Neural networks for structured data

Table of contents

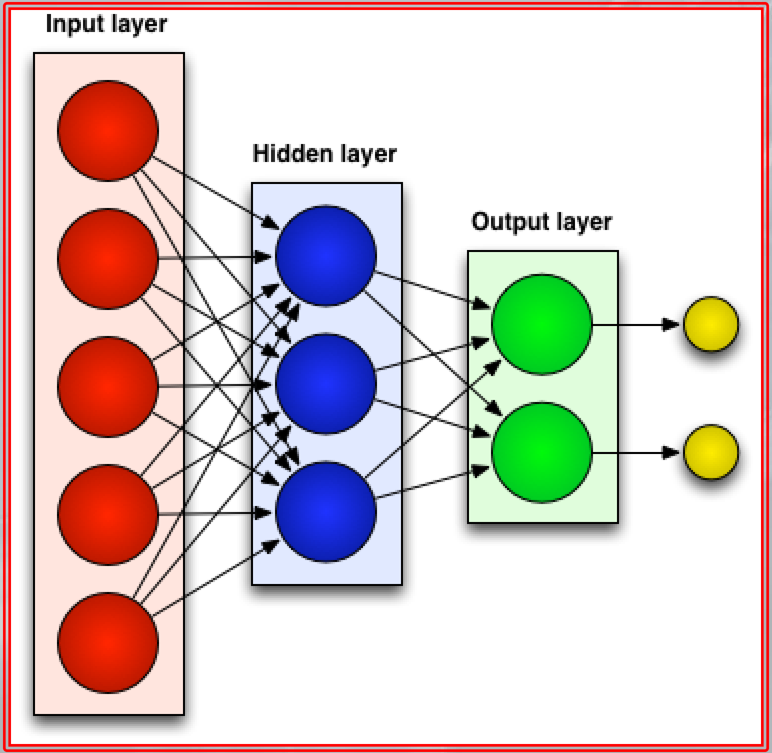
- Recurrent models
  - Partially recurrent neural networks
    - Elman networks
    - Jordan networks
  - Recurrent neural networks
  - BackPropagation Through Time
  - Dynamics of a neuron with feedback
  - Long–Short Term Memories
- Recursive models
  - Recursive architectures
    - Learning environment
  - Linear recursive networks
  - Recursive networks for directed, cyclic graphs
- Graph neural networks

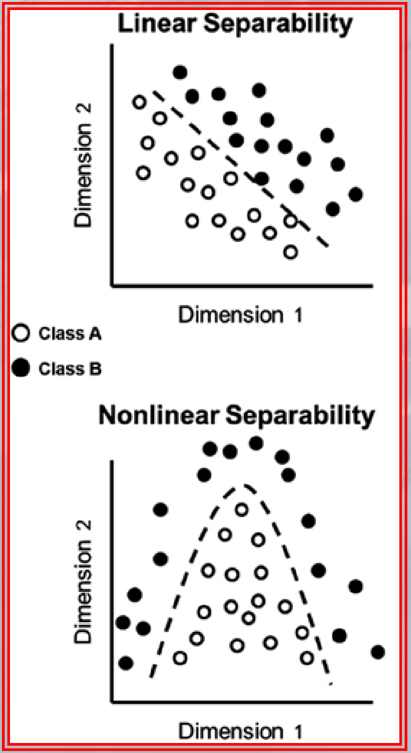
Introduction – 1

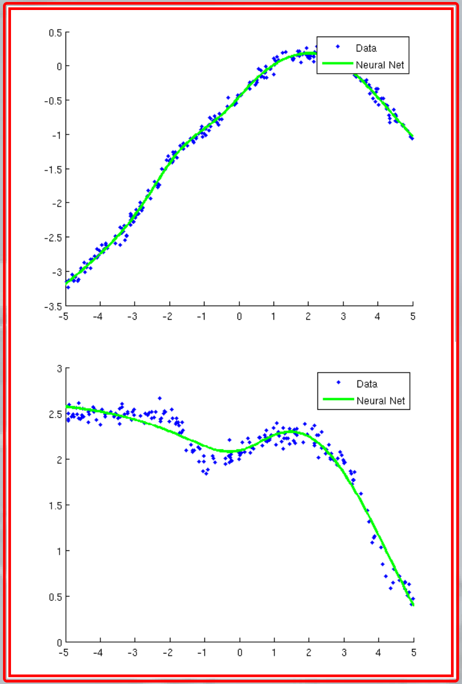
- Network architectures able to process “plain” data, collected within arrays, are said to be static; they just define a “mapping” between the sets of input and output values
- In other words… once the network has been trained, it realizes a function between inputs and outputs, calculated according to the learning set
- The output at time t depends only on the input at the same time
    - The network does not have “short–term” memory

Introduction – 2

- Static networks: Feedforward neural networks
  - They learn a static I/O mapping, $\boldsymbol{Y}=\boldsymbol{f}(\boldsymbol{X})$, $\boldsymbol{X}$ and $\boldsymbol{Y}$ static patterns (arrays)
  - They model static systems: classification or regression tasks







Introduction – 2

- Static networks
  - For classification problems:

    $\boldsymbol{f} : \mathbb{R}^n \rightarrow \{0,1\}^m$

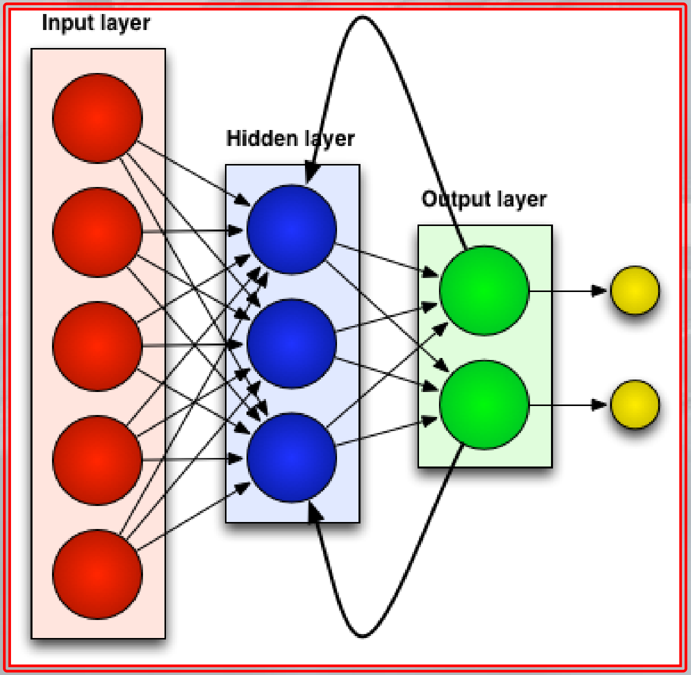
    where n is the dimension of the input vector and m is the number of different classes that patterns belong to (sigmoid output neurons)
  - For regression problems

    $\boldsymbol{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$

    where n is the dimension of the input vector and m is the dimension of the target (linear output neurons)

Introduction – 4

- Dynamic networks: Recurrent neural networks
  - They learn a non–stationary I/O mapping, $Y(t)=f(t,X(t))$, $X(t)$ and $Y(t)$ are time–varying patterns
  - They model dynamic systems: control systems, optimization problems, artificial vision and speech recognition tasks, time series prediction



Dynamic networks

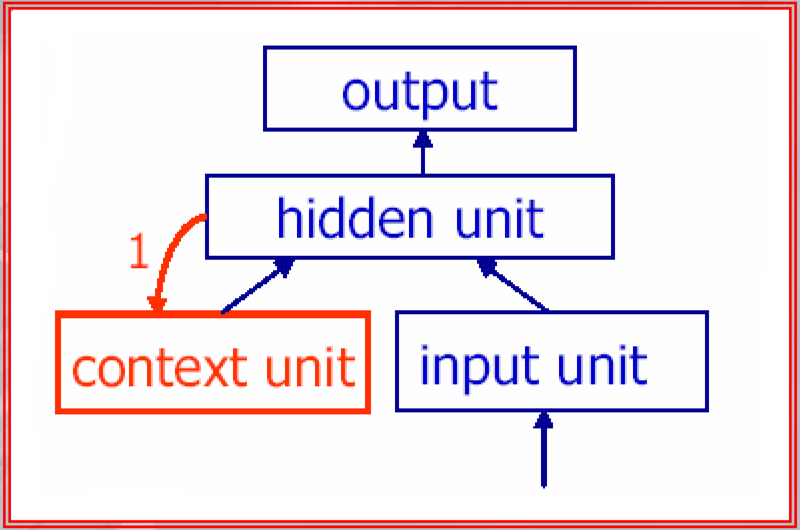
- Equipped with a temporal dynamics, these networks are able to capture the temporal structure of the input and to “produce” a timeline output
  - Temporal dynamics: unit activations can change in time even in presence of the same input pattern
- Architectures composed by units having feedback connections, both between neurons belonging to the same layer or to different layers
  - Partially recurrent networks
  - Recurrent networks

Partially recurrent networks

- Feedforward networks endowed with a set of input units, called *state* or *context units*
- The context layer output corresponds to the output, at the previous time step, of the units that emit feedback signals, and it is sent to the units receiving feedback signals
  - Elman networks (1990)
  - Jordan networks (1986)

Elman networks – 1

- Feedback connections on the hidden layer, with fixed weights all equal to one
- Context units equal, in number, to the hidden units, and considered just as input units

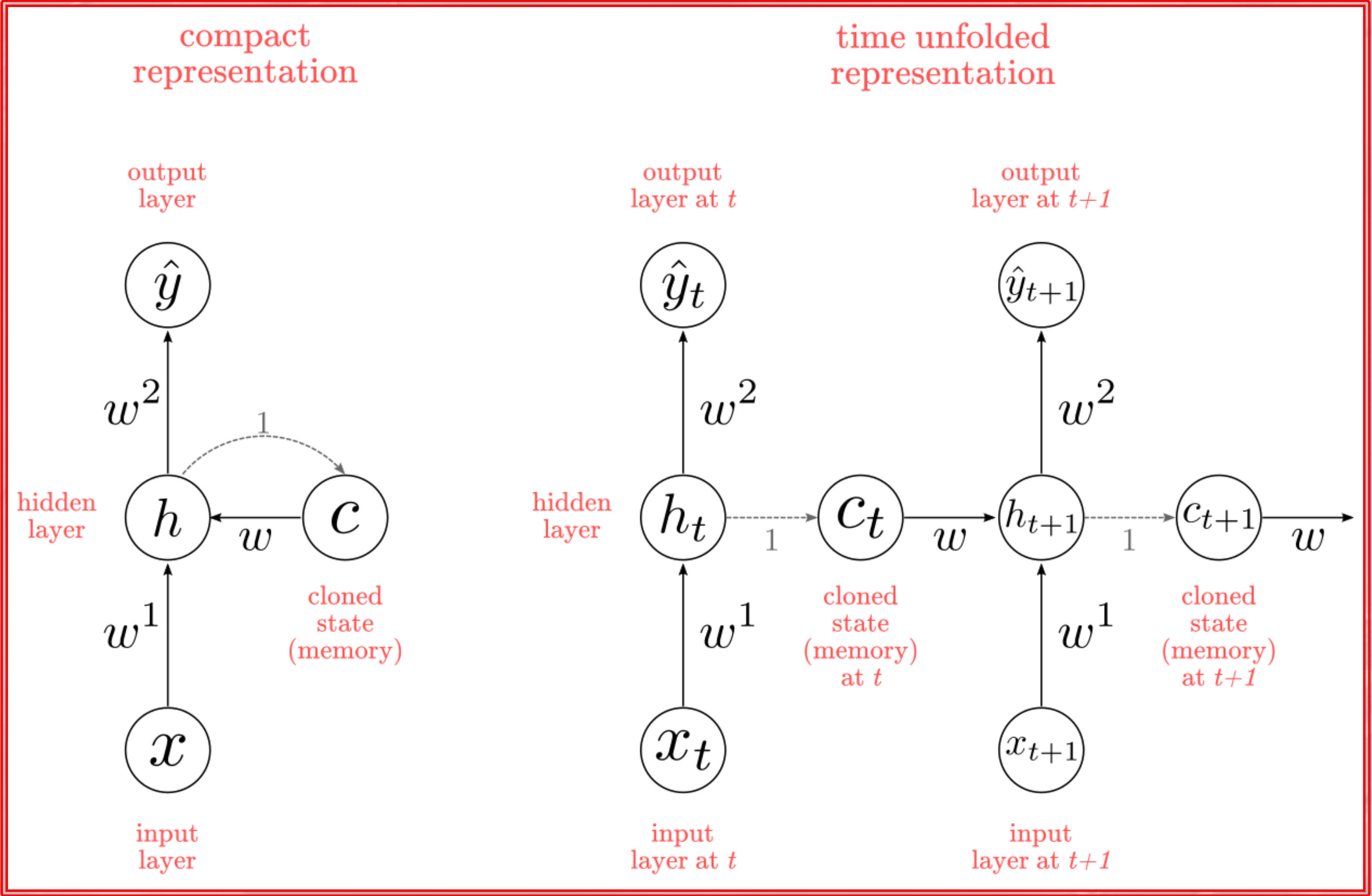


- The output of each context unit is equal to that of the corresponding hidden unit at the previous (discrete) instant:

$$x_{\mathrm{c},i}(t) = x_{\mathrm{h},i}(t-1)$$

- To train the network, the Backpropagation algorithm is used, in order to learn the hidden–output, the input–hidden and the context–hidden weights

Elman networks – 2



Elman networks – 3

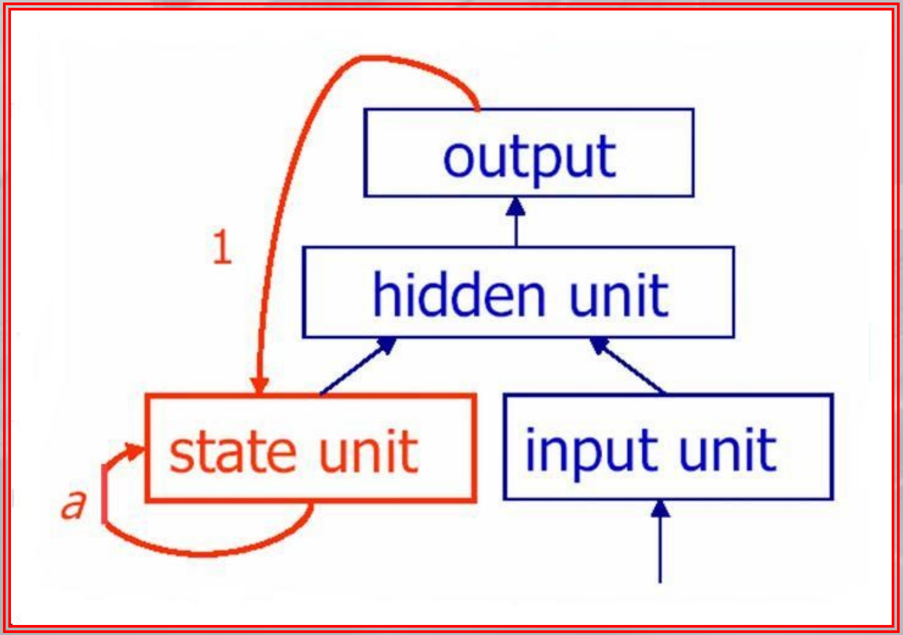
- All the output functions operate on the weighed sum of the inputs, except for the input and the context layers, that act just as "buffers"
  - Actually, sigmoidal functions are used in both the hidden and the output layer
- The context layer inserts a single–step delay in the feedback loop: the output of the context layer is presented to the hidden layer, in addition to the current pattern
  - The context layer adds, to the current input, a value that reproduces the output achieved at the hidden layer based on all the patterns presented up to the previous step

Elman networks – 4

- Learning – all the trainable weights are attached to forward connections
  1) The activation of the context units is initially set to zero, i.e. $x_{c,i}(0)=0$, $\forall i$ at $t=0$
  2) Input pattern $\boldsymbol{x}_t$: evaluation of the activations/outputs of all the neurons, based on the feedforward transmission of the signal along the network
  3) Weight updating using Backpropagation (on–line)
  4) Let $t = t+1$, $\boldsymbol{x}_c(t) = \boldsymbol{x}_h(t-1)$ and go to 2)
- The Elman network produces a finite sequence of out-puts, one for each input
  - The Elman network is normally used for object trajectory prediction, and for the generation/recognition of lin-guistic patterns

Jordan networks – 1

- Feedback connections on the output layer, with fixed weights all equal to one
- Self–feedback connections for the state neurons, with constant weights equal to a; $a < 1$ is the *recency constant*



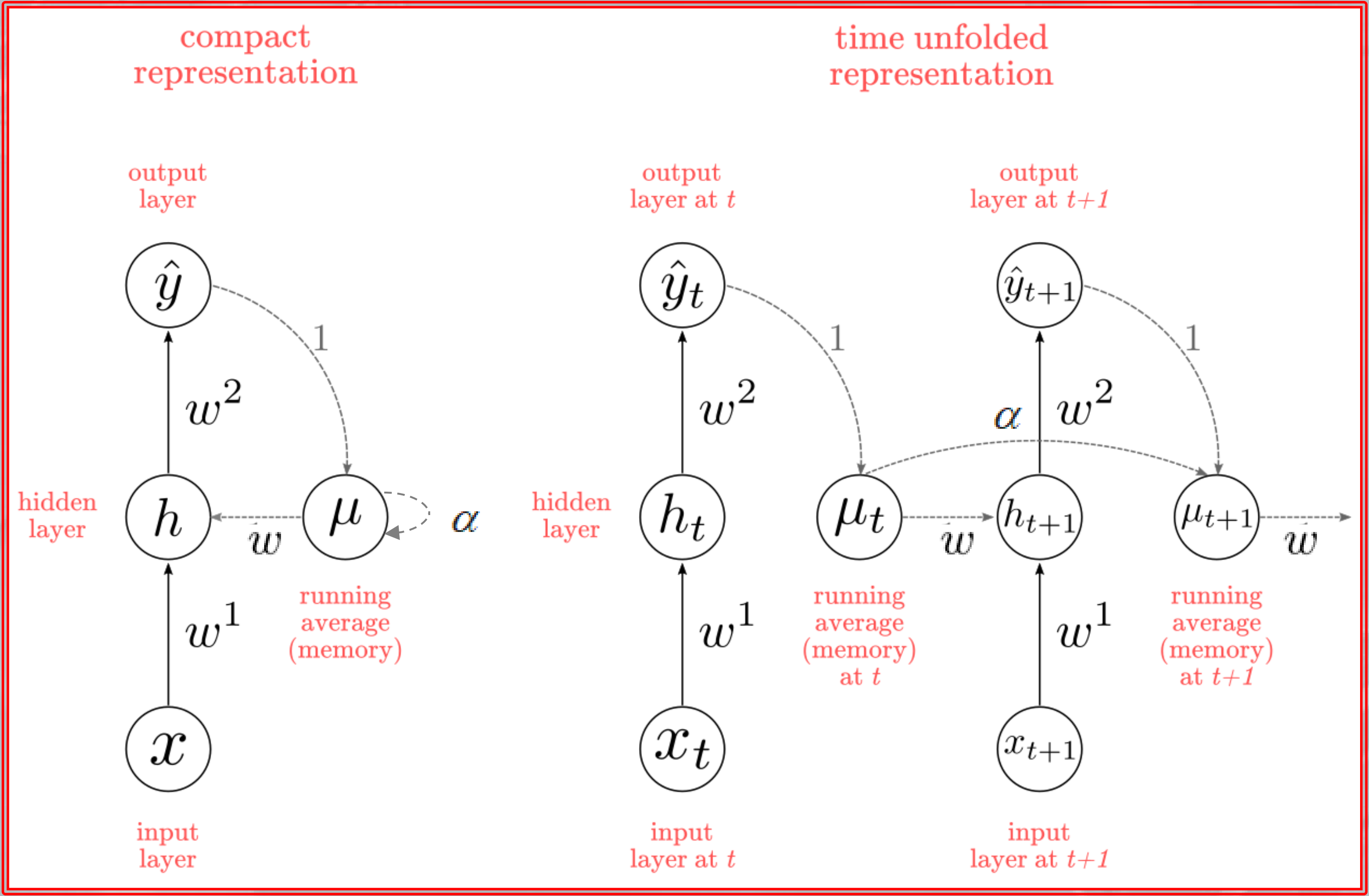
Jordan networks – 2

- The network output is sent to the hidden layer by using a context layer
- The activation, for the context units, is determined based on the activation of the same neurons and of the output neurons, both calculated at the previous time step

$$x_{\mathrm{c},i}(t) = x_{\mathrm{o},i}(t-1) + a\,x_{\mathrm{c},i}(t-1)$$

  - Self–connections allow the context units to develop a local or “individual” memory, which takes into account past information with a weight that decreases over time
  - To train the network, the Backpropagation algorithm is used, in order to learn the hidden–output, the input–hidden and the context–hidden weights

