like to mantain somehow the full-resolution image and so we build a network which is fully convolutional and never reduces the spatial extent of the input (no pooling, no stride > 1, no padding reductions). In this way, as output, we get an image already of the size of the input. The drawback of this approach is that the receptive field associated to each output pixel is very reduced: we lose some (relevant) global informations. The best way of proceding is a* ***bottleneck architecture****: a contractive (/downsampling) part of the network will extract the global informations, a following upsampling part will recover the original spatial extent. The the first part of the network (downsampling part) is composed by a combination of convolutional and pooling layers. How can we implement the second part? How can we perform upsampling? We can fill the image according to the nearest neighbor: this approach is based on replications, not so good. Otherwise we can keep track of the locations during the pooling and use the positions from the pooling layer (we fill the rest with 0s). Otherwise we can perform a transpose convolution (convolution in which we have filters larger than the input). All of these are not optimal. The solution is* ***skip connections****. Instead of doing the entire upsampling from the last compressed representation of the image, we add connections between the downsampling and upsampling parts, allowing the exchange of informations and so, gaining precision. The connections are made between the corresponding dimensions in the down and up sampling. An example of this method is the* ***U-net****. The U-net is a network composed by a contracting path and an expansive path. The network is symmetric (bottleneck architecture). In the contracting part convolutions increase the depth while poolings reduce the spatial extent. When there is a pooling there is also a skip connection to the upsampling counterpart.*

- **Global Average Pooling (GAP)**: prevent overfitting, localization (CAM))

  One of the major drawbacks of the most powerful architectures is the large number of coefficients spent on the fully connected layers at the end. These very large fully connected layers make the network incline to overfitting. The idea is to replace all the fully connected layers simply by a ***Global Averaging Pooling*** *(GAP) layer. Note that this layer affects (/substitutes) only the FC part of the CNN. The GAP layer computes the average of each slice of its input volume and place it in a vector. Of course we need the number of the slices to be exactly the number of classes. The prediction after the GAP layer is performed by a soft-max. With this approach we use only the CNN to learn the classifier (getting rid of the huge amount of parameters given by the FC part). This is a big step to prevent overfitting (due to the substantial reduction of parameters).*

  Localization. *The advantages of the GAP layer extend beyond simply acting as structural regularizer that prevents overfitting. In fact, the GAP layer gives informations on which portion of the image leads the network to make a certain decision. It can be used to get an explaination of the network decisions, but more importantly, it can be used to localize the relevant regions for classifications (we obtain localization as a consequence of searching the region of the input that contributes the most for the classification). How can we pass from, for instance, the simple classification to a localization? For classification we have a CNN + FC (where FC classify). For the localization we can simply take the trained CNN + FC, remove the FC at the end, place a GAP layer and a new FC and then re-train the network to perform classification. In this way we get for free the* ***Class Activation Maps*** *(CAM). The CAMs are images which underlies the parts of the input images that influence most the network decision. The GAP layer allows the localization through the creation of CAMs. Notice that we're not training networks to have heatmpas, we train network for classification (CAMs $\neq$ heatmaps).*

- **Object detection**: the task, sliding-window solution, R-CNN, fast R-CNN, faster R-CNN

  Object detection *task: given a fixed set of categories L and an input image I, which contains an unknown and varying number of instances, draw a bounding box on each object instance. The background class has to be included. The required training set is made of annotated images with labels and bounding boxes for each object. Each image requires a varying number of outputs. A box is determined through 4 values: (x, y) which define the corner pixel and (h, w) which define the height and width of the box. A straightforward solution is the sliding window approach. We train a model on fixed input size, then we feed the trained network with larger images and we analyze each larger image piecewise. This is very inefficient. How should we choose the crop size? It's difficult to detect objects at diffretent scales. What if an object does not fit the crop? Because of these drawbacks we consider a* ***region proposal*** *approach. Region proposal algorithms are meant to identify bounding boxes that correspond to a candidate object in the image. Then we classify with a CNN the image inside each proposal region. That's the main idea of* ***Region Convolutional Neural Networks*** *(R-CNN). Given an input image the R-CNN extract the region proposals. To classify each proposed region we need to feed a CNN with a fixed size, so, we re-size every proposed region obtaining a wrapped region (this is not good since we're changing the proportions of the region). From here we perform a standard CNN for features extraction and then a classification. A peculiarity is that in the last part of the procedure we add a* ***Bounding Box regressor*** *(BB regressor) that refine and correct the bounding box estimate of the region of interest.*

