11. Artificial Neural Network (ANN)

Artificial Neural Network = all possible ways in which we can create a network in which units are connected together.

general family of networks

Neural Network ≈ more general way to approximate a function.

GOAL: Estimate some function $f : X \mapsto Y$ where Y can be $Y = \{C_1, \dots, C_K\}$ or $Y = \mathbb{R}$

DATA: $D = \{(\vec{x}_n, t_n)_{n=1}^{N}\}$ such that $t_n \approx f(\vec{x}_n)$

FRAMEWORK: define $y = \hat{f}(\vec{x}, \vec{\theta})$, we define a function based on some parameter θ and we WANT TO LEARN IT in order to approximate the real f.

This definition is the same of the one seen from linear models. ANN draw inspiration from biological system (brain structures); THERE IS A PROCESS UNIT called NEURON connected with many other NEURONS.

The complexity of the model is not given by a single unit but from the many connections that there are

Feed Forward Network

FNN

FEED FORWARD: SIGNALS go FROM INPUT LEVEL to OUTPUT LEVEL only in one direction. Information flows from input to output without any loop

↓

No connection backword

NETWORK: f is a composition of elementary functions in an acyclic graph

WE MAKE THE ASSUMPTION OF A LAYERED ARCHITECTURE!

(*) $$f(\vec{x}, \vec{\theta}) = f^{(3)}(f^{(2)}(f^{(1)}(\vec{x}, \vec{\theta}^{(1)}); \vec{\theta}^{(2)}), \vec{\theta}^{(3)})$$

where $f^{(m)}$ is the m-th layer of the network; $\vec{\theta}^{(m)}$ the corresponding parameters

FNNs are CHAIN STRUCTURES. How do we have the connections? We can have many models, the generic model CALLED DENSE provides that each unit in the current layer is connected with all the units in the next layer.

We can consider each layer as a computation of a partial function. FNN is a composition of previous functions (*).

The lenght of the chain of a FNN is the DEPTH of the network and the last layer is called the output layer.

DEEP LEARNING: FOLLOWS FROM THE USE OF NETWORKS WITH A LARGE NUMBER OF LAYERS (large depth, more than 3 layer)

We have many parametric models, such as kernel methods, but we have to choose the kernel and the parameters, then the problem can be solved. So **Why FNNs?**

→ You don't have to define kernel

→ No parameter tuning

Non convex problem → more difficult, it cannot be solved with standard optimization method (we need to use stochastic gradient descent).

↓

complex combination of many parameters.

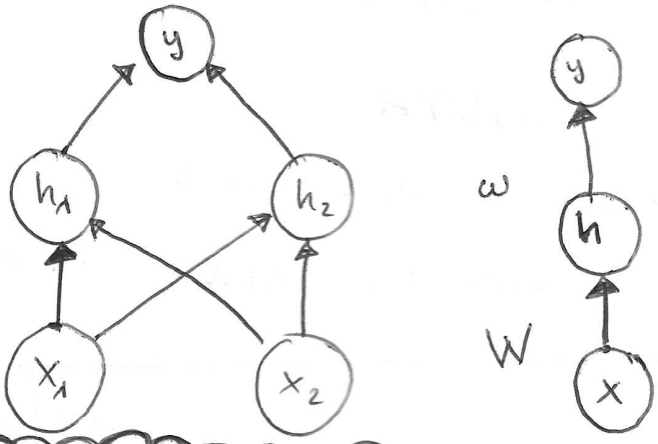
Deep Neural Network is an old concept. In these recent years those networks became interesting because of HIGH AMOUNT of DATA and HIGH COMPUTATIONAL RESOURCES. Before the usage of neural networks, human expedives were needed in a preprocessing phase to understand the correct set of features.