Jordan networks – 3



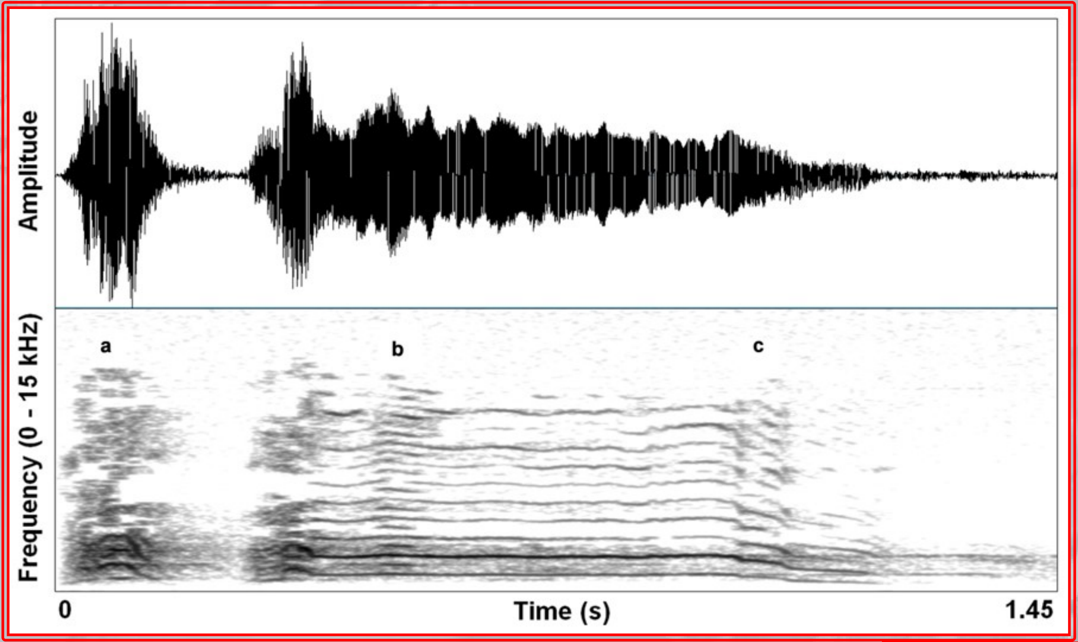
Jordan networks – 4

- The context layer inserts a delay step in the feedback loop: the context layer output is presented to the hidden layer, in addition to the current pattern
  - The context layer adds, to the input, a value that reproduces the output achieved by the network based on all the patterns presented up to the previous step, coupled with a fraction of the value calculated, also at the previous step, by the context layer itself (via self–connections)

Temporal data – 1

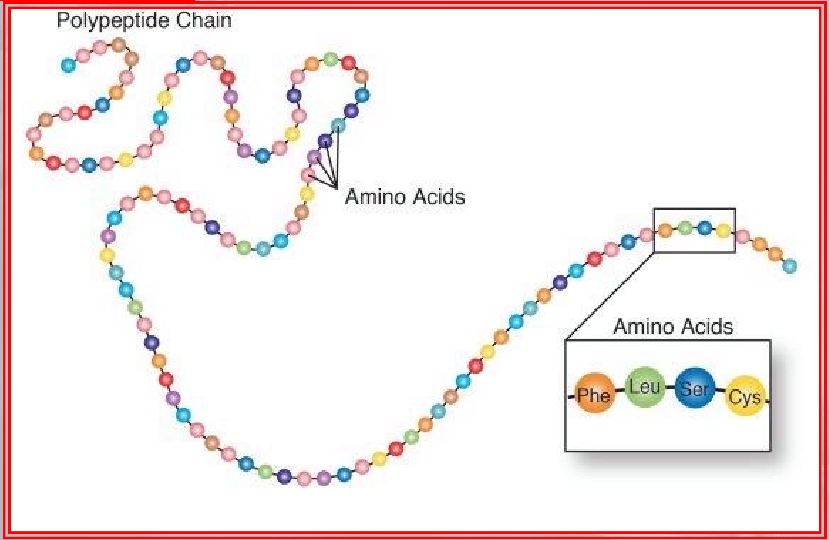
- The simplest *dynamic* data type is the *sequence*, which is a natural way to model temporal/sequential domains
  - In *speech recognition*, the words, which are the object of the recognition problem, naturally flow to constitute a temporal sequence of acoustic features
  - In molecular biology, proteins are organized in amino acid strings
  - Stock market, video analysis, weather forecasts,…
- The simplest dynamic architectures are recurrent networks, able to model temporal/sequential phenomena

Temporal data– 2

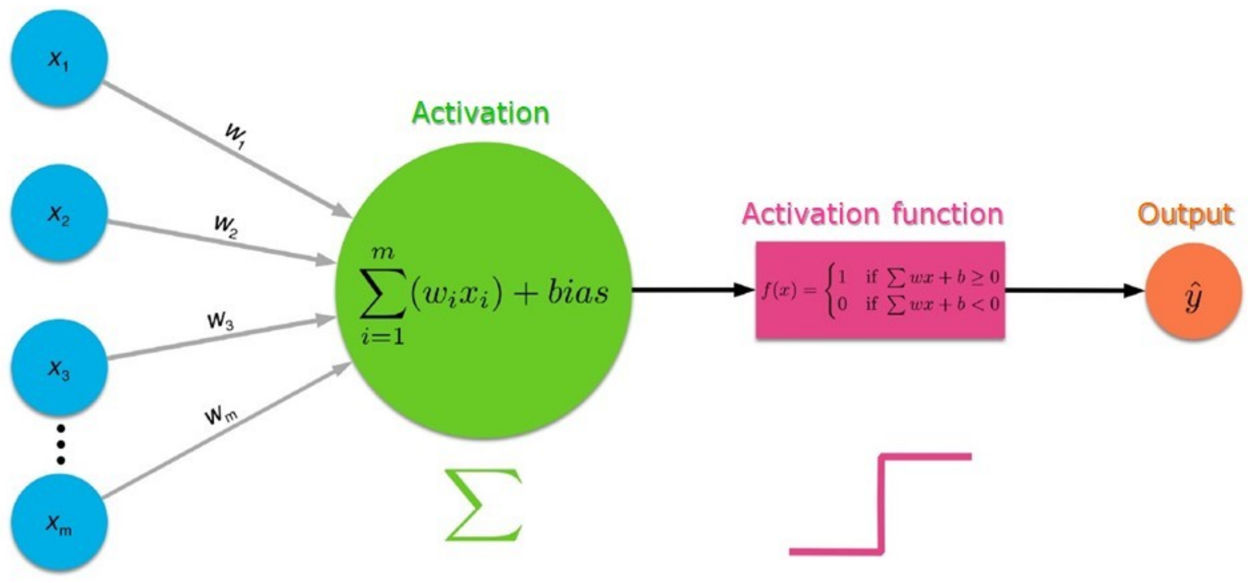


Temporal data describing acoustic features

Sequential data describing the primary structure of a protein



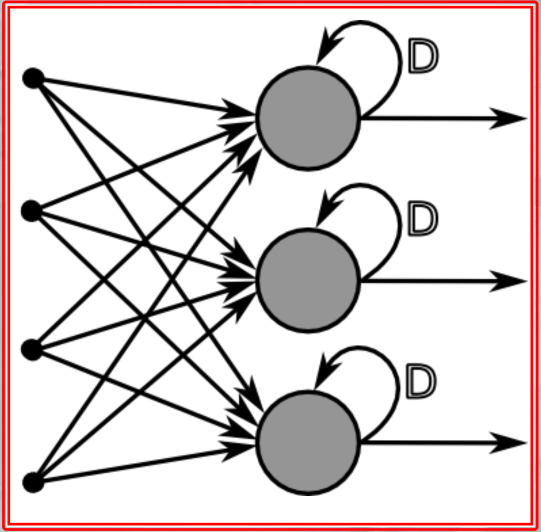
Recurrent networks – 1

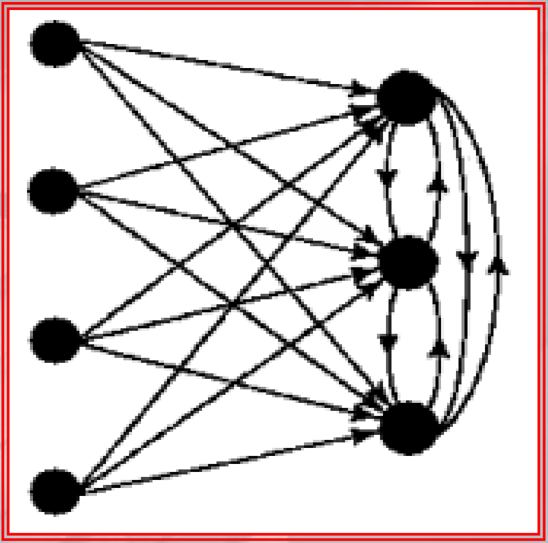


x_1
x_2
x_3
x_m
w_1
w_2
w_3
w_m
Activation
$\sum_{i=1}^{m}(w_i x_i) + bias$
Σ
Activation function
$f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases}$
Output
$\hat{y}$

Recurrent networks – 2

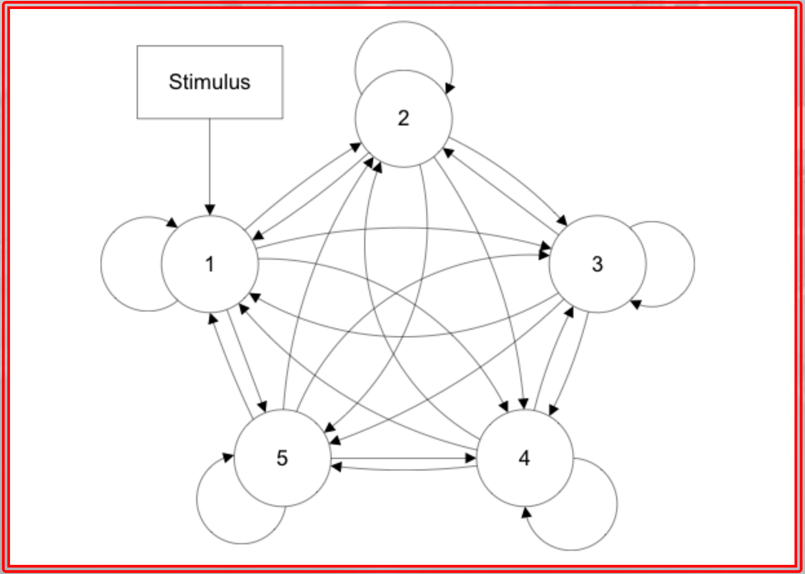
RNN with self feedbacks





RNN with lateral feedbacks

Fully connected RNN



Recurrent networks – 3

- A recurrent network processes a temporal sequence by using an internal state representation, that appropriately encodes all the past information injected into its inputs
  - memory arises from the presence of feedback loops between the output of some neurons and the input of other neurons
  - assuming a synchronous update mechanism, the feedback connections have a memory element (a one–step delay)
- The inputs are sequences of arrays:

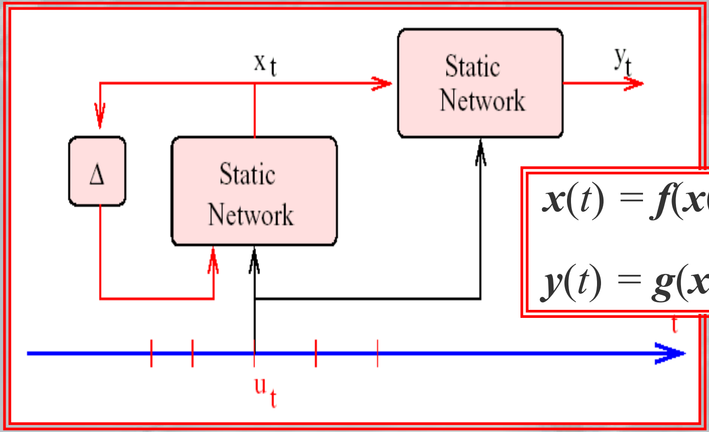
$$\mathcal{S}_p = \mathbf{x}_0^p(1), \mathbf{x}_0^p(2), \ldots, \mathbf{x}_0^p(T_p)\ , \quad \mathbf{x}_0^p(t) \in R^m, \quad t = 1, \ldots, T_p$$

where T_p represents the length of the p–th sequence (in general, sequences of finite length are considered, even if this is not a necessary requirement)

Recurrent networks – 4

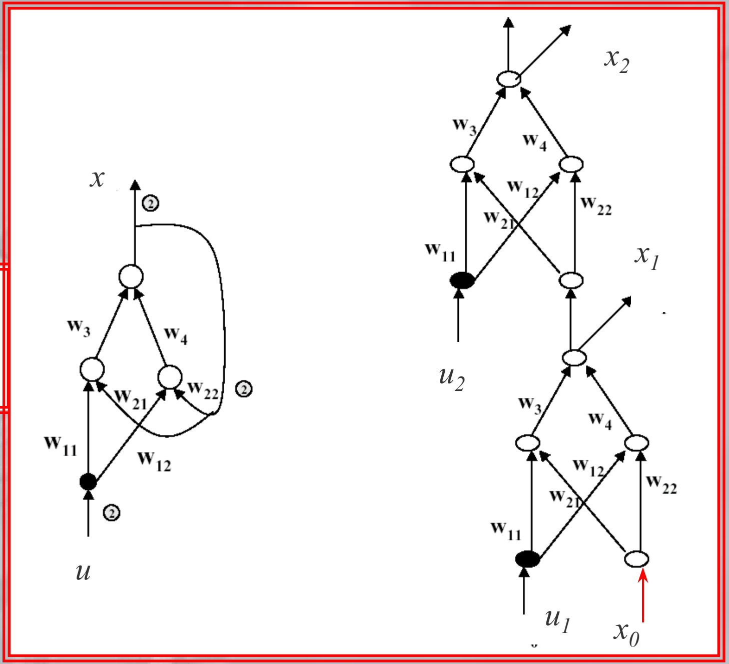
- Commonly using an MLP as the basic block, multiple types of recurrent networks may be defined, depending on which neurons are involved in the feedback
  - The feedback may be established from the output to the hidden neurons
  - The feedback may involve the output of the hidden layer neurons
  - In the case of multiple hidden layers, feedbacks can also be present on several layers
- Therefore, many different configurations are possible for a recurrent network
- Most common architectures exploit the ability of MLPs to implement non–linear functions, in order to realize networks with a non–linear dynamics

Recurrent networks – 5



$$\boldsymbol{x}(t) = \boldsymbol{f}(\boldsymbol{x}(t-1), \boldsymbol{u}(t))$$

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t))$$



- The behaviour of a recurrent network (during a time sequence) can be reproduced by *unfolding it in time*, and obtaining the corresponding feedforward network

Recurrent processing

- Before starting to process the $p\text{–}th$ sequence, the state of the network must be initialized to an assigned value (initial state) $\boldsymbol{x}^p(0)$
  - Every time the network begins to process a new sequence, there occurs a preliminary “reset” to the initial state, losing the memory of the past processing phases, that is, we assume to process each sequence independently from the others
- At each time step, the network calculates the current output of all the neurons, starting from the input $\boldsymbol{u}^p(t)$ and from the state $\boldsymbol{x}^p(t-1)$

Processing modes

- Let us suppose that the *L–th* layer represents the output layer
  - The neural network can be trained to transform the input sequence into an output sequence of the same length (realizing an *Input/Output transduction*)

$$\begin{tikzcd}
\mathbf{x}_0^p(1) \arrow[d] & \mathbf{x}_0^p(2) \arrow[d] & \ldots \arrow[d, phantom, "\ldots"] & \mathbf{x}_0^p(T_p-1) \arrow[d] & \mathbf{x}_0^p(T_p) \arrow[d] \\
\mathbf{x}_L^p(1) & \mathbf{x}_L^p(2) & \ldots & \mathbf{x}_L^p(T_p-1) & \mathbf{x}_L^p(T_p)
\end{tikzcd}$$

  - A different case is when we are interested only in the network response at the end of the sequence, so as to transform the sequence into a vector (*supersource transduction*)
    - This approach can be used to associate each sequence to a class in a set of predefined classes

$$\begin{tikzcd}
\mathbf{x}_0^p(1) & \mathbf{x}_0^p(2) & \ldots & \mathbf{x}_0^p(T_p-1) & \mathbf{x}_0^p(T_p) \arrow[d] \\
 & & & & \mathbf{x}_L^p(T_p)
\end{tikzcd}$$

Learning Set

- Let us consider a supervised learning scheme in which:
  - input patterns are represented by sequences

$$\mathcal{S}_p = \mathbf{x}_0^p(1), \mathbf{x}_0^p(2), \ldots, \mathbf{x}_0^p(T_p)\ , \quad \mathbf{x}_0^p(t) \in R^m, \quad t = 1, \ldots, T_p$$