  Fast R-CNN. *The whole R-CNN process is slow, that's why we introduce Fast R-CNN. In the R-CNN procedure we run the entire CNN for every proposed region. In this version the CNN is used only one time: the whole input image is fed to a CNN that extracts feature maps, only then the region proposals are identified and projected into the feature maps. Regions are directly croped from the feature maps rather than from the input image. This time we're extracting ROIs already in the CNN. This causes a problem since we need to feed the fully connected layers with the same input size. We introduce the ROI pooling layer: it extract a fixed size from the last convolutional layer for whatever selected region (this is done by max pooling). From here we perform classification through FC layers and again we put a BB regressor to improve the box estimate for the ROI.*
  Now that convolutions are not repeated we spend the majority of the time in ROI extraction.

  Faster R-CNN. *To reduce the time spent on the ROI extraction we introduce the Faster R-CNN. Instead of the ROI extraction algorithm, we train a region proposal network (RPN), which is a F-CNN. The goal of this network is to assign to each spatial location K different estimates of a possible bounding box. These estimates are called anchors. The training is divided in 2 parts (3 if we consider the initial dimension reduction): a classification part is used to predict the probability of a pixel being included in the instance for which we're creating the anchors, a regression part is used to adjust each anchor to better match the location.*
  ($\exists$ region-free methods (YOLO/You Only Look Once): object detection is treated as single regression problem).

- **Residual Learning**: ResNet(2015)

  In 2015 was build the first very deep neural network: ***ResNet****. The leading question was: is it possible to continuously improve the accuracy by adding more and more layers? It turned out that increasing the network depth does not always improve the performance, but this is not due to overfitting, since the same trend is shown in the training error. Consider the following case. If we have two networks, say A and B, A composed by 20 layers and B by 50, and A is optimal (we assume to know it), we expect the first 20 layers of B to be equal to A and the remaining 30 to represent some sort of identity. However, it comes out that the identity function is not easy to train. This means that if the optimal number of layers is N and we use M > N layers we won't learn optimality + identity. That's why ResNet introduces the* ***ResNet block****: we have some standard layers (convolutional, &.) but we also have the possibility of a skip connection (that represent the identity function). If the output of a previous block is already the best representation we can get, then in the current block we follow the identity path. Precisely: if $H(x)$ is the best representation for x in the $H(x)$'s block, then in the $F(x)$'s block $F(x)$ learns the residual, namely: $H(x) - x$. $F(x)$ does not learn anything that $H(x)$ already knows.*

- **Generative Adversarial Networks (GANs)**: autoencoders, generative models

  Autoencoders. *Autoencoders are neural networks used for data reconstruction (unsupervised learning). We want a network capable of learning inputs. We use a bottleneck architecture, putting together an encoder and a decoder. The role of the encoder is to extract meaningful informations about the data. The role of the decoder is to reconstruct the input data starting from its encoding. What the network is trying to minimize (loss function) is the distance between the reconstructed data and the original one. If this distance is reasonably small then we found a meaningful compact latent representation of the data: the encoding. The more deep the encoder part is, the more compact the representation will be (attention: if the encoder goes too in depth maybe the decoder won't be able to properly reconstruct the input). Autoencoders can be used to initialize a classifier (especially when we have few annotated data and many unlabeled ones). To do it we train the autoencoder in a fully unsupervised way using all the unlabeled data. Then we get rid of the decoder part and keep the encoder weights. We add a FC part for classifying starting from the latent representation and then we train the whole model on the supervised part of the dataset. Autoencoders provide a good initialization because their latent representation of the input already proved to be good. This is a good alternative to the pre-trained models (transfer learning).*