Example: learning a XOR function

The XOR function can be represented by a 2D input and 1D output.

$$D = \{((0,0)^T, 0), ((0,1)^T, 1), ((1,0)^T, 1), ((1,1)^T, 0)\}$$

THIS IS A NON LINEAR SEPARABLE DATASET, IT CANNOT BE SOLVED WITH A LINEAR MODEL. Let's define another model to solve the problem, by introducing two new variables h_1 and h_2:



$$h_i = g(\vec{x}^T W_{:,i} + c_i)$$

with $g(\alpha) = \max(0, \alpha)$

OUTPUT: $y = \vec{w}^T \vec{h} + b$

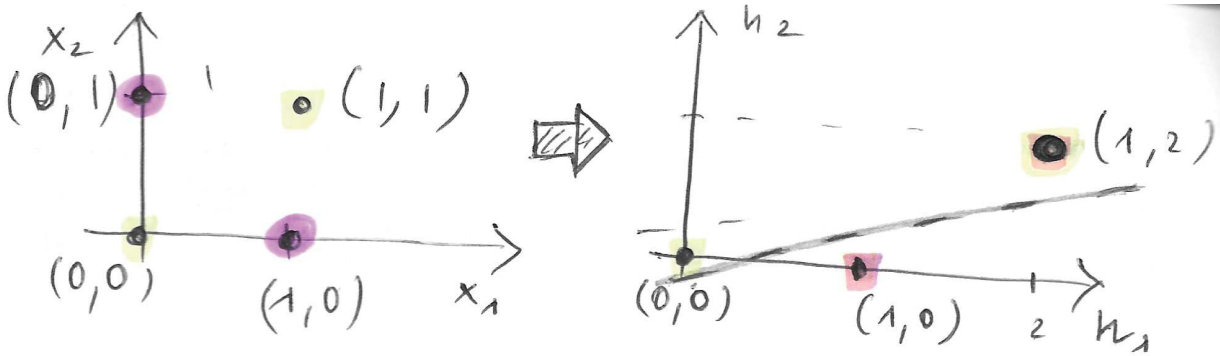
FULL MODEL!

$$f(\vec{x}, W, \vec{c}, b, \vec{w}) = \vec{w}^T \max(0, W^T \vec{x} + \vec{c}) + b$$

each h_i is an activation function applied to a linear combination of the input.

The solution is $W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$, $w = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$, $b = 0$

If we compute now h_1 and h_2 for each possible combination of the input values and we represent them this is what we get!



= 1

= 0

ORIGINAL SPACE

LEARNED h SPACE

Data now are linear separable!

THE NETWORK ITSELF WAS ABLE TO UNDERSTAND THE RIGHT TRANSFORMATION FROM X TO H, to ensure linear separability

↘ No tuning parameters.

Overall structure of the network:

- How many hidden layers? DEPTH
- How many units in a layer? WIDTH
- Which kind of units? ACTIVATION FUNCTIONS
- Which kind of cost function? LOSS FUNCTIONS.

Note!

With kernel methods we have to choose the kernel for any dataset, since if you change dataset the old chosen kernel may not work. THIS IS NOT the case with neural network.

For the above questions there is no a theoretical support, the good answers can be done by practice.