  - target values are represented by subsequences

$$\mathcal{D}_p = \mathbf{d}^p(t_1), \mathbf{d}^p(t_2), \ldots, \mathbf{d}^p(t_{K_p})\ , \quad \mathbf{d}^p(t) \in R^{n(L)}, \quad t = t_1, \ldots, t_{K_p} \in [1, T_p]$$

  - Therefore, the supervised framework is supposed to provide a desired output possibly with respect to a subset of the processing time steps
    - In the case of sequence classification (or sequence coding into vectors) there will be a single target value, at time T_p

Cost function

- The learning set is composed by sequences, each associated with a target subsequence

$$L_e = \left\{ (\mathcal{S}_p, \mathcal{D}_p) | \mathcal{S}_p \in \mathcal{S}(R^m), \mathcal{D}_p \in \mathcal{S}(R^{n(L)} \cup \{\epsilon\}), p = 1, \ldots, N \right\}$$

where ϵ stands for empty positions, possibly contained in the target sequence

- The cost function, measuring the difference between the network output and the target sequence, for all the examples belonging to the learning set, is defined by

$$\mathcal{E}(L_e, \mathbf{W}) = \sum_{p=1}^{N} E_{\mathbf{W}}(\mathcal{S}_p, \mathcal{D}_p) = \sum_{p=1}^{N} \sum_{i=1}^{K_p} e^p_{\mathbf{W}}(t_i) = \sum_{p=1}^{N} \sum_{i=1}^{K_p} \frac{1}{2} ||\mathbf{x}^p_L(t_i) - \mathbf{d}^p(t_i)||^2$$

  where the instantaneous error $e^p_{\mathbf{W}}(t_i)$ is expressed as the Euclidean distance between the output vector and the target vector at time t_i (but, other distances may also be used)

Learning in recurrent networks

- *Backpropagation Through Time* (BPTT, Rumelhart, Hinton, Williams, 1986)
  - The temporal dynamics of the recurrent network is "converted" into that of the corresponding unfolded feed-forward network
    - Advantage: very simple to calculate
    - Disadvantage: heavy memory requirements
- *Real–Time Recurrent Learning* (Williams, Zipser, 1989)
  - Recursive calculation of the gradient of the cost function associated with the network
    - The derivatives of states and outputs with respect to all weights are computed as the network processes the sequence, that is, during the forward step
    - No unfolding is performed or necessary
    - Disadvantage: computationally expensive

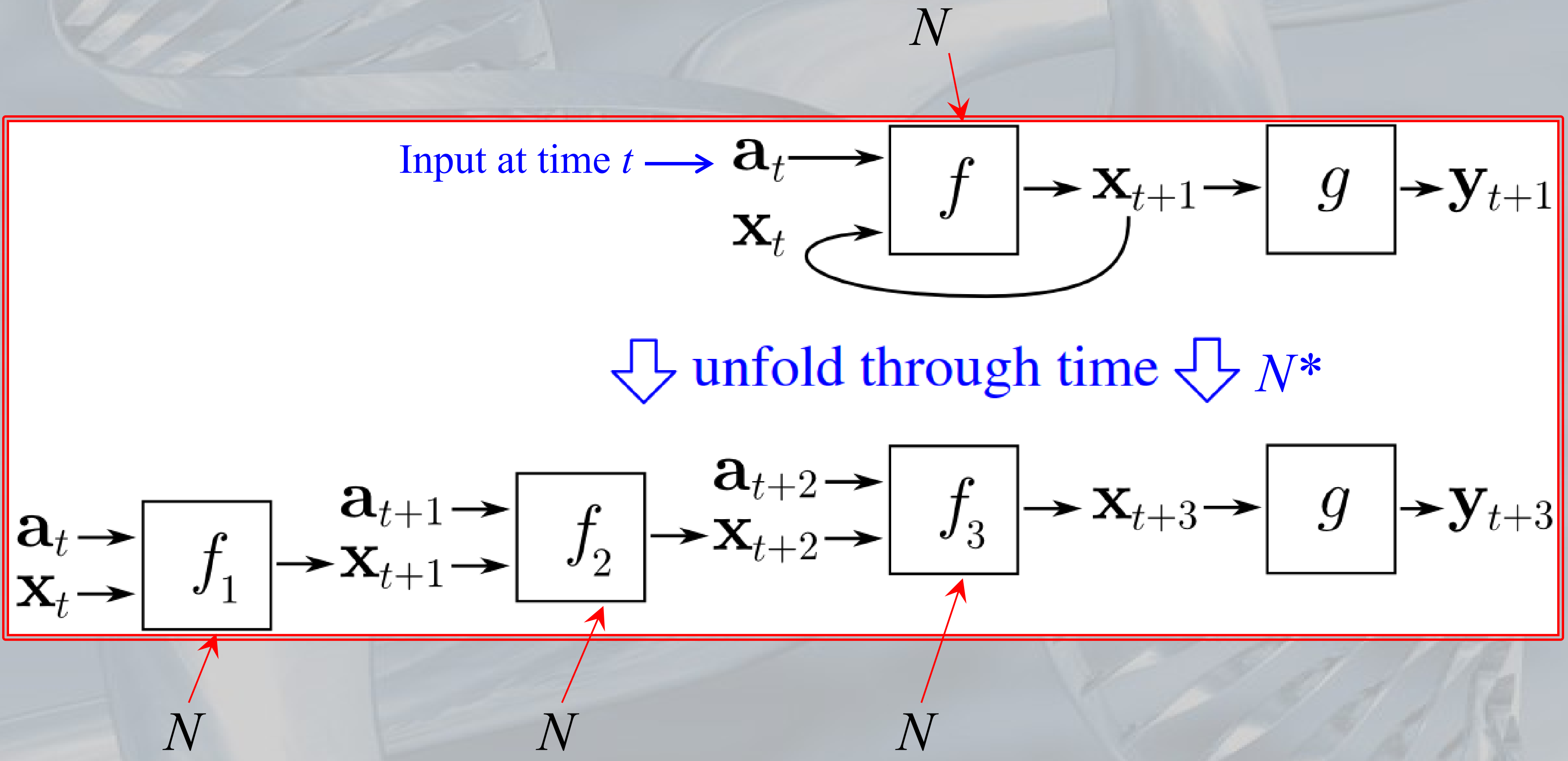
BackPropagation Through Time – 1

- Given the targets to be produced, the network can be trained using BackPropagation Through Time (BPTT)
- Using BPTT means…
  - …considering the corresponding feedforward network unfolded in time ⇨ the length T_p of the sequence to be learnt must be known
  - …updating all the weights $w_i(t)$, $t=1,\ldots,T_p$, in the feedforward network, which are copies of the same w_i in the recurrent network, by the same amount, corresponding to the sum of the various updates reported in different layers ⇨ all the copies of $w_i(t)$ should be maintained equal

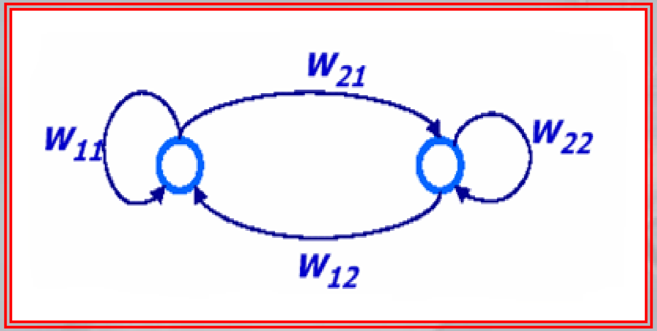
BackPropagation Through Time – 2

- Let N be a recurrent network that must be trained, starting from 1, on a sequence of length T_p
- On the other hand, let N^* be the feedforward network obtained by unfolding N in time
- With respect to N^* and N, the following statements hold:
  - N^* has a "hyperlayer" that contains a copy of N, corresponding to each time step
  - Each hyperlayer in N^* collects a copy of all the neurons contained in N
  - For each time step $t \in [1, T_p]$, the synapse from neuron i in layer l to neuron j in layer $l+1$ in N^* is just a copy of the same synapse in N

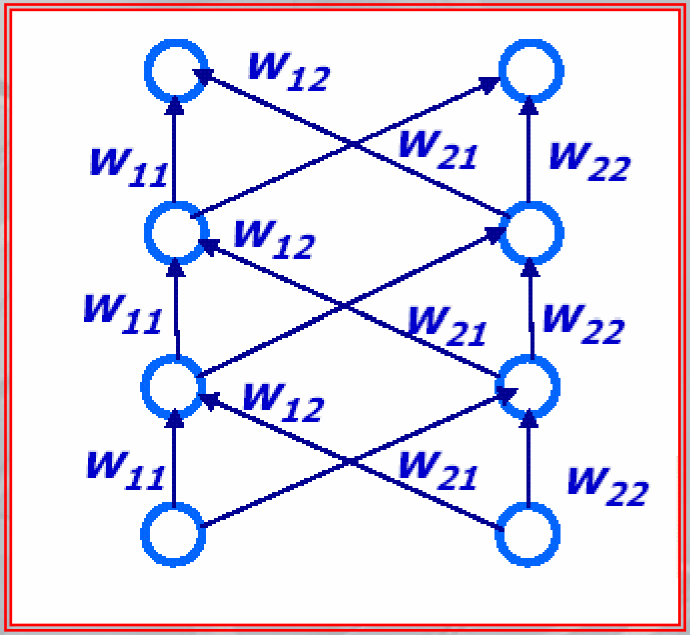
BackPropagation Through Time – 3



BackPropagation Through Time – 4



Recurrent network



Feedforward network corresponding to a sequence of length T=4

BackPropagation Through Time – 5

- The gradient calculation may be carried out in a BP–like style
  - The algorithm can be derived from the observation that recurrent processing in time is equivalent to constructing the corresponding unfolded feedforward network
  - The unfolded network is a multilayer network, on which the gradient calculation can be realized via standard BackPropagation
  - The constraint that each replica of the recurrent network within the unfolding network must share the same set of weights has to be taken into account (this constraint simply imposes to accumulate the gradient related to each weight with respect to each replica during the network unfolding process)

BackPropagation Through Time – 6

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \delta_j x_i$$

$$\delta_j = \frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial \text{net}_j} = \begin{cases} (x_j - t_j) x_j (1 - x_j) & \text{if } j \text{ is an output neuron} \\ (\sum_{\ell \in L} w_{j\ell} \delta_\ell) x_j (1 - x_j) & \text{if } j \text{ is an inner neuron} \end{cases}$$

- In other words…
  - We can think of the recurrent network as a feed-forward architecture with shared weights and then train this unfolded network with weight constraints
  - The training algorithm works in the *time domain*:
    - The forward pass builds up a stack of the outputs of all the neurons at each time step
    - The backward pass pulls outputs off the stack to compute the error derivatives at each time step
    - After that, the derivatives at all the different times, for each weight, are added together

BackPropagation Through Time – 7

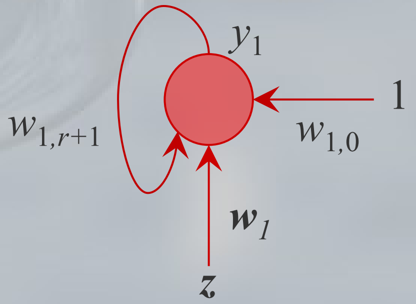
- The meaning of *backpropagation through time* is highlighted by the idea of network unfolding
- The algorithm is *non–local in time* – the whole sequence must be processed, by storing all the neuron outputs at each time step – but it is *local in space*, since it uses only local variables to each neuron
- It can be implemented in a modular fashion, based on simple modifications to the Back-Propagation procedure, normally applied to static MLP networks

Dynamics of a neuron with feedback – 1

- Let us consider a network constituted by only one neuron, equipped with a self–feedback; then:

$$y_1 = f(a_1) \quad a_1 = \sum_{i=1}^{r} w_{1,i} z_i + w_{1,r+1} y_1 + w_{1,0} \Rightarrow y_1 = a_1 \lambda + \mu$$

with: $\lambda = \dfrac{1}{w_{1,r+1}} \quad \mu = -\dfrac{1}{w_{1,r+1}} \left(\sum_{i=1}^{r} w_{1,i} z_i + w_{1,0} \right)$

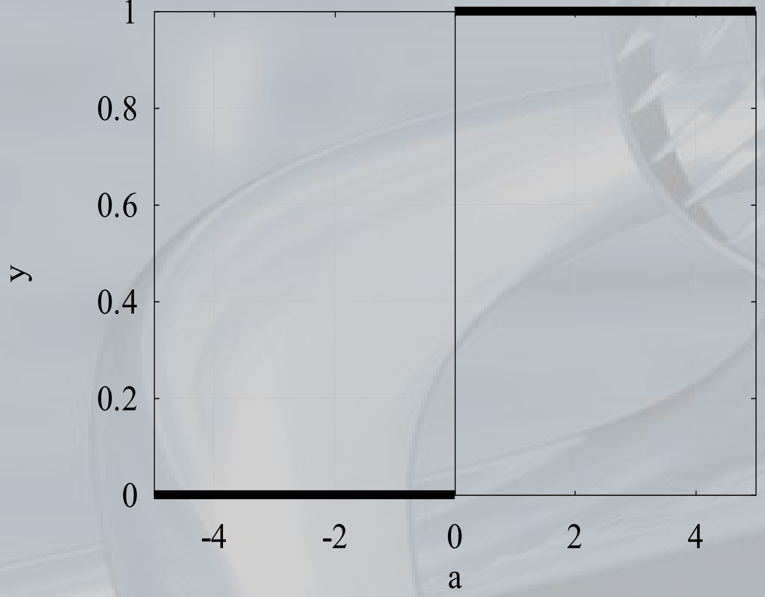


- Since a_1 and y_1 are functions of t, the neuron dynamics converges when they stop changing time after time
- Depending on synaptic weigths and inputs, functions $f(a_1)$ and $a_1\lambda+\mu$ may intersect in one or more points – or may have no intersections (i.e., solutions) –, that are equilibrium points of the dynamic system
- If a solution exists ⇨ *information latching*

Dynamics of a neuron with feedback – 2

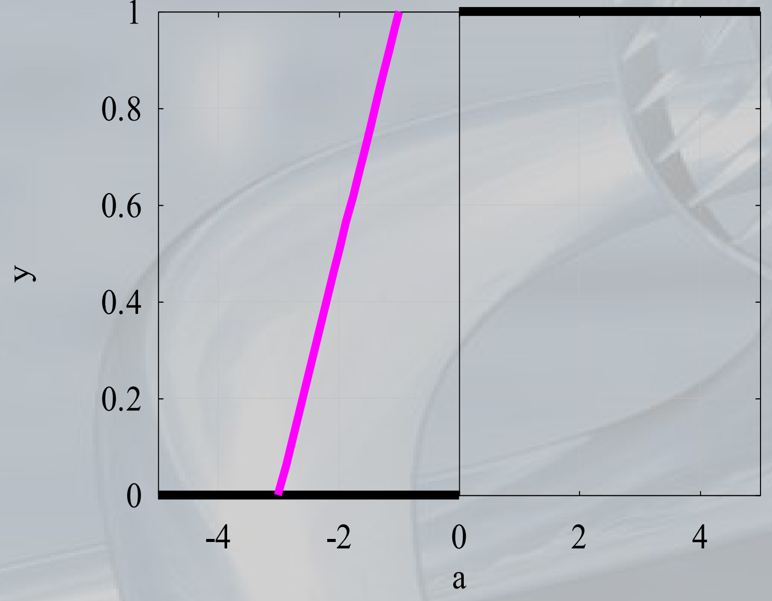
- When using a step transfer function, there can be two solutions at most

Step transfer function



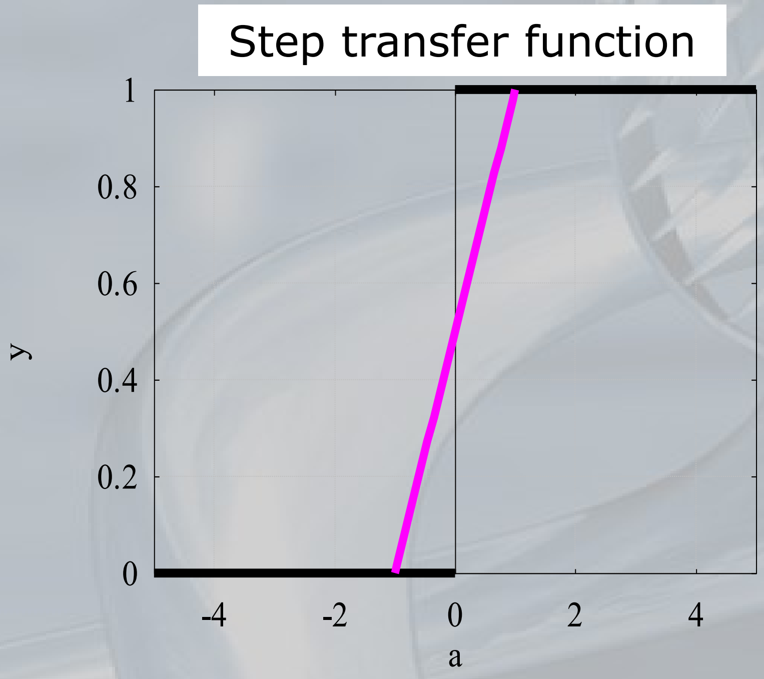
Dynamics of a neuron with feedback – 2

Step transfer function



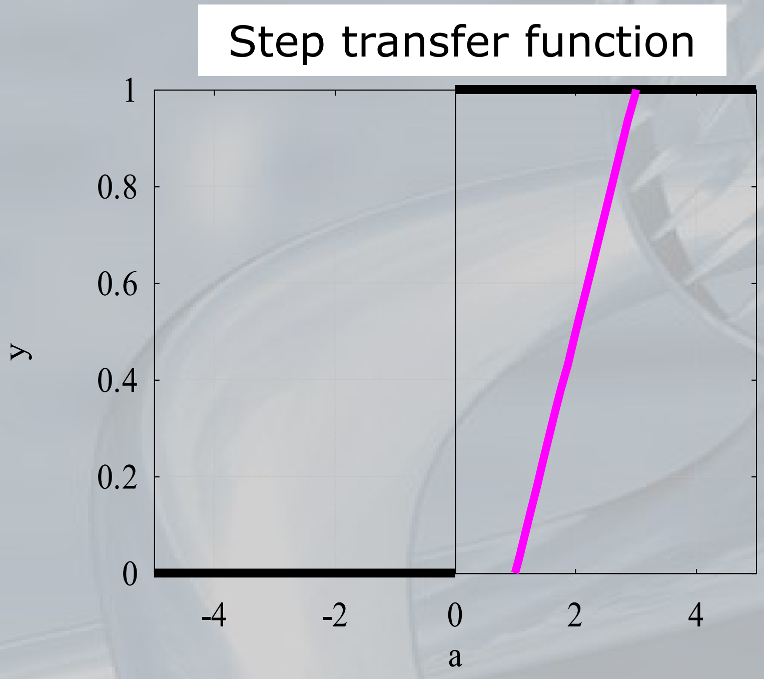
- When using a step transfer function, there can be two solutions at most
- λ=0.5, μ=1.5 (1 solution)