  Generative models. *We would like to generate similar images to the ones provided by a training set S. By "similar" we mean "coming from the same distribution". We introduce generative models: networks able to generate realistic images. One option is to use autoencoders. We can train an autoencoder on the training set of images, we discard the encoder part and we feed the decoder with random vectors which denotes the latent representation. However, for this to work properly we should know the distribution of the latent representation. vectors. In this way we're able to generate other vectors to feed the decoder with.*
  Based on another approach and idea, we introduce the ***Generative Adversial Networks*** *(GANs). The biggest challenge in generating fake images is define a good loss function able to tell when a generated image is similar to a real one. Instead of defining a loss function, GANs use two networks: a generator $\mathcal{G}$ produces fake images and a discriminator $\mathcal{D}$ judges the results and tell rather they're real or fake. The idea is to obtain such a good $\mathcal{G}$ that the generated images are not distinguishable from the fake ones through $\mathcal{D}$. (It's like training a generator of money and taking care of the police: at the beginning the police will be easily able to tell if the money are fake, but thanks to the feedback from the police the genarator can improve. Eventually the fake money will be so good that the police won't be able to distinguish them from the real ones.) How do they work?*

  - ***Generator*** $\mathcal{G} = \mathcal{G}(z, \theta_g)$*. We define a-priori a distribution ϕ_z, which we refer to as noise. We sample z from ϕ_z and we feed it to the network $\mathcal{G}$. The goal of $\mathcal{G}$ is to learn transformations that generates realistic images. The output will be a (generated) image. The network's parameters are contained in θ_g.*
  - ***Discriminator*** $\mathcal{D} = \mathcal{D}(s, \theta_d)$*. We feed the network $\mathcal{D}$ with an input image s. The goal of $\mathcal{D}$ is to learn transformations that lead to a proper classification of the input. It's a binary classification network: the output represents the a-posteriori probability of being real. The network's parameters are contained in θ_d.*

  A good discriminator is such that $\mathcal{D}(s, \theta_d)$ is maximum when s is a real image ($s \in S$) and $1 - \mathcal{D}(s, \theta_d)$ is maximum when s is fake ($s \sim \phi_z$). When the image is fake we can write: $\mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d)$. Training $\mathcal{D}$ consists in searching the parameters θ_d such that we have: $\max(\mathbb{E}_{s \in S}[\log \mathcal{D}(s, \theta_s)] + \mathbb{E}_{z \sim \phi_z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$. On the other hand, a good generator $\mathcal{G}$ wants $\mathcal{D}$ to fail, and thus training $\mathcal{G}$ means searching θ_g that minimizes the above: $\min \max(\mathbb{E}_{s \in S}[\log \mathcal{D}(s, \theta_s)] + \mathbb{E}_{z \sim \phi_z}[\log(1 - \mathcal{D}(\mathcal{G}(z, \theta_g), \theta_d))])$. The whole optimization ($\mathcal{D}$ and $\mathcal{G}$) is performed by an iterative approach: we fix one set of parameters and otpimize w.r.t. the other. Then we repeat changing the fixed set of parameters. How do we optimize? Gradient ascent for θ_d, gradient descent for θ_g.

5. Recurrent Neural Networks (and Sequential Data)

- **Recurrent Neural Networks (RNN)**: sequence modelling, adding memory via recurrent connections, back-propagation through time, vanishing/exploding gradient

Sequence modelling. *The feed-forward neural networks do not depend on the order we use for the inputs since the dataset is static. We end with a network which provides always the same output if fed with the same input. However, it might be that we have a temporal dependence in the dataset (dynamic dataset). This means that at time t the input might depend on what was the input at time $t-1$, $t-2$, .., till the first input. In this way also the output depends on the sequence of inputs we've seen in the past. Notice that dynamic data are not intended only as "temporal": a text can be viewd as dynamic dataset. A text is a sequence of characters composing words: the characters in a word are strongly correalted among each other. Also, a sequence of words may depend on a topic (context) and so also words may result correlated. There are 2 ways to deal with dynamic data: we can either use memoryless models or we can consider models with memory. Memoryless models use only the previous (/2 previous) input to predict the output. For example, autoregressive models have as output a weighted average of 1/2 previous times. This model has no memory: after 2 steps behind it does not remember anything. Another memoryless model is the feed-forward neural network. How can we add* ***memory****?*