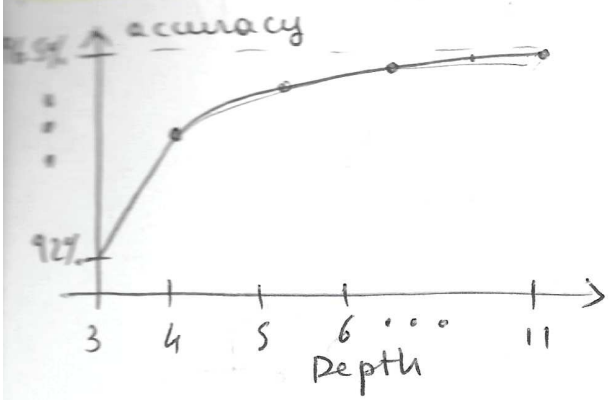
How many hidden layers? DEPTH

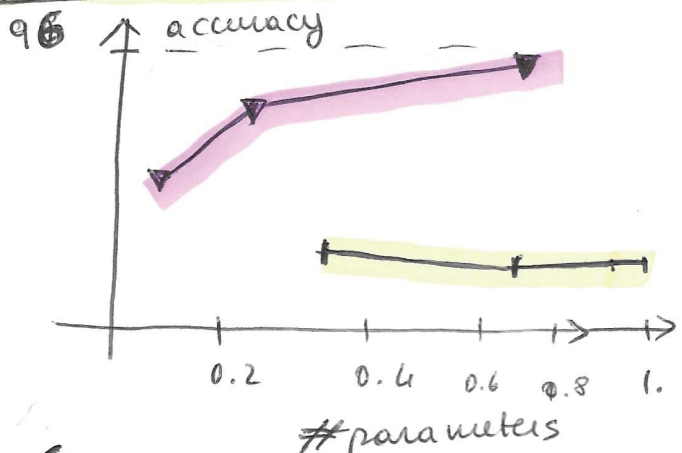
UNIVERSAL APPROXIMATION THEOREM:

FFN can approximate any function with an arbitrary precision, with only one layer, but enough hidden UNITS.

The Universal approximation theorem does not say how many units. In general, the unit number is EXPONENTIAL IN THE INPUT SIZE. In theory a SHORT and WIDE NETWORK can approximate any function, BUT IN PRACTICE THIS IS NOT TRUE.

> In practice, a deep and narrow network is EASIER TO TRAIN and PROVIDES better result in generalization



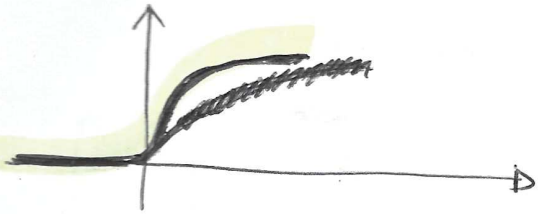


even if you increase param. you don't increase perform.

- 11 layers, convolutional
- 3 " 4 ", fully connected

About the kind of units and the cost function, these two questions are related each other, in particular THE CHOICE of THE ACTIVATION FUNCTION IN THE OUTPUT LAYER IS PARTICULARY IMPORTANT. We are using the gradient descent (compute the gradient and moving into the direction of the gradient by a quantity that is specified by the learning rate), the problem is that there are some activation functions that produces "areas" in the error function that are FLAT, so THE ERROR IS ZERO and we don't have any indication on which is the right direction to go.

SIGMOID ACTIVATION FUNCTION (goes from zero to one) SATURATES in most regions.



→ produces the error function with flat parts.

We cannot increase too much the learning rate, because it converge only if the learning rate is small enough.

- The solution stands in the right combination of the activation function (specially the one in the output layer) and the cost function.

The most common way of defining the cost function is to CONSIDER THE MAXIMUM LIKELIHOOD PRINCIPLE:

$$J(\theta) = -\ln\left(P(\vec{t}\,|\,\vec{x}, \vec{\theta})\right) \quad \text{(cross-entropy)}$$

$J(\theta)$: cost function.

$P(\vec{t}\,|\,\vec{x}, \vec{\theta})$: probability distr. of the output given the input and the parameters of the model.

When we assume gaussian noice we have:

$$P(\vec{t}\,|\,\vec{x}, \vec{\theta}) = \mathcal{N}\left(\vec{t}\,|\,f(\vec{x}; \vec{\theta}), \beta^{-1} I\right)$$

and the cost function corresponds to:

$$J(\vec{\theta}) = \frac{1}{2}\left(\vec{t} - f(\vec{x}; \vec{\theta})\right)^2$$

MAXIMUM LIKELIHOOD ESTIMATION WITH GAUSSIAN NOISE corresponds to MEAN SQUARED ERROR MINIMIZATION.

We distinguish three cases depending on the problem we are considering:

- REGRESSION
- BINARY CLASSIFICATION
- MULTI-CLASSES CLASSIFICATION

REGRESSION

Regression means that I want a real number as output, so I can consider LINEAR UNITS. Using a Gaussian distribution as noise model, we get the situation seen before where our cost function is THE MEAN SQUARED ERROR.