Dynamics of a neuron with feedback – 2



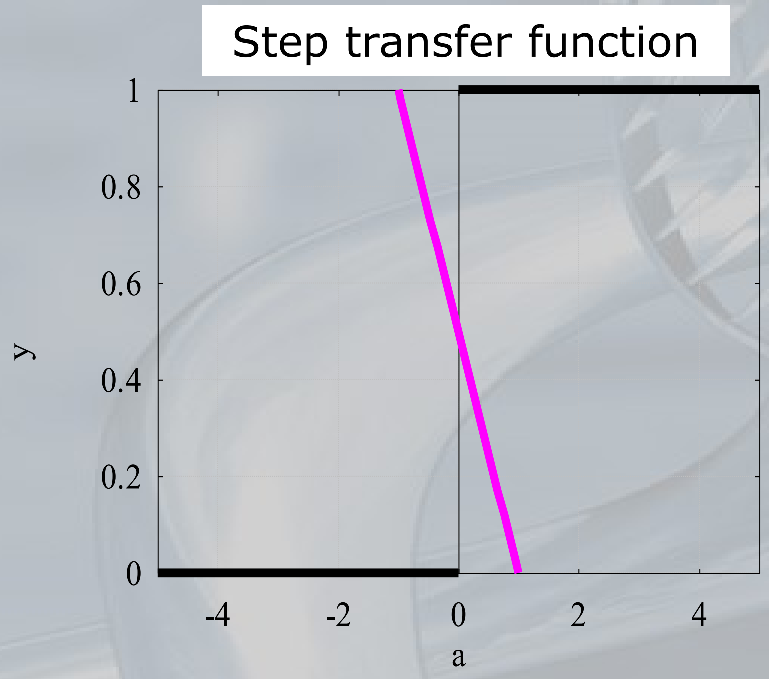
- When using a step transfer function, there can be two solutions at most
- λ=0.5, μ=1.5 (1 solution)
- λ=0.5, μ=0.5 (2 solutions)

Dynamics of a neuron with feedback – 2



- When using a step transfer function, there can be two solutions at most
- $\lambda=0.5$, $\mu=1.5$ (1 solution)
- $\lambda=0.5$, $\mu=0.5$ (2 solutions)
- $\lambda=0.5$, $\mu=-0.5$ (1 solution)

Dynamics of a neuron with feedback – 2

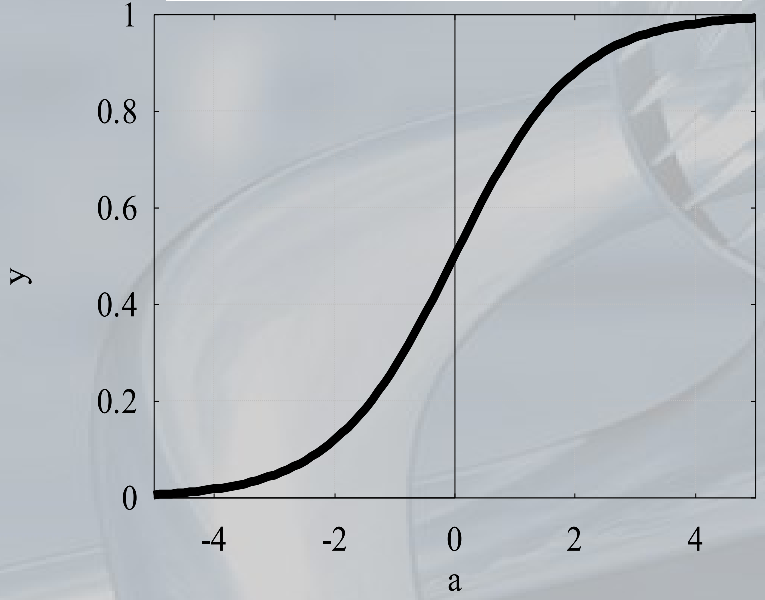


- When using a step transfer function, there can be two solutions at most
- $\lambda=0.5$, $\mu=1.5$ (1 solution)
- $\lambda=0.5$, $\mu=0.5$ (2 solutions)
- $\lambda=0.5$, $\mu=-0.5$ (1 solution)
- $\lambda=-0.5$, $\mu=0.5$ (0 solutions)

Dynamics of a neuron with feedback – 3

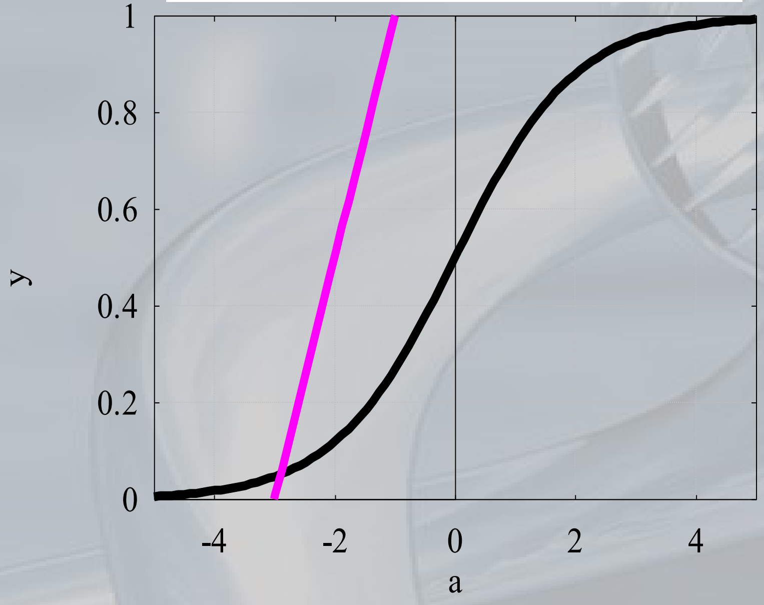
- In the case of continuous transfer functions, at least one solution always exists
- It can be shown that the same property holds true for any recurrent network

Logistic transfer function



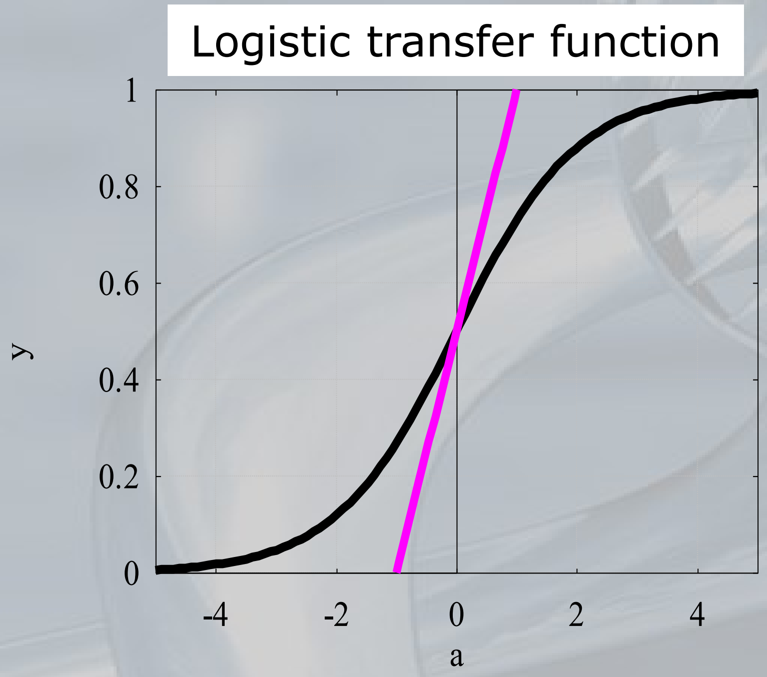
Dynamics of a neuron with feedback – 3

Logistic transfer function



- In the case of continuous transfer functions, at least one solution always exists
- It can be shown that the same property holds true for any recurrent network
- $\lambda=0.5$, $\mu=1.5$ (1 solution)

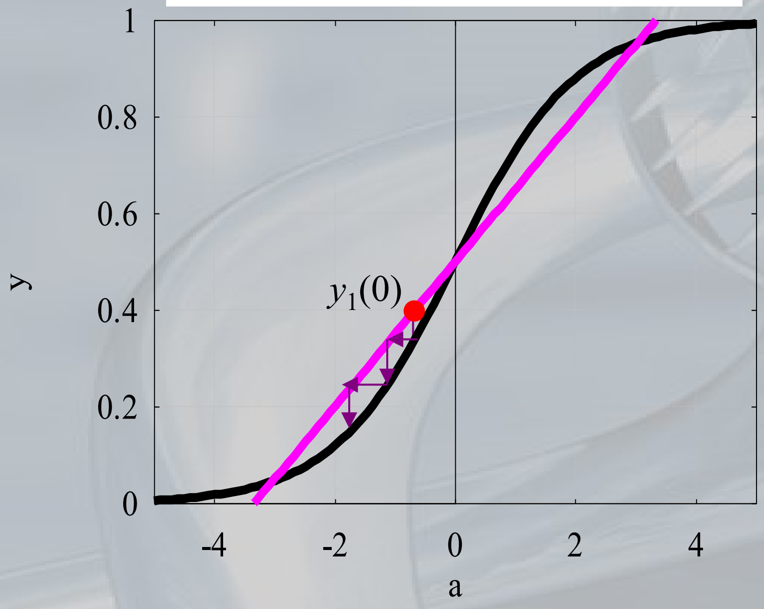
Dynamics of a neuron with feedback – 3



- In the case of continuous transfer functions, at least one solution always exists
- It can be shown that the same property holds true for any recurrent network
- $\lambda=0.5$, $\mu=1.5$ (1 solution)
- $\lambda=0.5$, $\mu=0.5$ (1 solution)

Dynamics of a neuron with feedback – 3

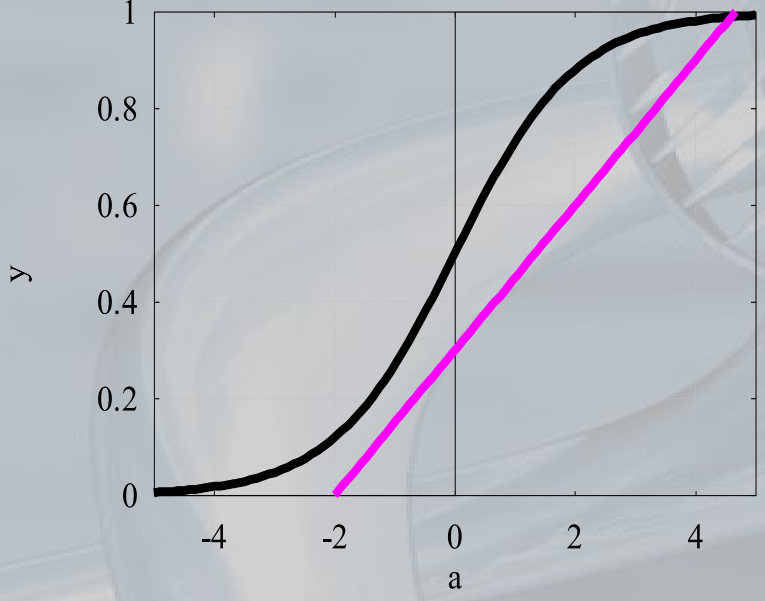
Logistic transfer function



- In the case of continuous transfer functions, at least one solution always exists
- It can be shown that the same property holds true for any recurrent network
- λ=0.5, μ=1.5 (1 solution)
- λ=0.5, μ=0.5 (1 solution)
- λ=0.15, μ=0.5 (3 solutions)

Dynamics of a neuron with feedback – 3

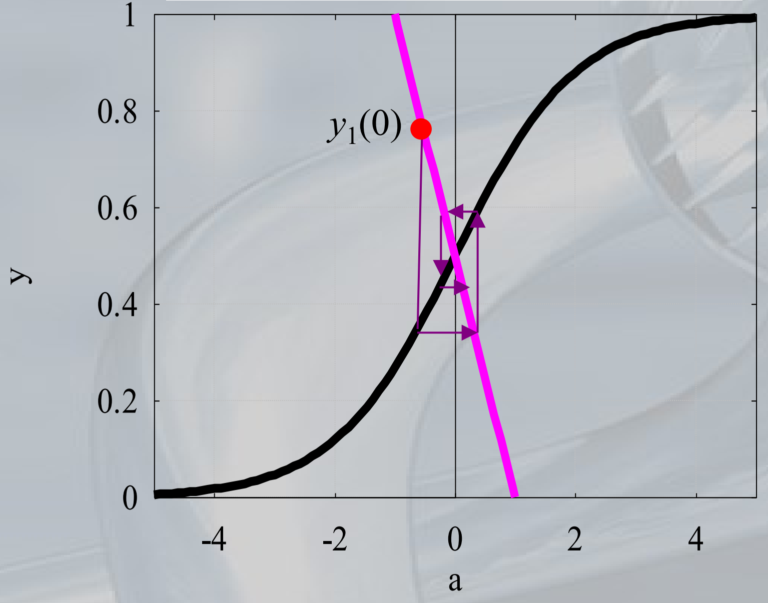
Logistic transfer function



- In the case of continuous transfer functions, at least one solution always exists
- It can be shown that the same property holds true for any recurrent network
- λ=0.5, μ=1.5 (1 solution)
- λ=0.5, μ=0.5 (1 solution)
- λ=0.15, μ=0.5 (3 solutions)
- λ=0.15, μ=0.3 (1 solution)

Dynamics of a neuron with feedback – 3

Logistic transfer function



- In the case of continuous transfer functions, at least one solution always exists
- It can be shown that the same property holds true for any recurrent network
- λ=0.5, μ=1.5 (1 solution)
- λ=0.5, μ=0.5 (1 solution)
- λ=0.15, μ=0.5 (3 solutions)
- λ=0.15, μ=0.3 (1 solution)
- λ=–0.5, μ=0.5 (1 solution)

Dynamics of a neuron with feedback – 4

- The inability to obtain closed–form solutions imposes to proceed in an iterative fashion
- Given the weights and fixed the inputs, an initial value $y_1(0)$ is assigned to the output vector
- Starting from this initial condition, the network then evolves according to a dynamic law for which any solution represents an equilibrium point
- Ultimately, the presence of cyclic paths "introduces a temporal dimension" in the network behaviour

The vanishing/exploding gradient problem – 1

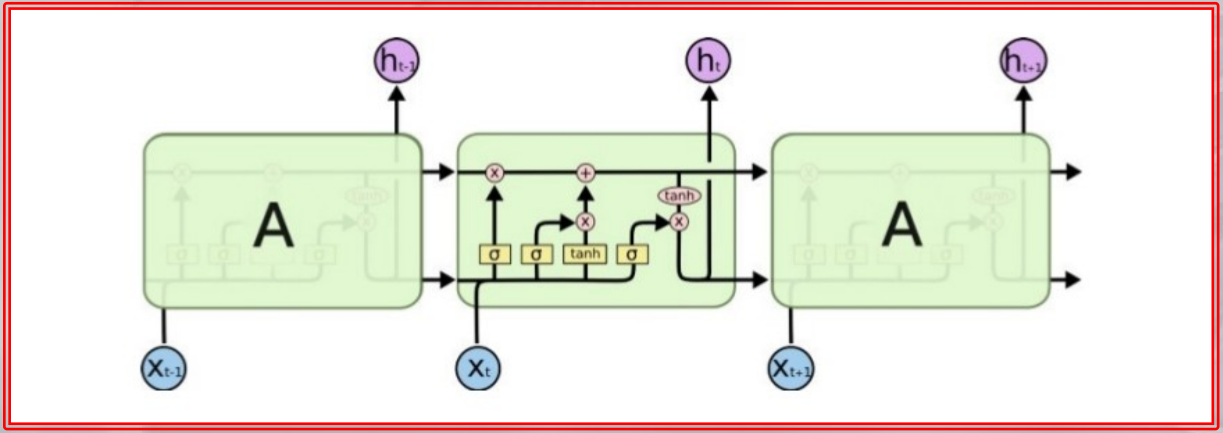
- When training a deep neural network – as the unfolded network actually is – with gradient based learning and backpropagation, partial derivatives are calculated by traversing the network from the final layer to the initial layer; using the chain rule, the deeper layers in the network go through continuous matrix multiplications to calculate their derivatives
- If the derivatives are large then the gradient will increase exponentially during backpropagation, eventually exploding
- Alternatively, if the derivatives are small then the gradient will decrease exponentially, possibly vanishing

The vanishing/exploding gradient problem – 2

- In the case of exploding gradients, the accumulation of large derivatives results in the model being very unstable and incapable of effective learning
- Conversely, the accumulation of small gradients results in a model that is incapable of extracting meaningful information from data, since the weights and biases of the initial layers, which tend to learn the core features from the inputs, will not be updated effectively
- Anyway, long–term dependencies, are difficult to be learned due to the very deep architecture they correspond to

Long–Short Term Memories – 1

- Long–Short Term Memories, LSTMs for brevity, are a variant of RNNs that introduce a number of special, internal gates
- Internal gates help with the problem of learning relationships between both long and short sequences
  - Con: Many more internal parameters, which must be learned, are introduced → Time consuming
  - Pro: Many more internal parameters, which must be learned, are introduced → Flexible



Long–Short Term Memories – 2

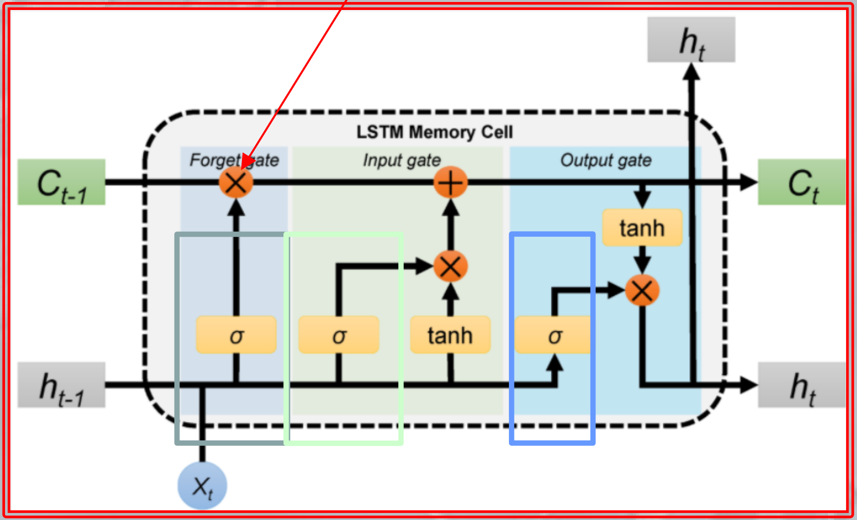
- Forget gate f:
  - Takes previous output h_{t-1} and current input x_t.
  - $f_t \in (0,1)$
  - $f_t = \sigma(\theta_{xf}x_t + \theta_{hf}h_{t-1} + b_f)$
  - If $f_t = 0$: **Forget** previous state, otherwise pass through prev. state.