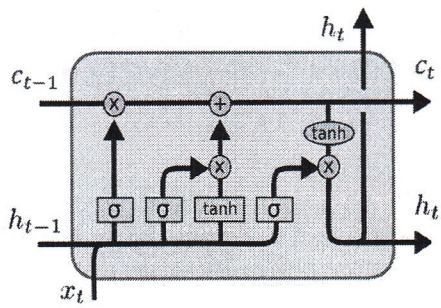
Recurrent Neural Networks (RNN). *Our goal is to create a structure comprehending: input that modifies the internal representation of the system, hidden state which evolves during time, output generated by the current internal state. The way to implement it is via a recurrent neural network. These networks break the feed-forward assumption and they have recurrent connections. In a RNN we start from the structure of a feed-forward NN and we add a feedback loop (which is the implementation of the memory). The feed-forward part and the feedback loop are connected and they influence each other. In this way the output has a part that depends on the current input and a part which depends on the story of the system. Notations: we store the weights for the static part in the matrix W, the ones going from the static to the dynamic in the matrix W_B, the weights for the dynamic part in the matrix V_B and the ones going from the dynamic to the static in the matrix V. We call the J activation functions of the static part $h_j(\cdot)$, the B activation functions for the dynamic part $c_b(\cdot)$ and the output activation function $g(\cdot)$. With this notation we resume a RNN with 1 hidden layer with:*

- $g^t(x_n|w) = g^t(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^t(\cdot))$
- $h_j^t(\cdot) = h_j^t(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}^t + \sum_{b=0}^{B} v_{jb}^{(1)} \cdot c_b^{(t-1)}(\cdot))$
- $c_b^t(\cdot) = c_b^t(\sum_{j=0}^{J} v_{bi}^{(1)} \cdot x_{i,n}^t + \sum_{k=0}^{B} v_{bk}^{(1)} \cdot c_k^{(t-1)}(\cdot))$

The model is difficult to train because we cannot use the backpropagation as we did in the feed-forward neural netowrk case (one of the conditions for backpropagation was to not have loops). We have to perform a ***back-propagation through time.*** *This means that we have to backpropagate the error following the path of the loop till the very first input. Basically we have to "unroll" the recurrent part (obtaining a feed-forward neural network). Notice that since we're unrolling, all the weights corresponding to the recurrent part are the same. How can we do it in practice? We unroll for U steps (it's not useful to unroll all the steps, we'll perform a truncated backpropagation). In this way the network doesn't learn any connection between events that are more than U steps apart. Then we initalize V and V_B (which are the weights that we replicate) to have the same replicas. Finally we compute the derivative of the output w.r.t. each of the gradients and we update the weights. The update of V and V_B is not done independently: we update all the replicas together with the average gradient through time. In principle we would like to have U as large as possible, however RNN cannot learn too many steps back in the past. This is because of the* ***vanishing gradient*** *problem (which occurs whenever we have a deep network). When we use backpropagation we have to deal with the multiplication of the derivatives of the activation functions of every neuron we meet. In our case we have as many products as the number of steps we go in the past. The derivatives of the activation functions (such as sigmoid or hyperbolic tangent) are < 1, and so, if we keep multiplying them we get a gradient which is ~ 0. This is a problem since the gradient tells us which direction to take to optimize our network. A possible solution is to change the activation functions. Activation functions such as sigmoid or hyperbolic tangent are very efficient (universal approximation theorem), but the maximum values of their derivatives is 0.25 for the sigmoid and 1 for the hyperbolic tangent. If we switch to the* ***ReLU*** *activation function we force all gradients to be either 0 or 1. Since the derivative of the ReLU is either 0 or 1 we don't have problems with vanishing or exploding gradient (which is also a situation to avoid: if we consider an activation function which derivative is > 1 we end up with a huge gradient), however this is not yet an optimal solution because a linear activation function only accumulates the inputs. Instead of just accumulating inputs, we want to have a "proper" memory.*