LINEAR UNITS: Good because do not saturate ⟶ NO FLAT REGIONS

BINARY CLASSIFICATION

The output should be between zero and one, so what we use is the sigmoid function; the output unit will also have a sigmoid activation function:

$$y = \sigma\left(\vec{w}^T \vec{h} + b\right)$$

The likelihood corresponds to a Bernoulli distribution.

$$J(\theta) = -\ln\left(P(t|\vec{x})\right) = -\ln\left(\sigma(\alpha)^t\right)\left(1-\sigma(\alpha)\right)^{1-t}$$

$$= -\ln \sigma((2t-1)\alpha) = \text{softplus}((1-2t)\alpha)$$

$$\alpha = \vec{w}^T \vec{h} + b$$

Even if the sigmoid function has problem, because it saturates, when you compute the logarithm of the error function, it compensates for the exponential and the problem of flat surfaces comes only when we are close to zero, meaning when it gives the correct answer.

MULTICLASS CLASSIFICATION

You will have an output vector with K components, where K is the number of classes.

SOFTMAX UNITS: Softmax activation function

$$y_i = \text{softmax}(\alpha)_i = \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)}$$

The softmax function is a function where you have an exponential and you divide that exponential by the sum of all exponential, in such a way the sum of the all the numbers is one.

LIKELIHOOD CORRESPONDS TO MULTINOMIAL DISTRIBUTION:

$$J(\vec{\theta})_i = -\ln \text{softmax}(\alpha)_i = \ln \sum_j \exp(\alpha_j) - \alpha_i$$

The unit saturates only when there are minimal errors.
How should I design my network? In summary!

- Regression → linear output unit ⊕ mean squared error loss function
- Binary classification → sigmoid output unit ⊕ binary cross entropy
- Multi-class classification → softmax output unit ⊕ categorical cross-entropy.

With these combination the gradient descent works well.
For the hidden unit is very difficult to decide, you have many choices, some intuitions, no theoretical principles.

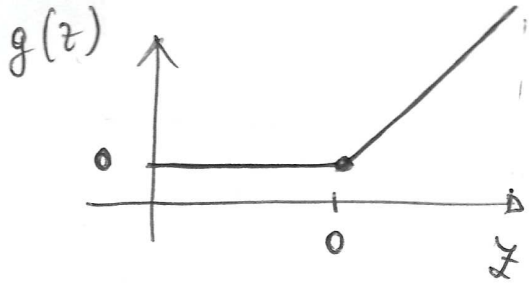
PREDICTING WHICH ACTIVATION FUNCTION WILL WORK BEST IS USUALLY IMPOSSIBLE. At this moment everyone suggest to use!

current recommendation

Rectified Linear Units = ReLU : $g(\alpha) = \max(0, \alpha)$

Note: if all units are linear, the overall network is linear, ReLU is not linear (easy to compute and to compute the derivative).

A problem is that it is not differentiable in 0, but does not cause troubles because is very difficult to get to 0.