All θ are weights in single layer perceptrons

- Input gate i:
  - Takes previous output h_{t-1} and current input x_t.
  - $i_t \in (0,1)$
  - $i_t = \sigma(\theta_{xi}x_t + \theta_{ht}h_{t-1} + b_i)$

- Output gate o:
  - $o_t = \sigma(\theta_{xo}x_t + \theta_{ho}h_{t-1} + b_o)$
  - $o_t \in (0,1)$

Hadamard product (element–wise product)

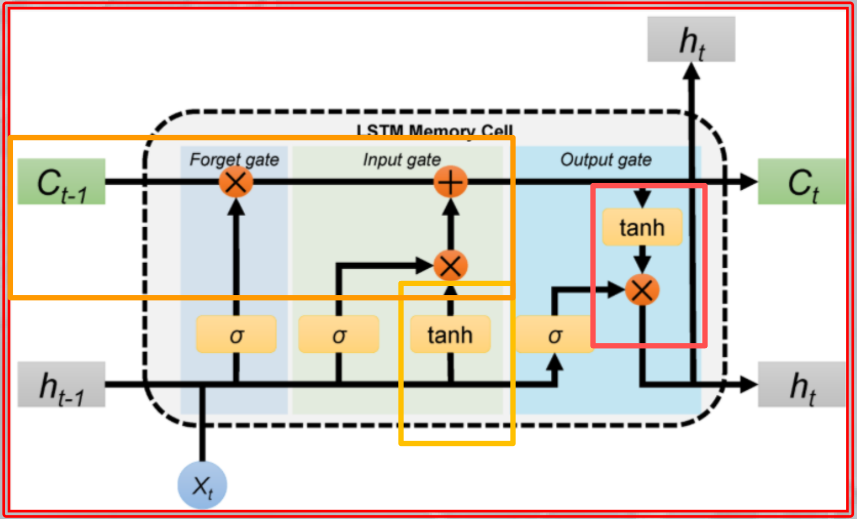


Long–Short Term Memories – 3

- Read gate g:
  - Takes previous output h_{t-1} and current input x_t.
  - $g_t \in$ (-1,1)
  - $g_t = \tanh(\theta_{xg} x_t + \theta_{hg} h_{t-1} + b_g)$

- Cell gate c:
  - New value depends on f_t, its previous state c_{t-1}, and the read gate g_t.
  - Element-wise multiplication: $c_t = f_t \odot c_{t-1} + i_t \odot g_t$.
  - We can learn whether to **store** or **erase** the old cell value.

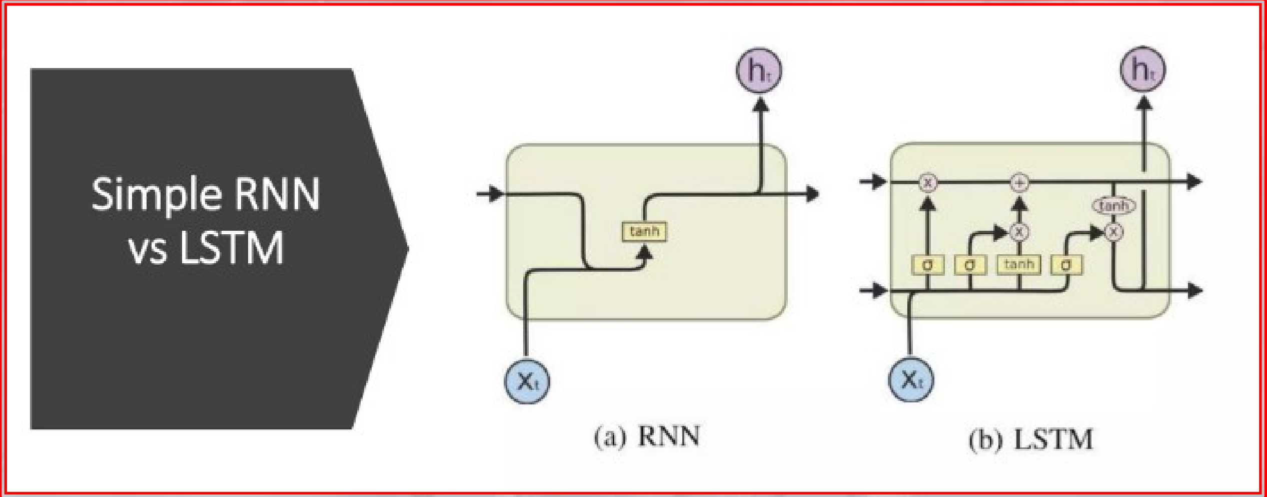
- New output gate h:
  - $h_t = o_t \odot \tanh(c_t)$
  - Will be fed as input into next block.

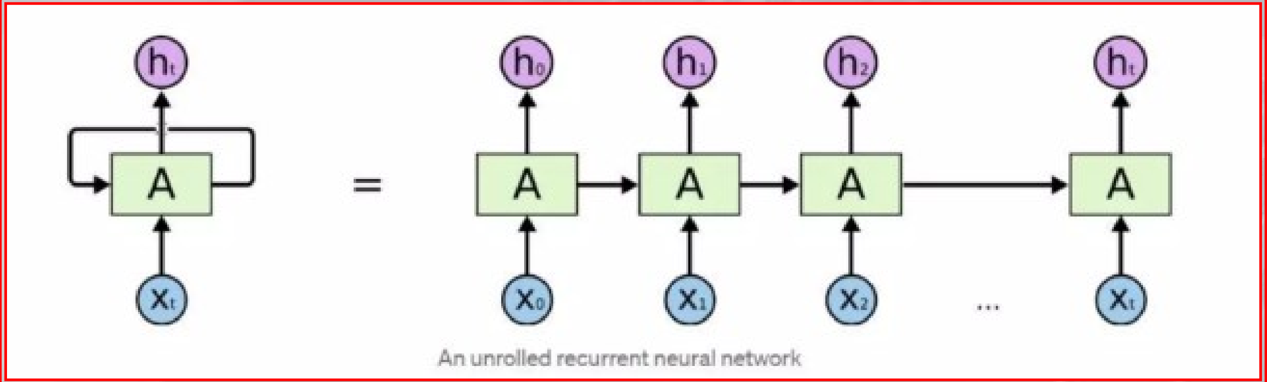


Long–Short Term Memories – 4

- Each cell is responsible for keeping track of the dependencies between the elements in the input sequence
  - The input gate controls the extent to which a new value flows into the cell
  - The forget gate controls the extent to which a value remains in the cell
  - The output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit
- LSTMs learn when to retain a state or to forget it
- Parameters are constantly updated as new data arrives

RNN vs LSTM


Simple RNN vs LSTM
h_t
tanh
X_t
(a) RNN
h_t
tanh
σ
σ
tanh
σ
X_t
(b) LSTM


h_t
A
X_t
=
h_0
h_1
h_2
h_t
A
A
A
A
X_0
X_1
X_2
...
X_t
An unrolled recurrent neural network

Transformers

- The RNN and LSTM neural models were originally designed to process language and perform tasks like classification, summarization, translation, and sentiment detection
- In both models, layers get the next input word and have access to some previous words, allowing it to use the word’s left context
- They used word embeddings where each word was encoded as a vector of 100–300 real numbers representing its meaning
- Transformers extend this to allow the network to process a word input knowing the words in both its left and right context
- This provides a more powerful context model
- Transformers add additional features, like attention, which identifies the important words in this context
- Transformers typically use semi–supervised learning with:
  - Unsupervised pretraining over a very large dataset of general text
  - Followed by supervised fine–tuning over a focused data set of inputs and outputs for a particular task

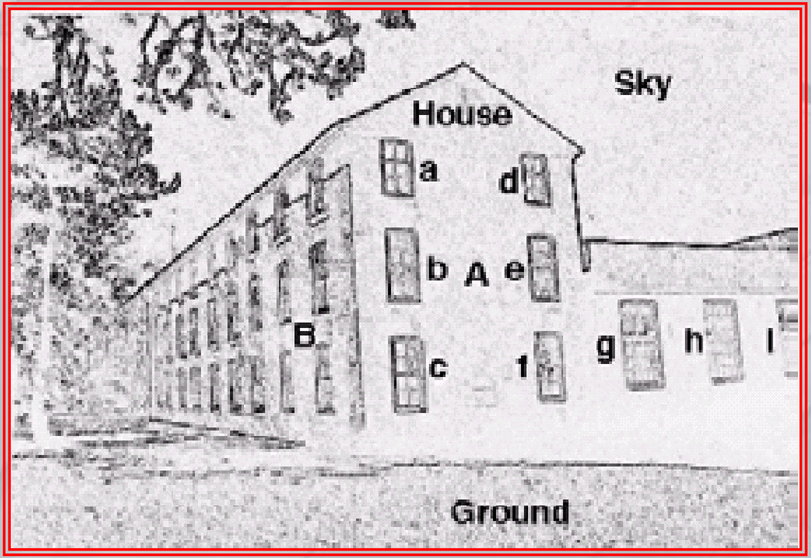
Structured domains – 1

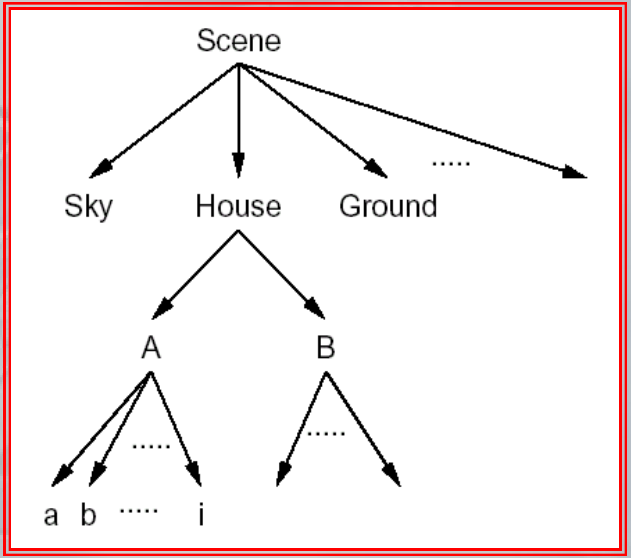
- Even if temporal/sequential data are pervasive in real–world applications, anyway there are also problems for which the information is naturally collected in more complex (possibly hierarchical) structures, like trees or graphs
- Such data have a hybrid nature, both symbolic and sub–symbolic, and cannot be represented regardless of the links between some basic constituent entities
  - Pattern recognition
  - World Wide Web
  - Natural language processing
  - Classification of chemical compounds
  - Analysis of DNA regulatory networks
  - Prediction of active sites on the protein surface
  - …

Structured domains – 2

- Recursive neural networks – that can be viewed as a generalization of recurrent archi- tectures – are modeled on the structure to be learnt, producing an output that takes into account both symbolic data, collected in the node labels, and sub–symbolic information, described by the graph topology

Example 1:
Pattern recognition

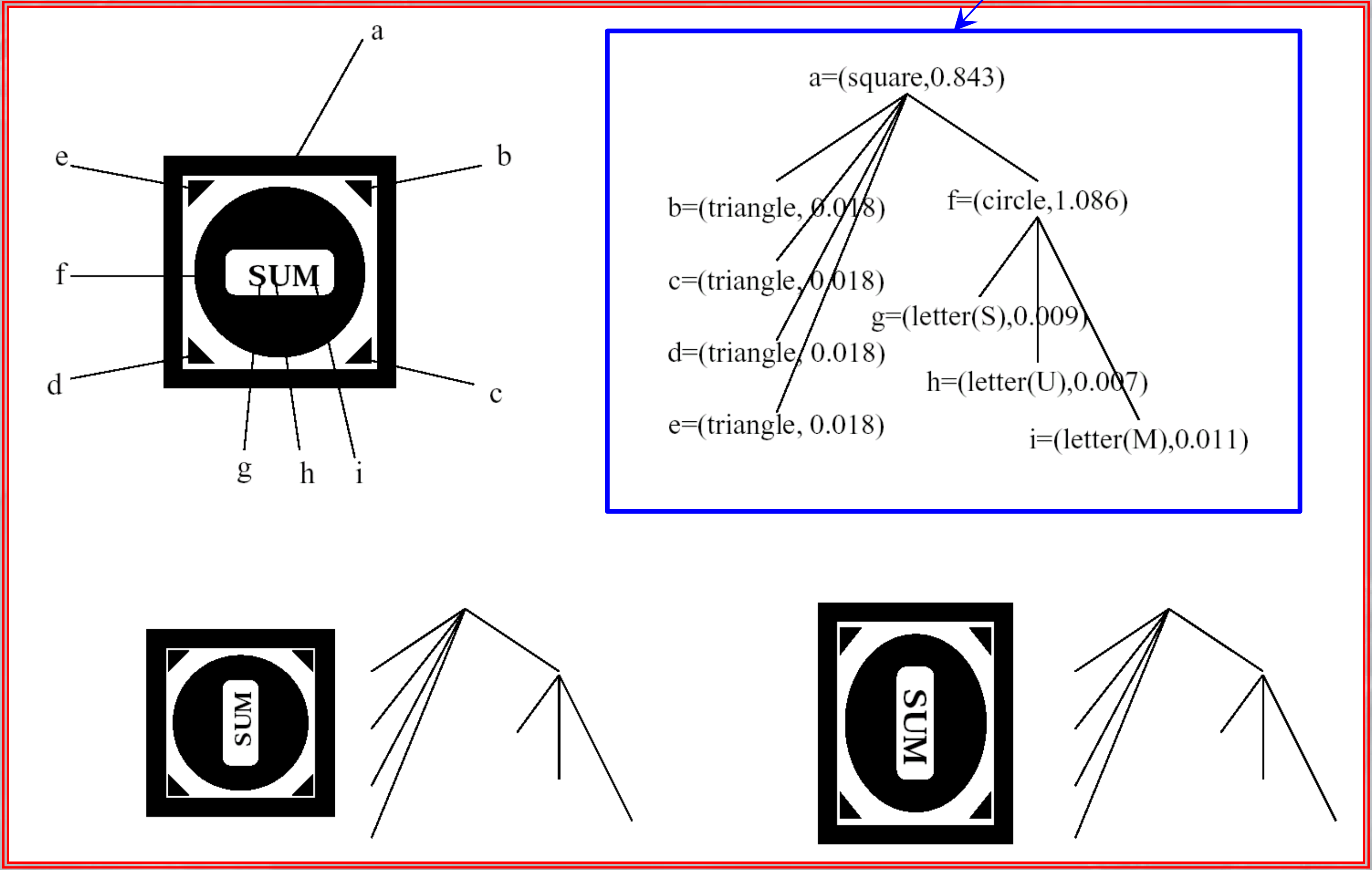




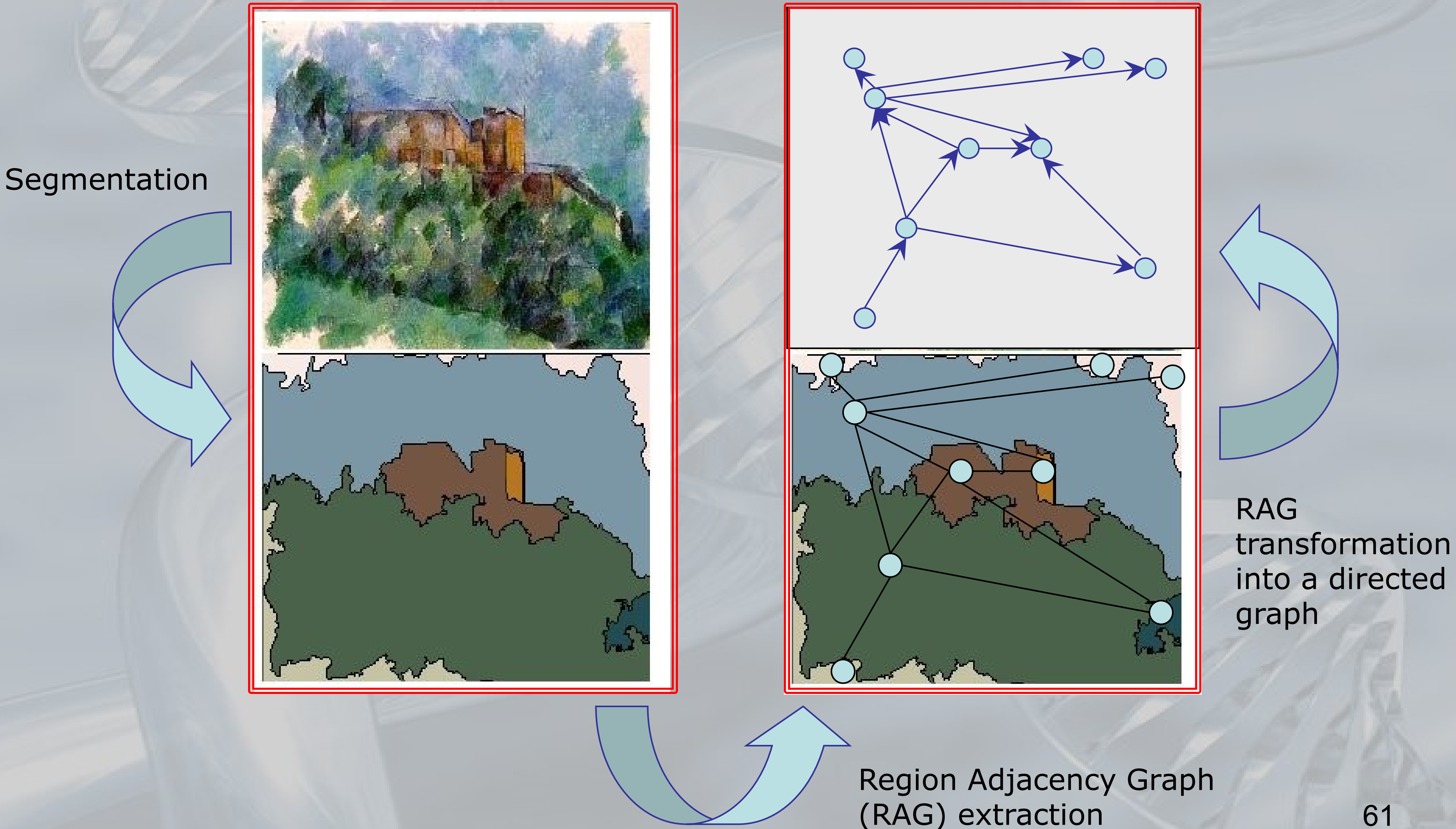
- Each node of the tree contains local features, such as area, perimeter, shape, color, texture, etc., of the related object, while branches denote inclusion relations
- Object detection and recognition