- **Long Short-Term Memories (LSTM)**

Another (and more efficient) way to implement the memory and avoid the vanishing gradient problem is through ***Long Short-Term Memories*** *(LSTM). The main idea is: changing the activation function of the recurrent part of the RNN in a linear function (≠ ReLU) we learn how to accumulate inputs (since we just accumulate them we call the cumulative component* ***Constant Error Carousel*** *(CEC)), if we can add a mechanism which will manage the accumulated inputs we can create an optimal memory. A good mechanism is a 3 gates mechanism: informations get to the memory cell through the "write" gate, informations stay in the memory if the gate "keep" allows it, informations are read if the "read" gate is open. With this approach the informations can propagate as much as we want (we have no vanishing gradient problem thanks to CEC). Each input enters a hidden state where also the CEC enters (coming from the previous hidden state). In the current hidden state the input and the past are mixed to produce the output and the update of the CEC. How does it work at time t?*



1. based on the current input and the previous state we determine what to write in the memory and if it is worth to write it all

- ***Write the memory.*** *What do we do when we want to add a new information to the memory? We have to decide if the information is worth being saved. We take the new input x_t and the previous* ***state*** *h_{t-1} and we determine $\tilde{C}_t$ as the hyperbolic tangent of a weighted combiantion of x_t and h_{t-1} (plus a bias): $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$. This is the information that we want to add to the memory. We define the index i_t as a sigmoid function which takes as input a weighted combination (plus bias) of x_t and h_{t-1}: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$. This index is in $[0, 1]$ and tells us how much of $\tilde{C}_t$ will be added to the memory.*

2. based on the current input and the previous state we determine how much of the memory we want to keep (we erase the rest)

- ***Keep/clean the memory.*** *We may decide to keep the whole memory or to clean it, even only partially. We define the value f_t as a sigmoid which take as input a weighted combination (plus bias) of x_t and h_{t-1}: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$. This value is in $[0, 1]$ and tells us how much of the memory we want to keep. If $f_t = 0$ we're resetting the memory, if $f_t = 1$ we're keeping the whole memory. Once we know how much of the memory we want to keep, what and how much of a new information we want to add, we can update the memory: $C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$.*

1. and 2. update the memory at each step.

3. based on the current input and the previous state we determine how much to read from the memory. The output is a mix of the memory and the input

- ***Read from the memory.*** *How can we use the informations in the memory? We want the output at time t to be a mix of the memory and the input. We define the value o_t as a sigmoid function which takes as input a weighted combination (plus bias) of x_t and h_{t-1}: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$. This value is in $[0, 1]$ and tells us how much we want to read from the memory given the input x_t and the previous state h_{t-1}. Notice that we don't take the memory as it is, we put the current memory C_t into a hyperbolic tangent function. We finally have the output at time t: $h_t = o_t \cdot \tanh(C_t)$.*

Of all this structure, what we learn during the backpropagation is: $[W_C, W_i, W_f, W_o, b_C, b_i, b_f, b_o]$.
An alternative (even if not remarkably better) is the ***Gated Recurrent Unit*** *(GRU). It has less gates than LSTM and it also re-uses a gate twice. It combines the "write" and "keep/clean" into one: "update" gate. However, the performance is not very different, so we remain on LSTM. Now that we have defined it, we can talk about* ***LSTM networks****. We can build a sequence of networks: each one has the input, an LSTM and the output. This sequence is connected through the LSTMs, so that it is possible to backpropagate the error. In this settings, the whole LSTM part works as a feature extractor which aims to find an optimal representation of the input sequence. We can have also more complicated collection of networks: for example the hierarchical one. We can build a sequence of networks such that each one is composed by an input, a sequence of LSTMs, a ReLU activation layer and an output. The LSTMs of the same level connect the networks together. In this way we learn a hierarchical representation of the the input sequence. If we have the entire sequence available at the beginning we can also create bididrectional LSTM networks. We consider 2 collections of networks: one reads the sequence from left-to-right, the other reads from right-to-left. The LSTMs of the left-to-right are connected among each other and the same for the LSTMs of the right-to-left. Once we obtain 2 representations of the input sequence (one from the left-to-right flow and one from the right-to-left flow) we concatenate them. Then we go through ReLU layers and we produce the final output sequence.*