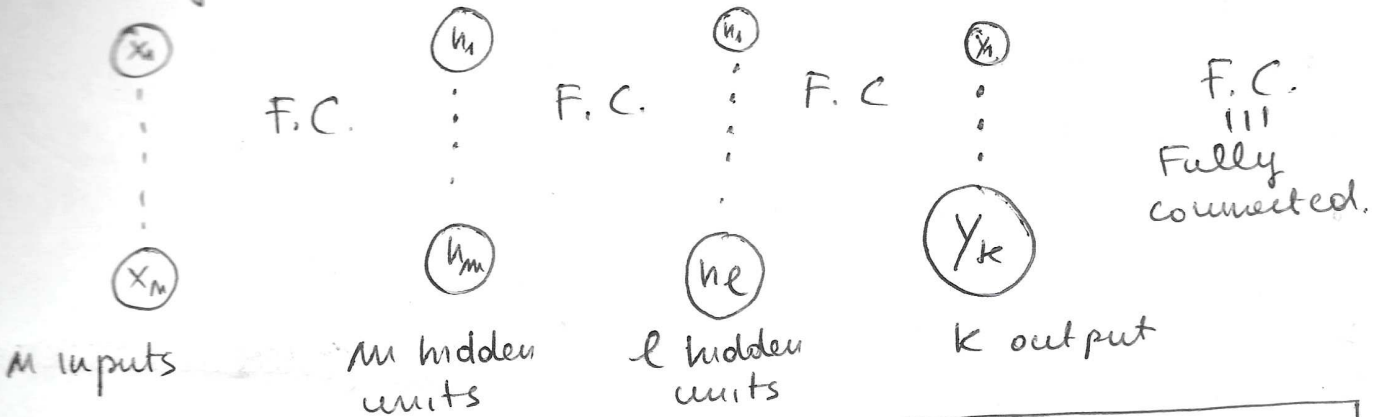


⊕ EASY TO COMPUTE,
⊕ EASY TO DERIVE,
⊕ SECOND DERIVATIVE IS ZERO

Alternatives!
- sigmoid and hyperbolic tangent
- What is important is to not introduce flat actions.

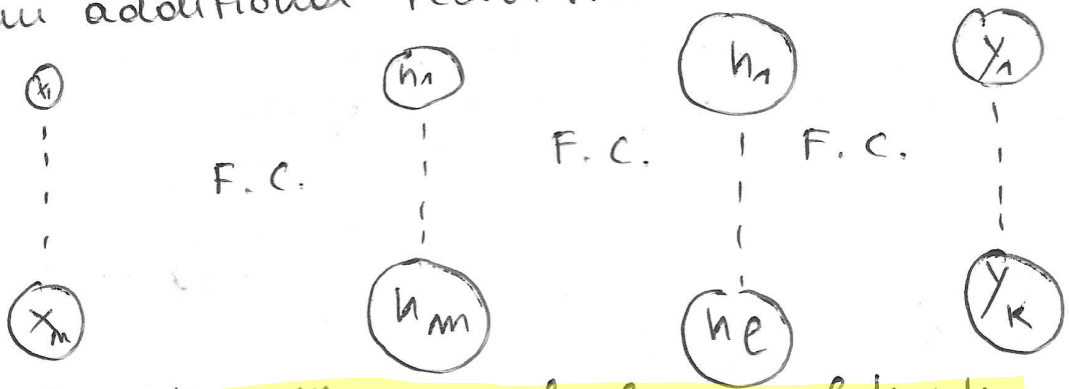
One more parameters!

If we use a fully connected network, in general the number of connection is given by the product of units of the previous and the current level. In general if we have a network with multiple layers!



$$M \cdot m + m \cdot l + l \cdot K = \text{number of connections}$$

This is also the number of parameters of the W matrix, we have $M \cdot m$, then $m \cdot l$ and so on. If you consider the bias terms you have to add an additional vector with the same size of the output



PARAMETERS: $M \cdot m + m$ $m \cdot l + l$ $l \cdot K + K$

The numbers of parameters in NN are very high! We cannot apply standard optimization methods to find a way to minimize the cost function. WE USE AN ITERATIVE METHOD CALLED STOCHASTIC GRADIENT DESCENT.

Computing the gradient is difficult since we have to derive the error wrt to the parameters. WE USE THE BACKPROP ALGORITHM.

Computing the gradient IS DIFFICULT vs. Apply stochastic gradient descent IS EASY.

→ back-propagate algorithm

To train the network we need to compute the gradients with respect to the network parameters θ. The back-propagation is used to propagate gradient computation from the cost through the whole network.

BACK-PROPAGATION IS SIMPLE and EFFICIENT:

- only used to compute the gradients
- IS NOT a training algorithm
- IS NOT specific to FNNs

The backpropagation works in two phases:

I
- I.(1) We give to the network the input and make an estimation having an output
- I.(2) compares the desiderated input (the one in the dataset) with the prediction.

THIS FIRST PHASE IS CALLED FORWARD PHASE or FORWARD STEP.

Then we have the backword step:

II The error is back propagated from the output to the input, to make an estimation of the gradient.