Example 2: Logo recognition

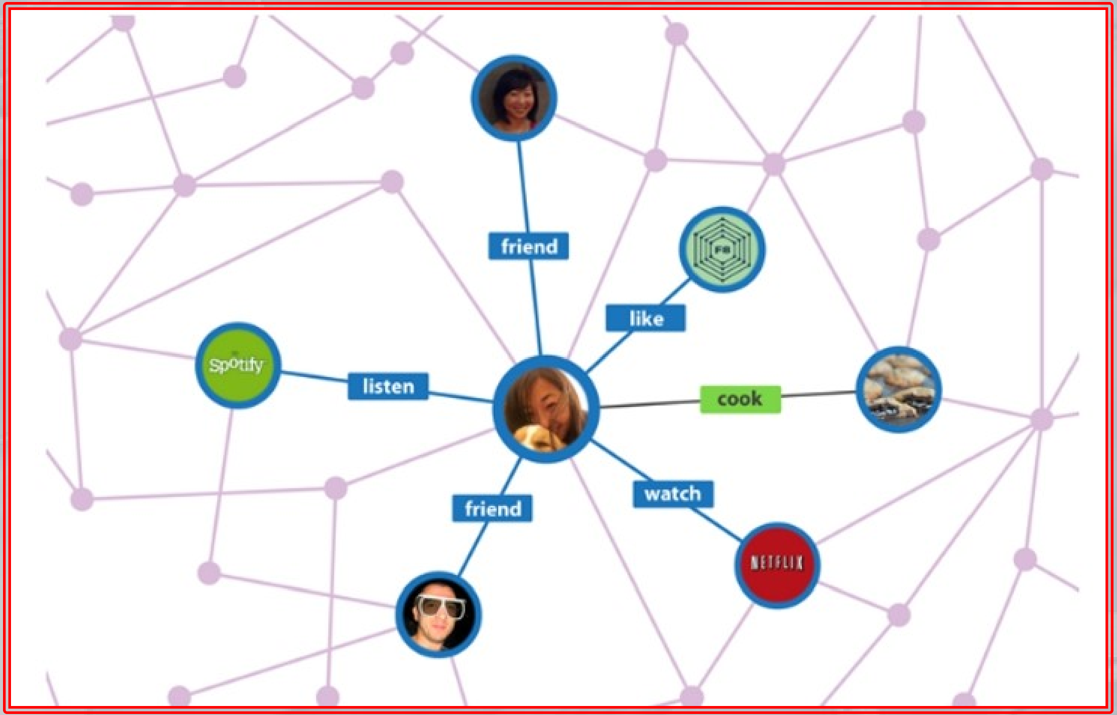
The tree–representation is invariant with respect to rotations, translations and rescaling



Example 3:
A different way for representing images

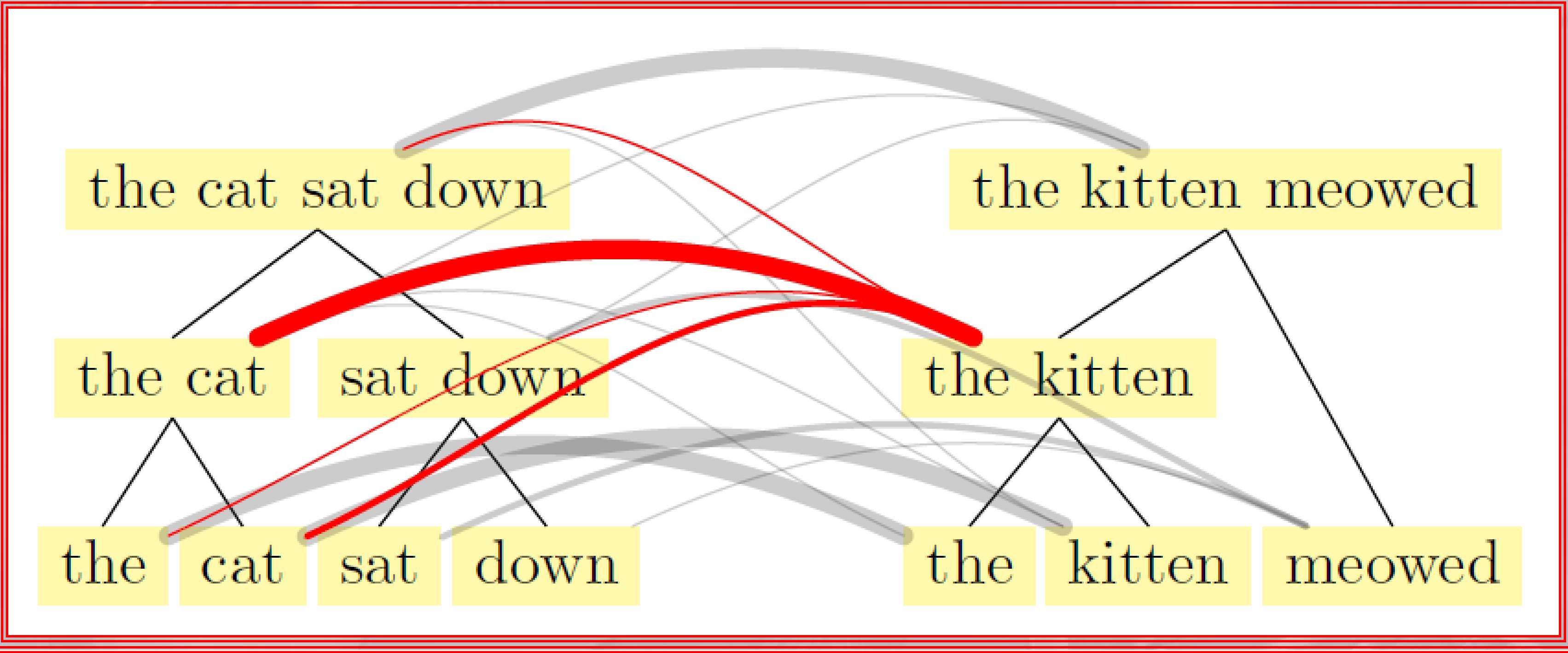

Segmentation
RAG transformation into a directed graph
Region Adjacency Graph (RAG) extraction

Example 4:
An excerpt of the Web



- Social network analysis for:
  - Searching web communities or spam web pages
  - Web document classification
  - Traffic forecasting

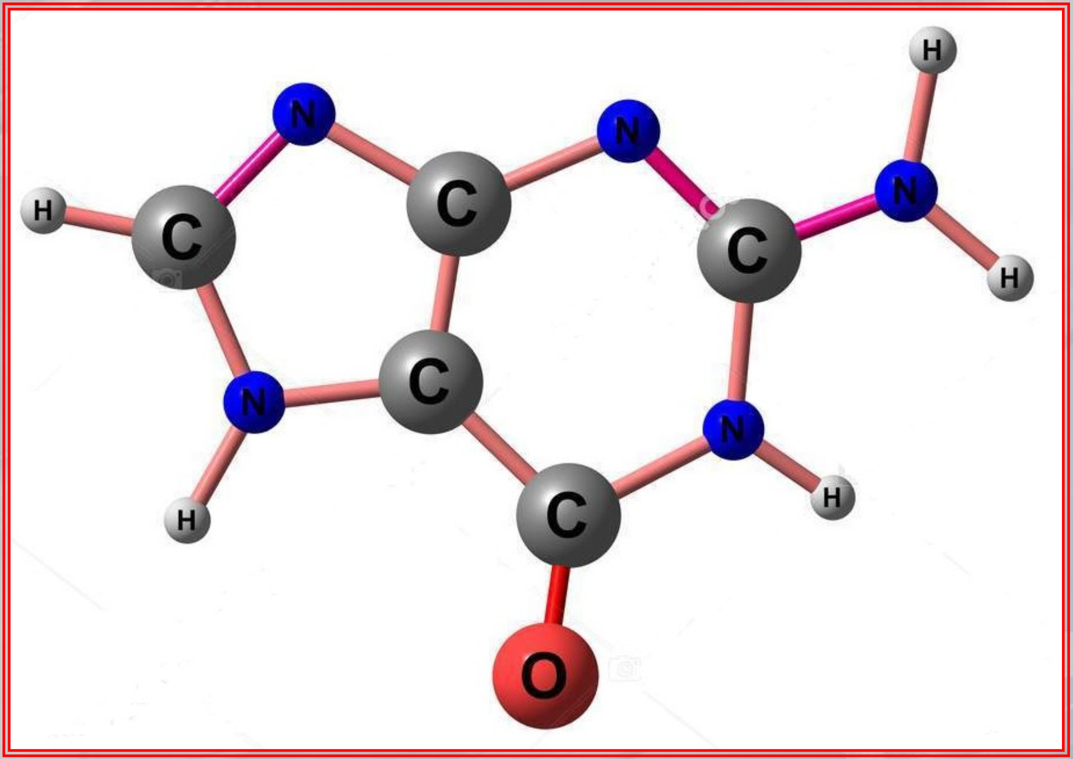
Example 5: Understanding textual entailment and contradiction



"*t* entails *h*" if a human reading *t* would infer that *h* is most likely true or, alternatively, if a human reading *t* would be justified in inferring the proposition expressed by *h* from the proposition expressed by *t*

- Syntactic trees describing the premise–hypothesis sentences for textual entailment

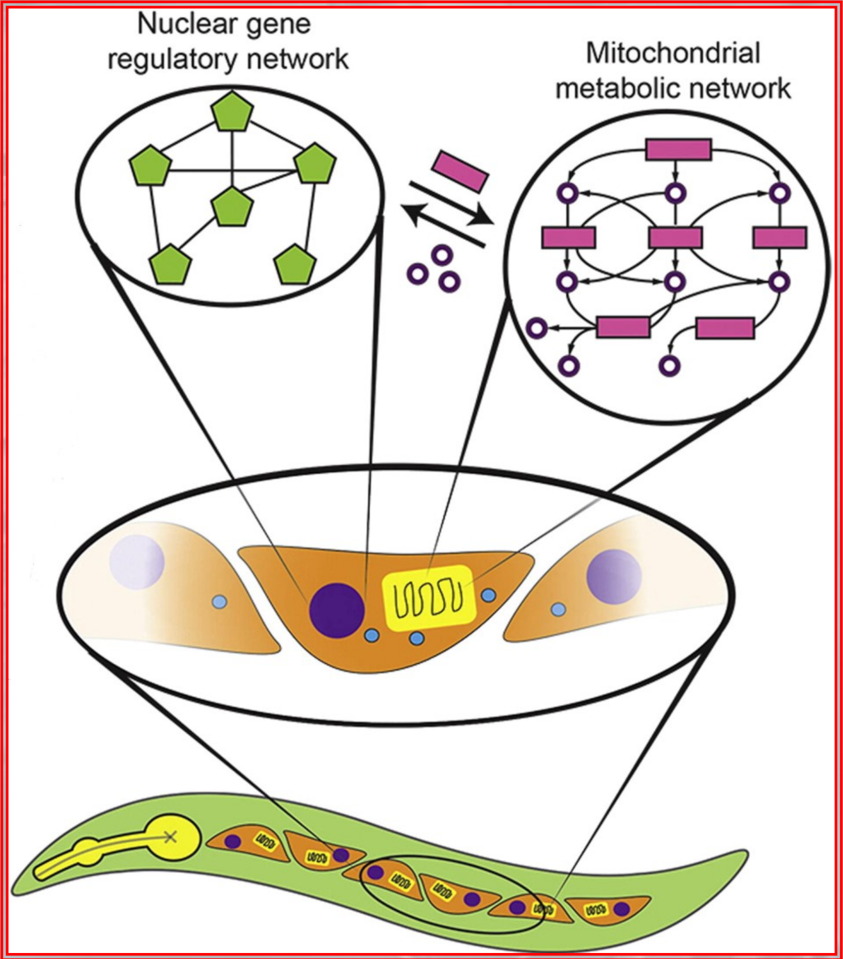
Example 6:
Inference of chemical properties



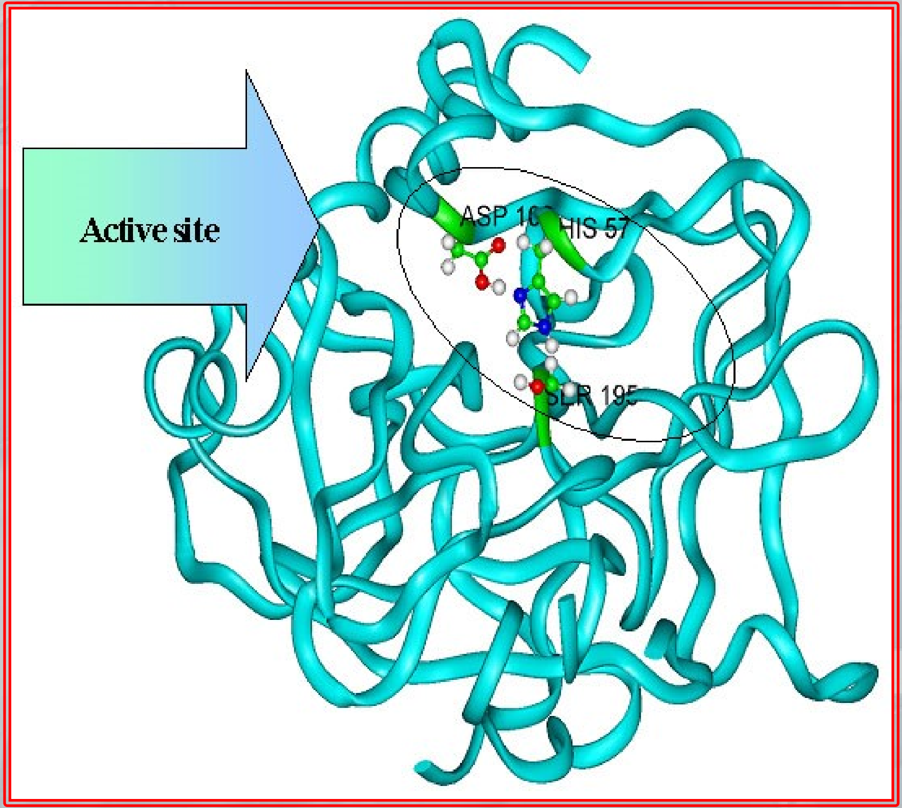
- Chemical compounds are naturally represented as graphs (undirected and cyclic)
- Mutagenicity, genotoxicity, carcinogenicity, etc.

Example 7:
Analysis of DNA regulatory networks

- A gene regulatory network is a collection of protein regulators that interact with each other to govern the gene expression levels of mRNA and proteins
- Regulatory network functionality and interactions


Nuclear gene regulatory network
Mitochondrial metabolic network

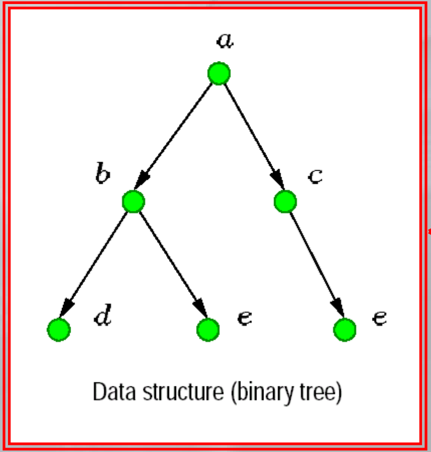
Example 8:
Active site prediction



- Active sites are specific regions of enzymes where chemical reactions occur
- The 3D structure of the enzyme near active sites is analyzed to design drugs which can fit into them

Can we process graphs as they were vectors?

- Graphs can be converted into vectors choosing a visiting criterion of all the nodes, but…
  - …by representing a graph as a vector (or a sequence), we will probably "lose" some potentially discriminating information
- If the "linearization" process produces long vectors/ sequences, learning can become difficult
- Example: the representation of a binary tree using brackets, or based on a symmetric, early, or post-poned visit



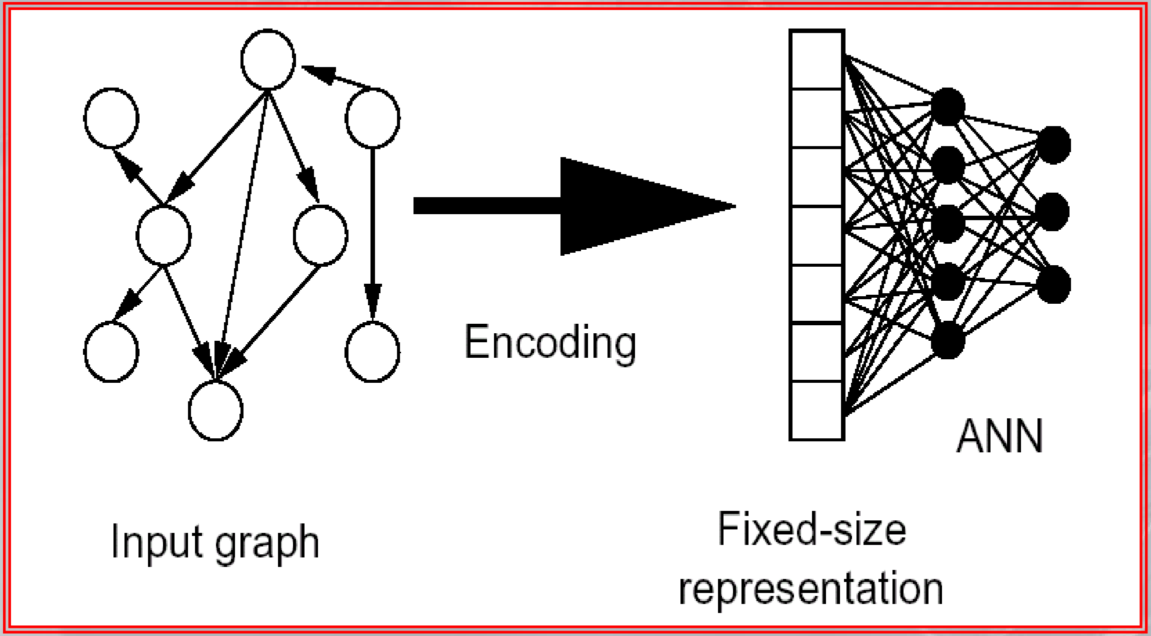
Data structure (binary tree)