- **Seq2Seq and beyond**: sequential data, Seq2Seq, attention mechanism, attention mechanism in Seq2Seq

RNNs are used to build different architectures, each for a different sequential data problem. We divide architectures in 5 groups. [1] *One-to-one (e.g. image classification) is the traditional architecture of the neural network: it has a fixed-sized input and output.* [2] *One-to-many (e.g. image captioning: the input is an image and the output is a sequence of words describing the image) has one input and produce a sequence of varying length as output.* [3] *Many-to-one (e.g. sentiment classification: classifies a sequence of words into a positive/negative sentiment) has a varying-length input and produce one single output.* [4] *Many-to-many with input and output of varying, but same, length (e.g. word-word translation: the input is a sequence of words, for each one we get an output (in sync)).* [5] *Many-to-many with input and output not in synchrony and of varying-length (e.g. text translation: the input is a sequence of words and the model waits to read the whole input to produce the output).*

Seq2Seq. *We consider the not-in-synchrony many-to-many architecture, which goes under the name of Seq2Seq. The architecture is a encoder-decoder. The encoder takes as input a sequence and encodes it. Based on the encoding of the input, the decoder produces the first output, then the decoder feeds the output of each time step as input to the next one. To let the network understand when the encoding is done (so that it can starts to produce outputs) we use the special character <SOS>. Other special characters are: <PAD>, used to adapt the lengths of the elements of the sequence (all the input elements must have the same length), <EOS>, produced by the decoder when the output sequence is complete, <UNK>, which replaces rare elements and causes the network to ignore these elements. How do we* ***train*** *this model? We take batch_size pairs of [source_sequence, target_sequence]. We add <SOS> at the end of source_sequence and <EOS> at the end of target_sequence. We pad to the max_input_length and max_target_length. We encode each token (e.g. word embedding if the Seq2Seq is a text translator) and, in case, we replace the rare elements with <UNK>. The optimization is perform as: given a pair [S, T] = [source, target] we want the network to produce T' that is as close as possible to T. Hence, we can see the prediction problem as a classification problem (during the sequence).*

Attention Mechanism. *An augmented version of the RNN is the* ***Neural Turing Machine****, which connects a RNN with an external memory (array of vectors (represented by arrows)). The RNN can read from the memory and write on it (modify it). In this way, the output depends on the RNN and also on what is written in the memory. To train a neural network we perform backpropagation (compute derivatives) and so we have to find a way to describe the "read" and "write" operations in a differentiable way. Moreover, we want to be able to write and read from the whole memory, but at each step we want to focus only on the relevant parts. This "everywhere but especially from here" is called* ***attention mechanism****. Instead of specifying a single location, the RNN produces an attention distribution that describes how much we read/write at every location. The result of the read operation is a weighted sum of the content of the memory: we read* $r = \sum_i a_i M_i$*, where* a_i *is how much (in %) we read from the memory-cell* M_i*. Similarly, the result of the write operation is that we rewrite every memory-cell* M_i *with a weighted sum of the value that we want to write and the previous value of the memory:* $M_i = a_i w_i + (1 - a_i) M_i$*, where* a_i *is the attention weight associated to the i-th memory-cell. Both results are differentiable, so we have no problems with backpropagation. However, how does the network decide the attention distribution? It uses a combiantion of 2 methods: content-based and location-based attentions (used in this order).* ***Content-based attention*** *tells the network to focus on the memory-cells that are similar to the current input element (elaborated by the RNN): more attention is assigned to cells that have high similarities with the input.* ***Location-based attention*** *allows movements in the memory: the network has to pay attention also to the neighbours of the input element since its output may depend on the context.*