Intuition (Not going into details): use function separation, the chains rule, in computing the derivative:

$$z = f(g(x)) = f(y) \qquad y = g(x)$$

$$\boxed{\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}}$$

If we have vectors, we have PARTIAL derivative w.r.t each component.

We can express this by producing a matrix in which component of the matrix is the partial derivative of one component of the output w.r.t to one component of the input (JACOBIAN MATRIX).

$\frac{\partial \vec{y}}{\partial \vec{x}}$ the $m \times m$ Jacobian matrix.

Let's suppose that our error is just the absolute distance between the prediction and the target value: $|y - t|$. In order to compute the gradient, I have to back propagate this error to all the layers, BECAUSE THE ERROR IS DETERMINED BY SOME WEIGHT THAT IS WRONG. I have to distribute the error into the network.

The idea is to compute the gradient iteratively and refine it step by step, by starting from the last layer moving back to the initial

In each iteration you update the gradient with some terms that are computed (a kind of weighted back propag.)

WEIGHTED BACK PROP => the distribution of the error depends on the weight. When a connection has more absolute weight it will receive more error.

Once we have the gradient, how we change the parameters?

STOCHASTIC GRADIENT DESCENT (how to move on gradient)

In general we have the dataset and you can decide to:

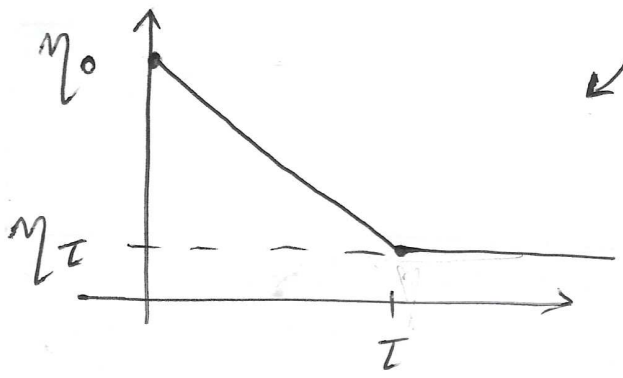
- compute all samples in the dataset (X_i, t_i) (BATCH)
- only one input x and one target value t. (INCREMENTAL)

Best approach is to use a subset of samples, called MINI-BATCH, from the dataset. Anytime I make a step of gradient descent I choose a different subset of samples.

GRADIENT IS ESTIMATED by mini-batch randomly chosen in each iteration in a different way. Then we move into the direction of the gradient by a learning factor η and we repeat it many times.

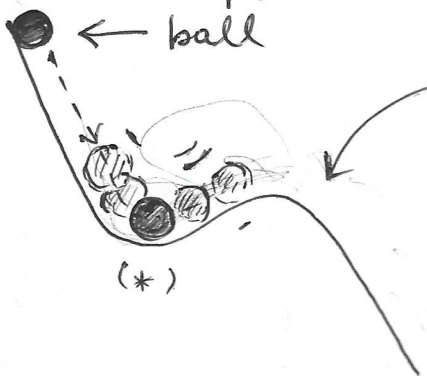
PROBLEM OF FIXING LEARNING RATE, the algorithm converges only if η is small enough. THIS ALGORITHM converges only to a LOCAL MINIMUM!

There is one approach usually used: LEARNING RATE VARIABLE OVERTIME, because in most of the cases you assume to be far away from the solution and when you are far away is convenient to have a large η. One way of having η changing over time is by decreasing it by using a function:



THE VALUE DECREASE BUT IT WILL BE CONSTANT AFTER REACHING T ITERATIONS (additional hyperparameters!)

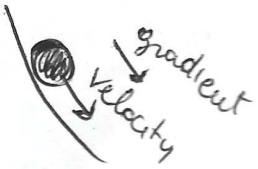
Another approach to reach convergence! ADDING MOMENTUM.

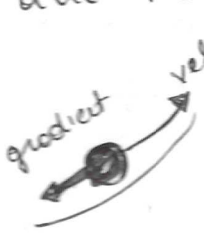


In some situation the ball can overcome this hill, this is THE MOMENTUM and this is what we want in SGD.