$a(b(d,e),c(\varnothing,e))$ Newick format

$dbeace$ Symmetric visit

Heuristics and adaptive coding

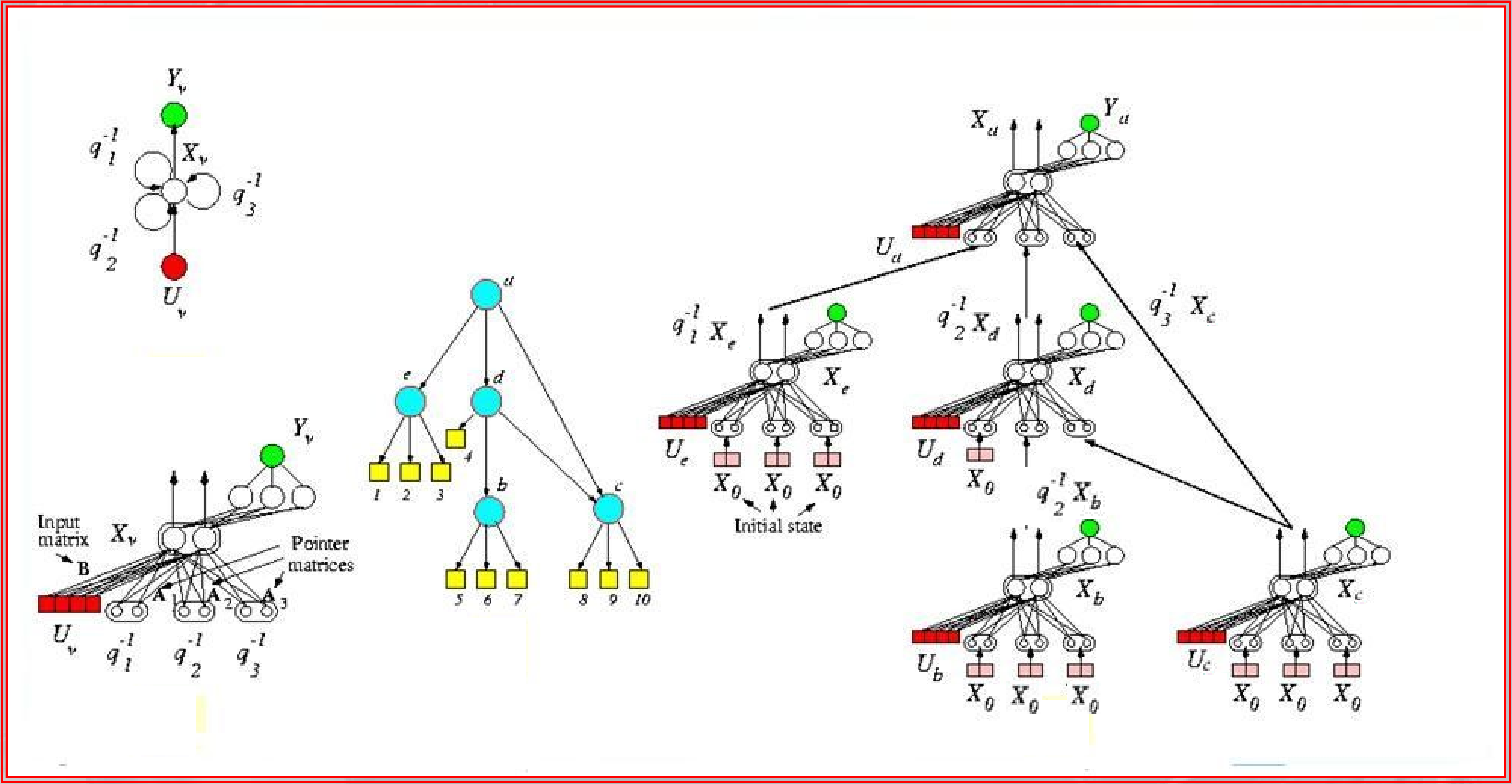
- Instead of selecting a fixed set of features, a network can be trained to automatically determine a fixed–size representation of the graph



Recursive neural networks – 1

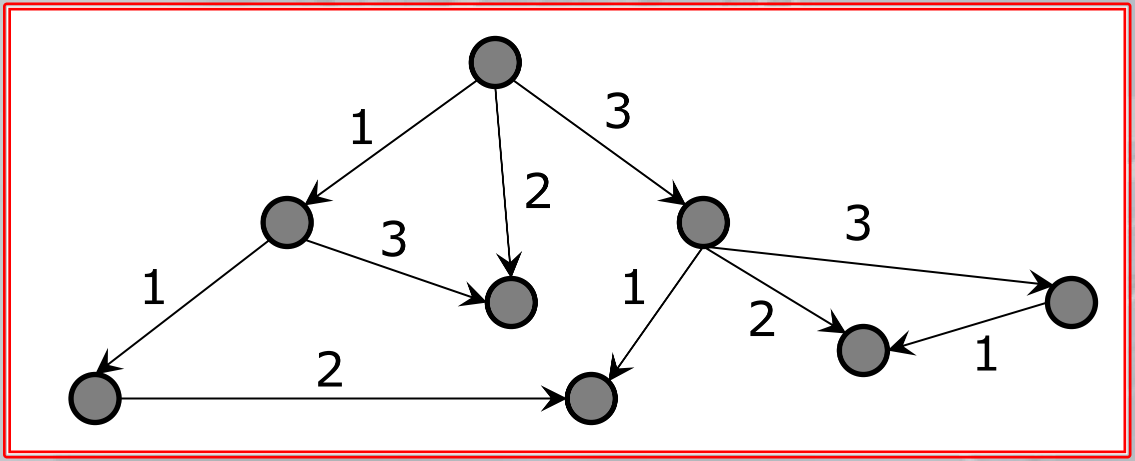
- Based on recursive processing, a fixed–size representation of the graph can be obtained just imposing the supervision on a unique node of the graph
- The recursive network unfolding happens in a spatio–temporal dimension, based on the underlying structure to be learnt
- At each node v, the state is calculated by means of a feedforward network as a function of the node label and of the states of the child nodes
- Moreover, based on the state of the node and on its label, also an output can be calculated at each v

Recursive neural networks – 2



The learning environment: DPAGs – 1

- The learning domain for recursive neural networks is the set of *Directed*, *Positional*, *Acyclic Graphs* (DPAGs)
- A directed graph G, with oriented edges, where edg(G) denotes the set of arcs, is said to be positional if, for each $v\in$vert(G), a total order relationship is defined on the (possibly missing) arcs outgoing from v



The learning environment: DPAGs – 2

- In the chosen framework, G is empty or it possesses a supersource, $s \in$vert(G), such that, for each $v \in$vert(G), v can be reached following a path starting from s
  - If a DPAG does not have a supersource, s should be attached to it with a minimum number of outgoing edges and such that every other node of G is reachable from s
- Given G and $v \in$vert(G), $pa[v]$ is the set of the parents of v, whereas $ch[v]$ is the set of its children
  - The *indegree* of v is the cardinality of $pa[v]$, while its *outdegree* is the cardinality of $ch[v]$

The learning environment: DPAGs – 3

- Graphs used to store structured information are normally labelled:
  - each node contains a set of variables, namely a record, that constitutes the label at that node
  - each field within the label is an attribute, numerical (continuous–valued) or cathegorical (with values in a discrete and finite set)
  - a graph G is uniformly labelled if all the records are similar, that is, comprised of the same fields (in number and type)
- The presence of an arc (v,w) in a labelled graph establishes the existence of a causal link between the variables in v and w

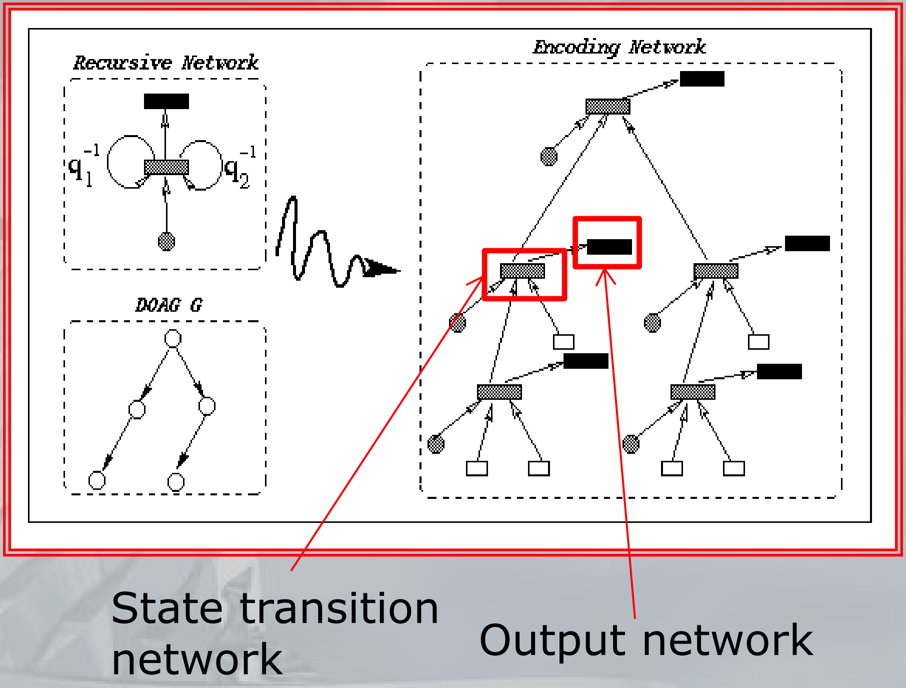
Recursive processing – 1

- The state transition network recursively calculates, the state of the nodes in the graph G

$$\boldsymbol{X}_v = \boldsymbol{f}(\boldsymbol{X}_{ch[v]}, \boldsymbol{U}_v, \vartheta_{(f,v)})$$

- Instead, the output network evaluates the output function $\boldsymbol{g}$

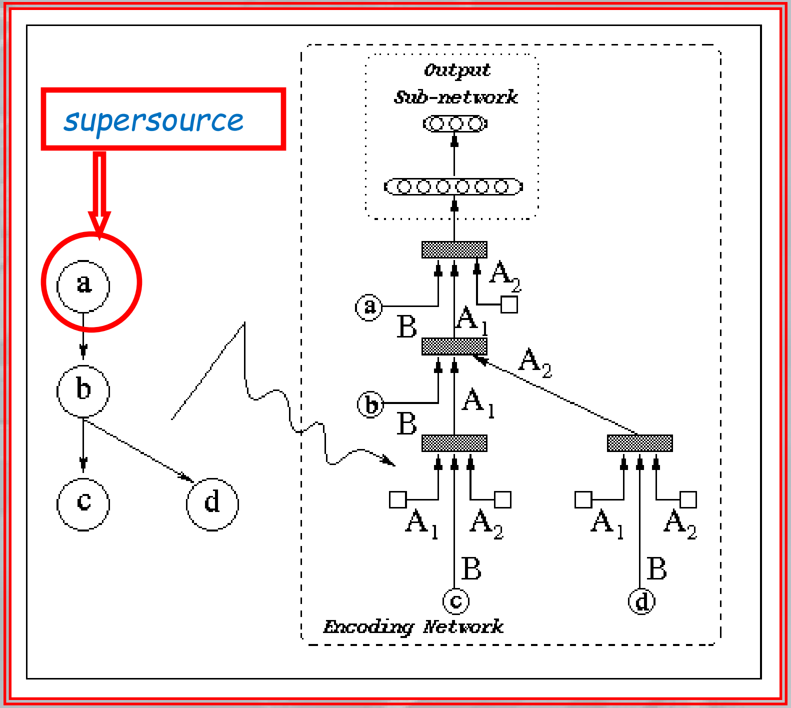
$$\boldsymbol{Y}_v = \boldsymbol{g}(\boldsymbol{X}_v, \boldsymbol{U}_v, \vartheta_{(g,v)})$$



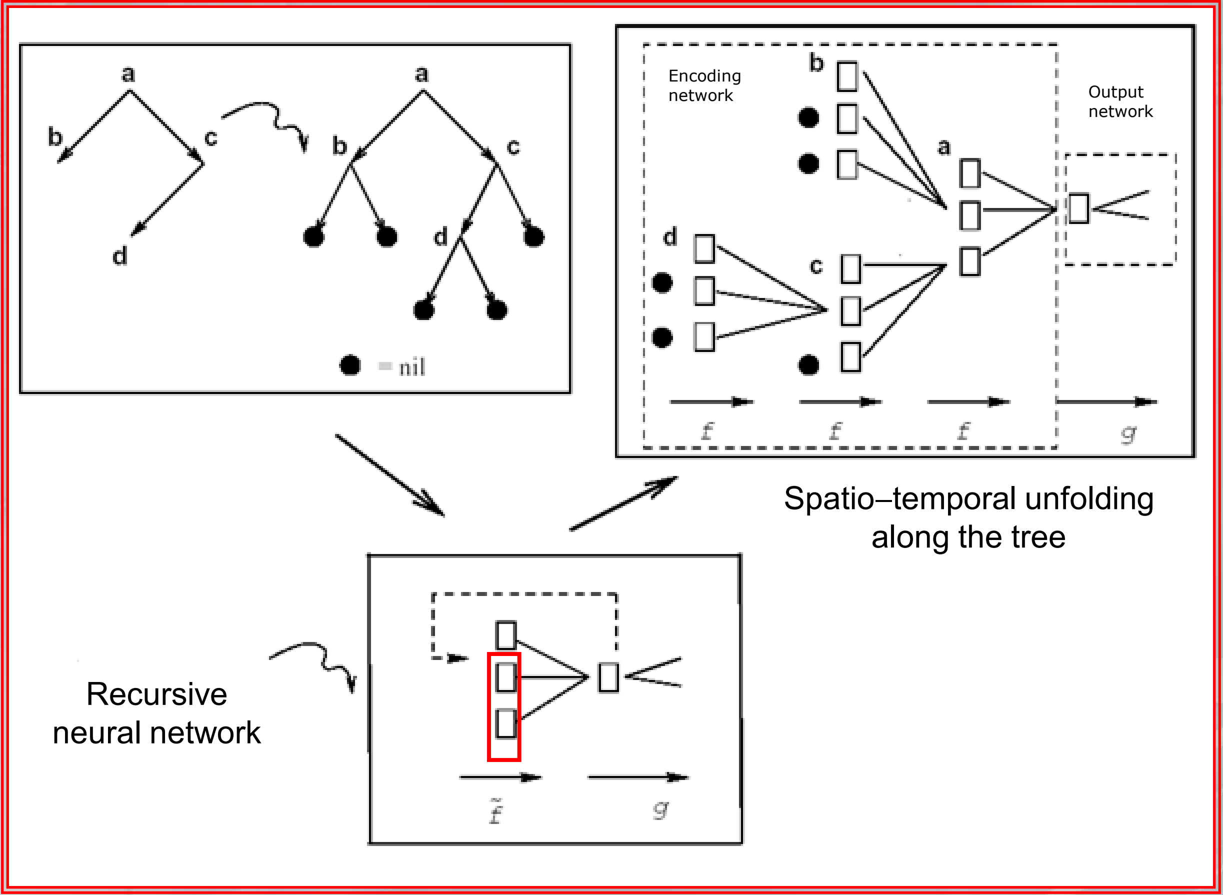
  - $\boldsymbol{U}_v$ is the node v label, $\boldsymbol{X}_{ch[v]}$ collects the states of the child nodes of v, $\vartheta_{(\boldsymbol{f},v)}$ and $\vartheta_{(\boldsymbol{g},v)}$ are connection weights
  - The parametric representations of $\boldsymbol{f}$ and $\boldsymbol{g}$ can be realized through any neural architecture

Recursive processing – 2

- If the output network is placed only in correspondence of the supersource…
  - The recursive network realizes a function from the space of directed, ordered, acyclic graphs to $\mathbb{R}^m$, with m number of output neurons in the output network; in this case, the recursive network produces a supersource transduction
  - The output at the supersource is a vector representation of the information content, both symbolic and topological, of the whole graph



The recursive architecture



Spatio–temporal unfolding along the tree

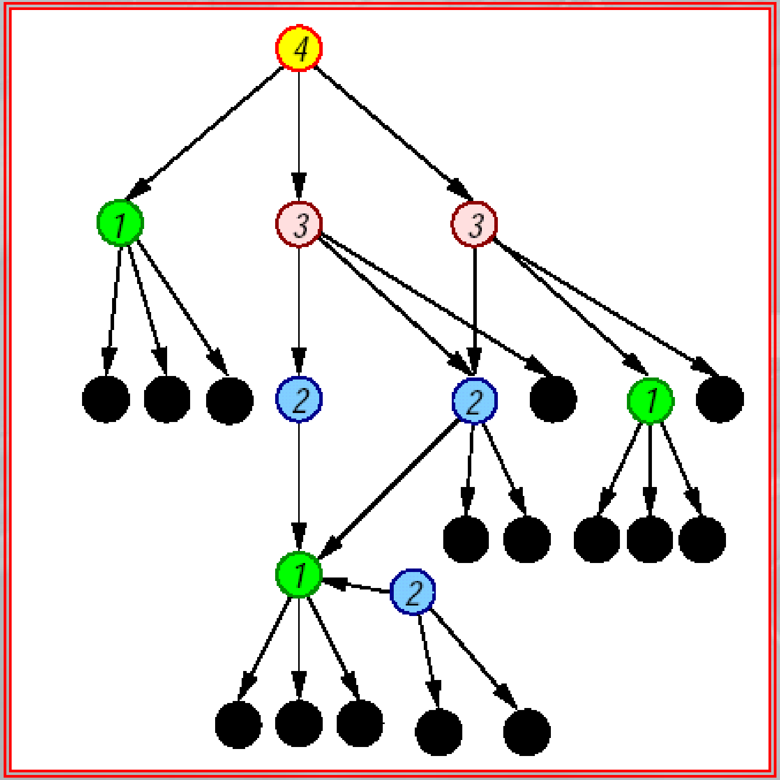
Recursive neural network

$$X_v = f\left(X_{ch[v]},\ U_v,\ \vartheta_{(f,v)}\right)$$
$$Y_s = g\left(X_s,\ U_s,\ \vartheta_{(g,s)}\right)$$

f and g may be realized using MLPs

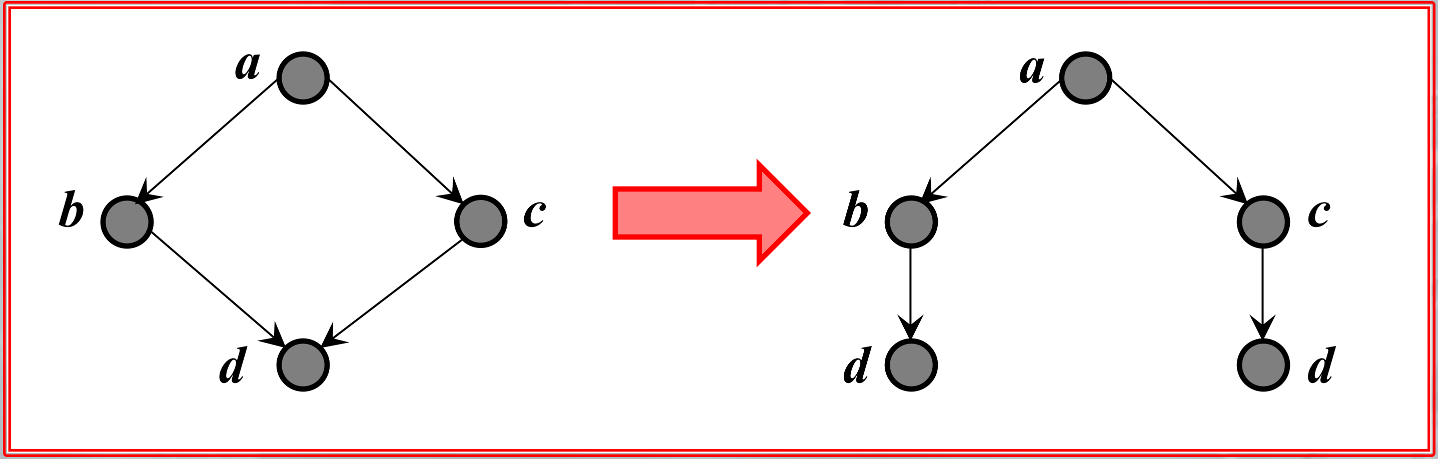
Scheduling

- BOTTOM–UP: we can follow any inverse topological sorting of the graph – *data flow* processing scheme
- Some vertices can be updated in parallel: sorting is not unique!