Attention mechanism in Seq2Seq. *A limit of the Seq2Seq model is that when we have very long sequences the encoding may fail to compress all the informations. To generate the target sequence the decoder starts from the embedding of the whole source sequence. The embedding makes no distinctions among the relevant informations of the sequence. We would like an attention mechanism that leads to a target-driven embedding of the sequence. How? We introduce an attention distribution over the previous hidden states. At inference time, the RNN takes as input an element t and compares its hidden state* h_t *with all the other hidden states* $\{h_s\}_s$ *(dot product), obtaining a vector of scores* $\{\alpha_s\}_s$*. Each score represents the similarity between* h_t *and* h_s*. Through a softmax we transform the scores in weights* $\{w_s\}_s$*. The embedding of the previous inputs is computed as a weighted sum of the previous hidden states:* $c_t = \sum_s w_s h_s$*. This embedding is called* ***context vector****. Now the decoder generates the output starting from a combination of the context vector and the current hidden state. This application of attention mechanism to Seq2Seq models can be applied to translation, voice recognition, image captioning and response generation (chatbots).*

- **Chatbots** - attention in response generation

*Chatbots can be define on 2 dimensions: core algorithm and contex handling. The **core algorithm** is how we implement the dialog. The algorithm can be of 2 types: generative or retrieval. The generative algorithm encodes the question into a context vector and generates the answer word by word using conditioned probability distribution over answers' vocabulary. The retrieval algorithm relies on a knowledge of question-answer pairs: every new question is encoded into a context vector and using similarity measures we get the top K neighbors in the knoledge base. The generative algorithm is the most advanced, however the training is very demanding. The **context handling** is what we take into account when we implement the dialog. We can select either a single-turn or a multi-turn approach. The single-turn associates a new question to a single input vector. The training is made by pairs:* $[question_i, answer_i] = [q_i, a_i]$. *This procedure may lose important information about the history of the conversation and generate irrelevant answers. Instead, the multi-turn associates to a new question an input that considers also the context. The training is made by:* $[q_{i-k}, a_{i-k}, .., q_{i-1}, a_{i-1}, q_i, a_i]$.

Hierarchical generative multi-turn chatbots. *To obtain a chatbot we can apply a Seq2Seq model to the conversation between two agents. Generative chatbots use RNNs and train them to map questions to answers. How can we implement the multi-turn? We may concatenate multiple pairs of question-answer into a single long input sequence, but this would probably perform poorly. The optimal way to implement the muli-turn is through a hierarchical attention mechanism (we pay attention hierarchically: characters → words → sentences). We consider the input at time i:* $[q_{i-k}, a_{i-k}, .., q_{i-1}, a_{i-1}, q_i, a_i]$. *The input has* $k+1$ *questions and* $k+1$ *answers. For each question/answer we compute a first level of attention: the **word attention**. Based on this, we embed the* $2(k+1)$ *elements in a vector: the utterance encoding. Here we perform a second level of attention: the **utterance attention**. From this we obtain a context vector, which, combined with the previous outputs, goes through the decoder and generates the output.*

- **Transformer**

*Is the attention mechanism build upon RNN? The main idea of the transformer is to implement an attention mechanism without recursions (so that we can parallelize the training of models for sequential data). Differently from the RNN, which reads an input sequence token by token, the transformer takes the whole sequence as input and puts it through a pile of encoder blocks followed by a pile of decoder blocks. The **encoder block** takes a sequence and puts it through a self-attention mechanism, which processes each word taking into account the dependencies inside the sentence. What the self-attention does is associating a weight to every word and then it defines the embedding of a word as a weighted sum of the others. Once it is done, each processed word goes through a feed-forward neural network. Here we have a parallelization: all the words enter the feed-forward networks independently, without waiting for the previous. Notice that, we don't want to lose the sequential information about the data. The previous operations mix and transform the words and so, to keep tracking of the positions, we associate to the very first input words a positional encoding vector.Once we get out of the encoder block we may enter in an other encoder block or we may enter in the **decoder block**. The decoder has the same structure as the encoder, with the only difference of a "encoder-decoder attention" layer in the middle. This is because each decoder has an attention mechanism that tells the attention to pay at the input based on the encoders. Once we get out of the decoder block we may enter an other decoder block or in the final softmax layer. Here, the last decoding of each word is projected on the space generated by the vocabulary. The softmax transforms these projections in probabilities and so we can select the cells with higher probabilities.*