We can COMPUTE THE VELOCITY, the velocity of the movement of the solution in the algorithm.

We have a contribution of gradient and velocity





We have another parameter, THE MOMENTUM μ THAT IS ABLE TO TELL HOW MUCH THE VELOCITY gained WILL BE STRONG wrt the GRADIENT.

Another hyperparam. useful to overcome local minima...

Momentum μ might also increase according to some rule though the iterations.

- apply μ before the computation of the gradient (NESTEROV APPROACH)
- apply μ after the gradient.

In theory this model converges faster, but in practice is not always true. Based on analysis of the gradient of the loss function we have understand that:

> WE CAN VARY PARAMETERS (such as η) OVERTIME.

It is important to define adaptive algorithms. These hyperparameters are not fixed, but it is adaptive w.r.t to the result that the algorithm obtains during the training. Examples:

- AdaGrad
- RMSProp
- Adam

} We don't know which is the best, it is application dependent. ADAM is one of the MOST USED, since also the momentum is adaptive, not only η.

Regularization ⟶ orthogonal aspect of optimization. We still have the problem of overfitting.

THE GOAL of OPTIMIZAT.: find the minimum of loss function, but it is computed with the dataset, so it can't be good for new instances.

REGULARIZATION IS AN IMPORTANT FEATURE TO REDUCE OVERFITTING.

- Norm penalties: (exactly of what we have seen in SVM) Add in the error function term that penalizes large absolute value of the parameters. The idea is to reduce the amplitude of parameters.
-
- Dataset augmentation: overfitting may be due to small dataset, one solution is to add data. Sometimes you cannot gather new data, but you produce some data by make some processing.

In Image Classification you add processing of images to generate new images that are variations of the one in the dataset (rotate, illumination changing, ...)

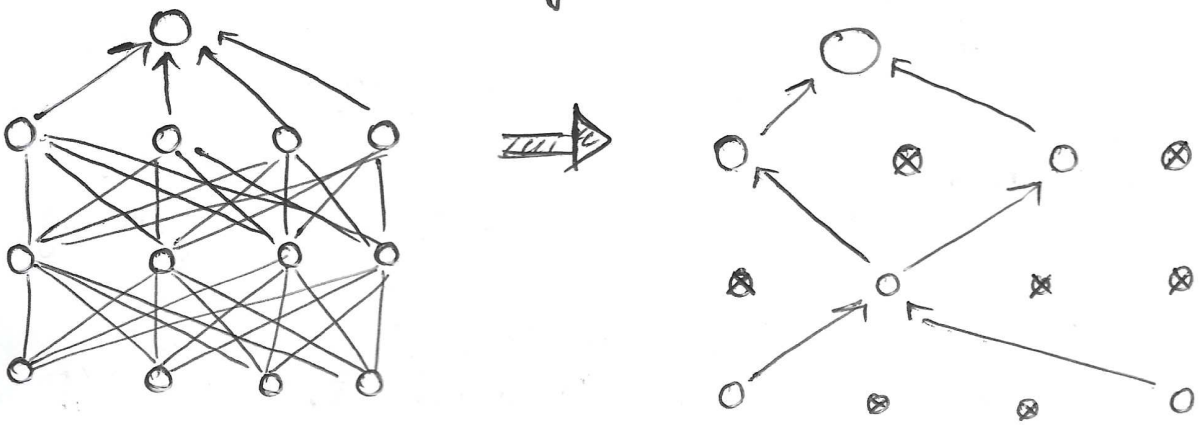
- **Early stopping** : one reason of NN's overfitting depends on how many iteration you do. OVERFITTING HAPPENS when you iterate too much. You have to use CROSS-VALIDATION.

  ↓

  Stop iteration when the error start increasing.

- **Parameter sharing** : we constraint some parameter to be the same. This is used in Convolutional Neural Network.

- **Dropout** : you randomly remove network units with probability α. Works quite well.



Note : the methods for regularization are not mutual exclusive.

Very often to decide one choiche we need to experience, there is not a theoretical support in most of the cases.