- **Word embedding**: one-hot-encoding, N-gram, word embedding, neural-net language model, Google's word2vec
How can we translate words in numeric data? How can we learn words representation? One way is ***one-hot-encoding****. We generate a vocabulary as a huge-dimensional vector of all words (one cell per word). Then, each time we have to translate in numbers a word we create a vector of zeros of the dimension of the vocabulary and we put a* 1 *in the word-corresponding position. In this way we can see a text as a bag of words: we sum (and normalize) all the vectors corresponding to all the words in a text and we get a single vector (bag of words) which tells the frequencies of the vocabulary's words in the considered text. This approach has a some drawbacks: every word is represented independently of the others (we're losing informations about the sequence) and it results orthogonal to all the others (we can't tell if there are similarities/dependencies among words). Moreover, vocabularies are often very big and so we may encounter dimensionality problems. Another approach is the* ***N-gram****, which is about modelling the probability of one word given the previous* N*. Due to conditioning, we can obtain the probability of a word as the product of the probabilities of each previous word conditioned to their previous ones. However, to compute the probability of one word we have take into account all the possible combination of the previous words. This leads to a curse of dimensionality. Moreover, also with this approach we are not able to tell the similarity/correlation between words. That's why we introduce the* ***embedding****. The idea is to take these sparse vectors representing words and project them in lower dimensional spaces. In this way we fight dimensionalily probelms and, in addiction, we get a similarily measure: closedness. The embedding is linked to the idea of the* ***autoencoder****. Shortly: the goal of the autoencoders is to learn a reduced representation of the input (they do it by training an encoder-decoder architecture which maps an input vecotr into a hidden (reduced) representation and then maps this hidden representation back; the loss function considers both the precision in the reconstruction and the dimensionality of the encoding).*

Neural Net Language Model. *Given the idea of the autoencoder, we want to apply it to solve the N-gram problem. We build a model composed of the input, projection, hidden and output layers. The idea is to take as input* N *tokens and put them through the projection layer. This layer's goal is to reduce the dimension of the input. Then, in lower dimensions, we perform a standard classification to predict which is the most likely output given the input. What about complexity? The training has a complexity of* $[N \times m] + [N \times m \times h] + [h \times |V|]$*. The first component is given by the encoding, where* N *is the number of tokens and* m *is the dimension of the smaller space on which we project the input. The second component is given by the hidden layer, characterize by the dimension* h*. The last component is given by the softmax that we put at the end.*

Google's word2vec. *For a proper embedding approach we think about context. This is the idea of word2vec. Given an input, instead of focusing on the prediction of the following word, we remove one word an we try to predict it from the surrounding words. In this way we interpret the projection on the smaller-dimensional space as the extraction of the word's features. This projection is based on the surrounding: we want to find a representation of the surrounding words good enough to predict the missing one. In this way, the encoding of a word is the feature extraction of the tex surrounding. This is inspired by the idea that the meaning of a word is written in the words that surround it (2 words surrounded by the same text are more likely to be related or even the same). The word2vec model is composed by* 2 *parts. The* ***skip-gram*** *architecture takes a word (the one-hot-encoding version), project it and from the projection predict all the possible surrounding words. The* ***continuous-bag-of-words*** *architecture takes all the surrounding words (the one-hot-encoding version), sums them and projects the sum, obtaining an encoding which allows the reconstruction of the input word. These* 2 *architectures form a sort of autoencoder. In this way the optimization is computed through a minimization of a loss function which tells how far the reconstructed word is from the original. The word2vec model shows improvements in terms of complexity, which now is* $[N \times m] + [m \times \log |V|]$*. Because of its idea, this model shows regularities in the embedding space, for example the word "king" is close to "queen", same for "prince" and "princess". Some applications of word2vec are document classification/similarity and sentiment analysis.*