From DPAGs to trees

- Recursive neural networks do not distinguish between DPAGs and trees that are *recursive–equivalent*
- Actually, each DPAG can be encoded by a recursive–equivalent tree
  - For each node v of the DPAG with $pa[v]>1$, there exist $pa[v]$ copies of v in the corresponding tree



Some comments…

- Recursive neural networks are a powerful computational tool for processing structured data, able to bridge the historical gap between classical connectionist techniques, tailored to poorly organized data, and a wide variety of real–world problems in which the information is “naturally” encoded in basic entities and relationships among them
- Recursive networks process information in the form of directed, positional and acyclic graphs or, more simply, of recursive–equivalent trees

Some more comments…

- At each pseudo–time, a feedforward neural network is "stimulated" with the label of a node of the graph, and with the states calculated at its child nodes, according to a training strategy similar to that of recurrent networks
- However, if the recurrent network processing is carried out on the basis of the natural flowing of data during time, recursive networks follow the partial order imposed by the arcs of the graph, and "unfold" in the spatio–temporal dimension under-lying it
- Some parallelism is permitted for those nodes whose child states have been always calculated, based on the reverse topological order processing

Useful notation for learning – 1

- $\boldsymbol{T}$ — is a set of trees with labelled nodes, and with a maximum *outdegree* equal to k
- $\boldsymbol{U}_v$ — represents the label at node v; it belongs to L, that can be constituted by a finite or an infinite set of elements
- $c(\boldsymbol{U}_v)$: $\forall \boldsymbol{U}_v \in L$, $c(\boldsymbol{U}_v) \in \mathbb{R}^l$

Useful notation for learning – 2

- Definition

  Let $\mathbf{Y}_{\boldsymbol{F}} \in \mathbb{R}^m$ be the encoding of the empty tree

  For $\boldsymbol{f}$: $\mathbb{R}^{l+km} \rightarrow \mathbb{R}^m$, the induced function $\tilde{\boldsymbol{f}}$ is recursively defined by

$$\boxed{\tilde{f}(t) = \begin{cases} \mathbf{Y}_F & \text{if } t \text{ is empty} \\ f(c(\mathbf{U}_v), \tilde{f}(t_1), \ldots, \tilde{f}(t_k)) & \text{otherwise} \end{cases}}$$

  where $t \in \boldsymbol{T}$ and $t_1, \ldots, t_k$ represent its k subtrees (originating from the root)

Useful notation for learning – 3

- The function $\boldsymbol{l}$:$\boldsymbol{T}\rightarrow\mathbb{R}^q$ can be calculated by a recursive neural network if the following two functions, that can be realized via MLPs, exist:

$$\tilde{\boldsymbol{f}}\colon \boldsymbol{T}\rightarrow\mathbb{R}^m$$

$$\boldsymbol{g}\colon \mathbb{R}^m\rightarrow\mathbb{R}^p$$

together with an affine transformation,

$$\boldsymbol{A}\colon \mathbb{R}^p\rightarrow\mathbb{R}^q$$

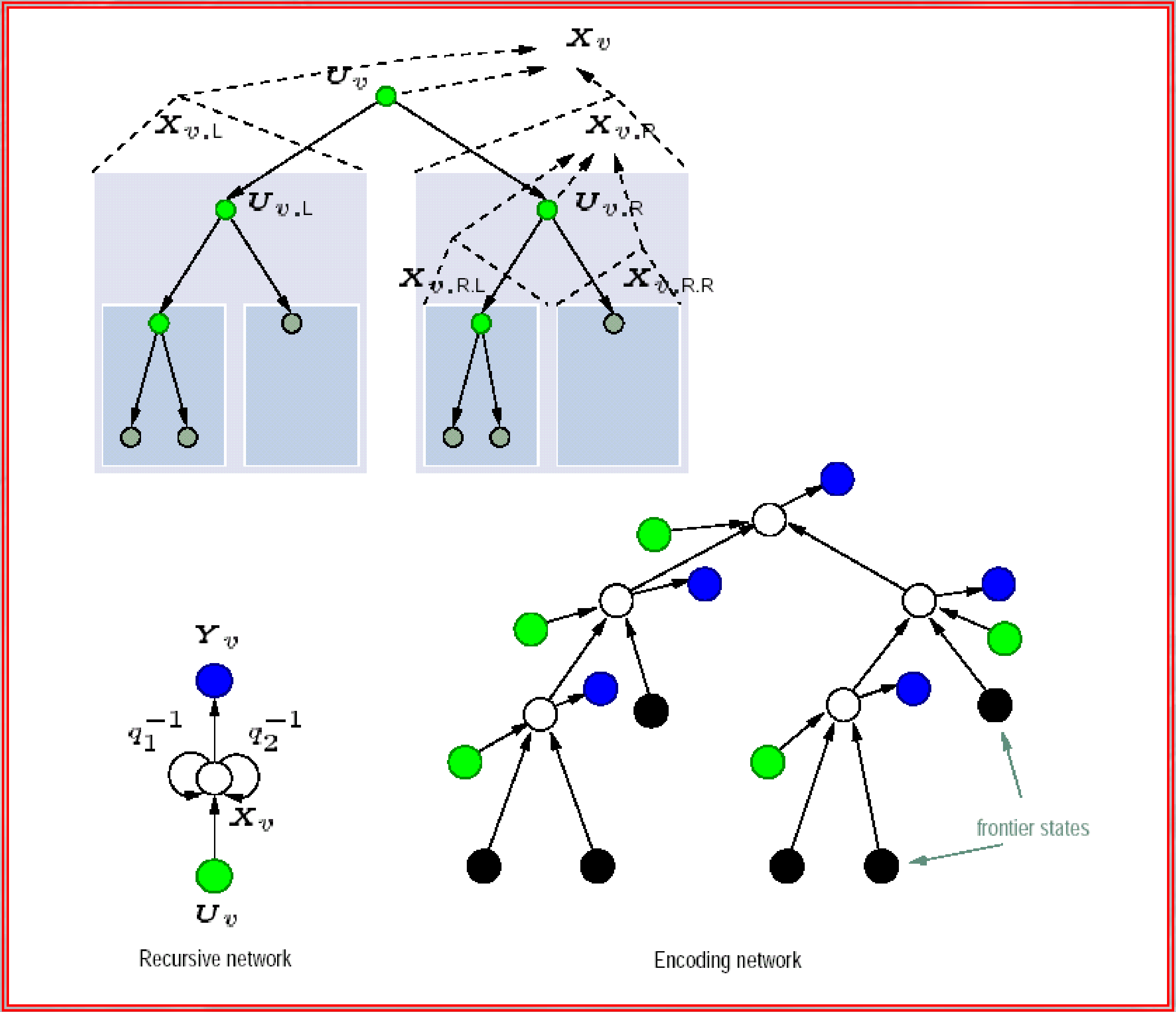
such that

$$\boldsymbol{l} = \boldsymbol{A} \circ \boldsymbol{g} \circ \tilde{\boldsymbol{f}}$$

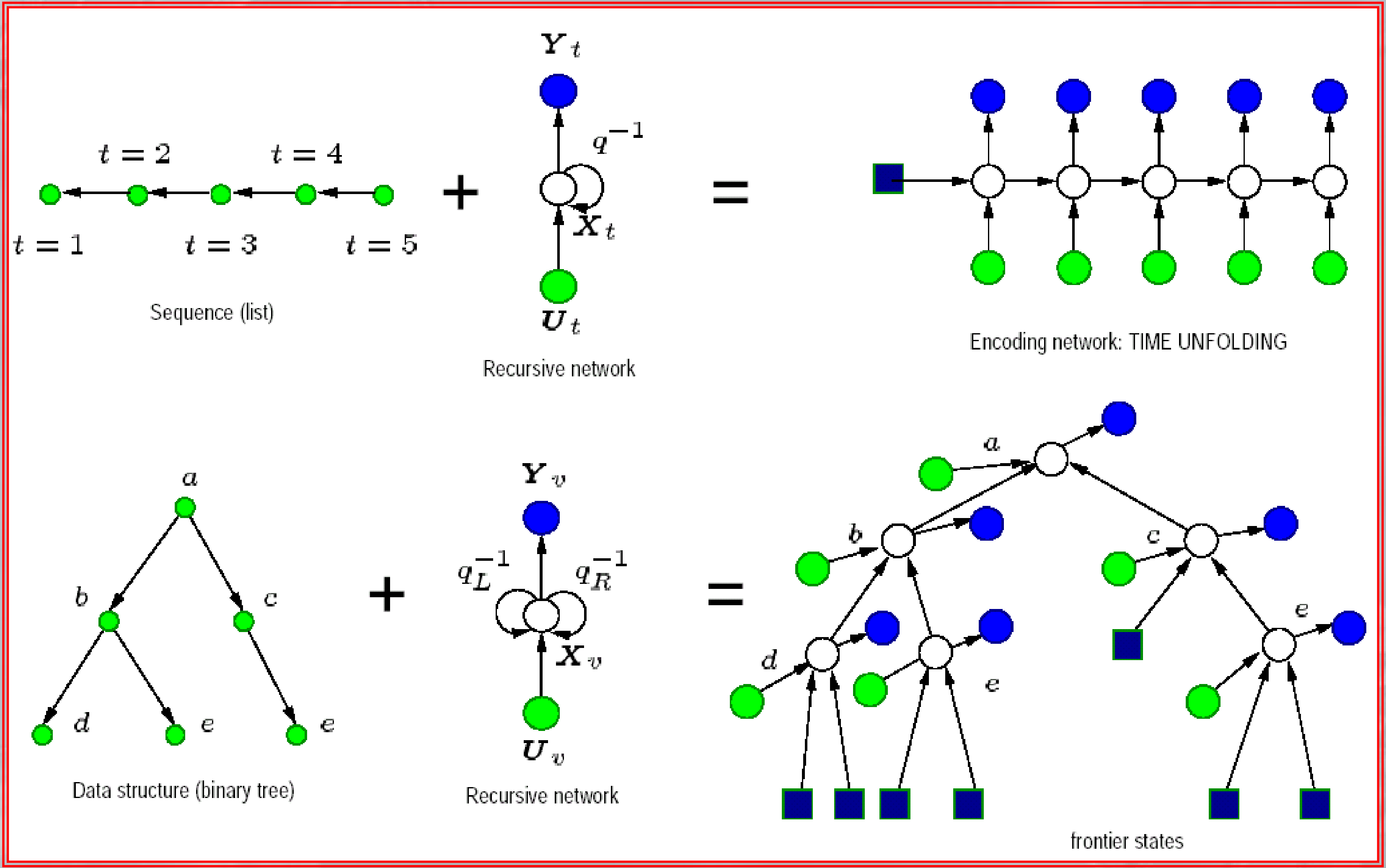
Useful notation for learning – 4

- Remarks
  - The weight updating process is performed by means of the *BackPropagation Through Structure* algorithm, which corresponds to the standard BP on the *unfolded* network, also called the encoding network
  - During learning, all the corresponding weights in distinct hyperlayers of the encoding network are forced to maintain the equality constraint

The encoding network: binary trees



Structured data + recursive networks = encoding networks

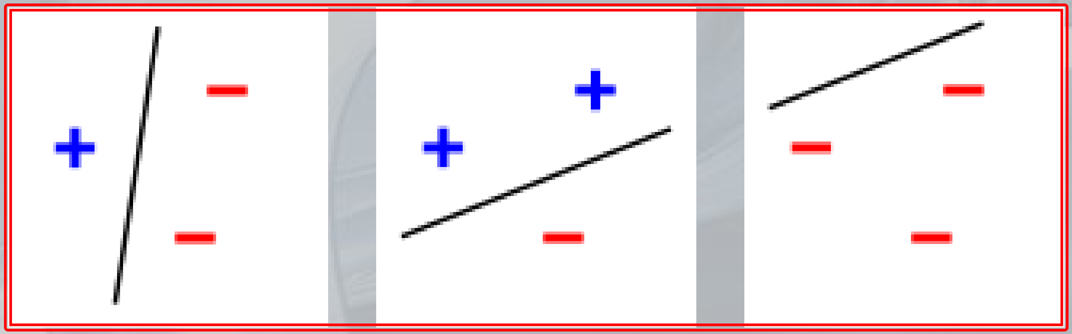
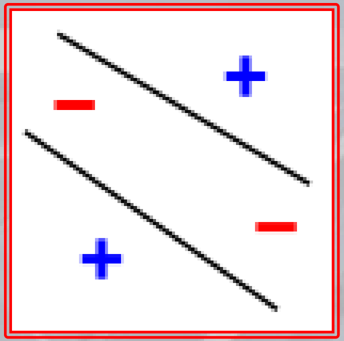


Recursive network training – 1

- So as in BPTT, the BackPropagation Through Structure (BPTS) algorithm collects, for each structure, the individual contributions of the gradients, corresponding to all the copies of the same weight, in a unique contribution, which will be used to update each copy
- The error backpropagation procedure follows the paths traced by the arcs of the graph, starting from the output subnet, through the encoding network, up to the leaves
- If *on–line* learning is performed, the weights are updated, structure by structure, after the presentation of each structure to the network; otherwise, in *batch mode*, the contributions to the gradient are stored with respect to the entire training set and the weights are updated only after all the structures have been presented to the network

Recursive network training – 2

- The *VC–dimension* (or the Vapnik–Chervonenkis dimension) is a measure of the neural network ability to learn (to generalize to new examples)
- Let f be a binary classifier depending on a set of parameters ϑ; f is said to *shatter* the set of data points $(x_1,x_2,\ldots,x_n)$ if, for all the possible assignments of targets to those points, there exists a ϑ such that the model f makes no classification errors

The VC–dim of a linear classifier is 3

Recursive network training – 3

- The VC–dimension of a classifier f is the cardinality of the largest set of points that the model can shatter
- Based on the VC–dimension, a probabilistic upper bound on the test error of a classification model may be estimated

$$\Pr\left(\text{test error} \leqslant \text{training error} + \sqrt{\frac{1}{N}\left[D\left(\log\left(\frac{2N}{D}\right)+1\right)-\log\left(\frac{\eta}{4}\right)\right]}\right) = 1-\eta$$

where D is the VC–dimension, $\eta\in(0,1]$, and N is the dimension of the training set

- The formula is valid when $D << N$ otherwise the test error may be much higher than the training error, due to overfitting

Recursive network training – 4

- Since there is a direct dependence, a high VC–dimension should be a marker of a classifier having poor generalization capabilities
- Conversely, an efficient learning algorithm has to ensure a well–trained network in a “reason-able” amount of time

Recursive network training – 5

- If a gradient method (such as BPTS) is used, numerical problems may occur: because of the error backpropagation, learning in deep networks can have a prohibitive duration or produce instability behaviours
- Having no *a priori* knowledge on the probability distribution of the training set, there are no guarantees that the trained network provides good performance when processing new examples
  - The VC–dimension of recursive networks grows quadratically with the dimension of the patterns to be learnt, e.g., sequence length, tree height, etc., rapidly tending to infinity

Collisions

- Definition

  Given a recursive neural network, trees t_1 and t_2 *collide* if the root states, calculated by the network for both trees, are identical, i.e.,

$$\tilde{\boldsymbol{f}}(t_1) = \tilde{\boldsymbol{f}}(t_2)$$

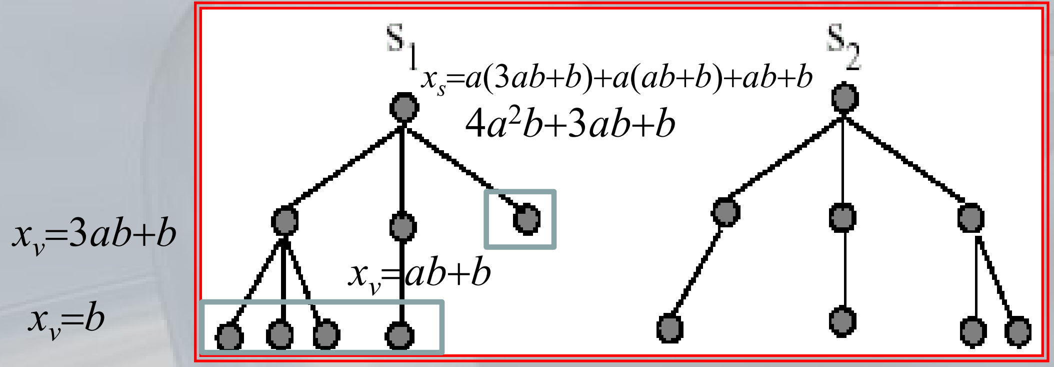
- Remark

  Given an injective function $\boldsymbol{l}$, do a function $\boldsymbol{g}$ exist such that $\boldsymbol{l} = \boldsymbol{g} \cdot \tilde{\boldsymbol{f}}$ on $\boldsymbol{T}$?

  - If $\tilde{\boldsymbol{f}}$ does not produce collisions on $\boldsymbol{T}$, $\boldsymbol{g}$ exists
  - If $\tilde{\boldsymbol{f}}$ causes collisions on $\boldsymbol{T}$, $\boldsymbol{g}$ does not exist

A collision example

- Hypotheses:
  - l=m=1 (label and state dimensions), k=3 (out-degree); x_v represents the state at node v
  - Network architecture: linear recursive network having equal weights w.r.t. each subtree, a_k=a, $\forall k$; b weighs the node label
  - Null frontier state, x_f=0, for each leaf; U_v=1, $\forall v$



Trees that share the same number of nodes at each layer do collide!

$$\boxed{x_{s_1} = x_{s_2} = 4a^2b+3ab+b}$$

Recursive computational power

- Theorem 1

  Collisions occur when

  $$am < k^h$$

  [actually $(k^{h+1}-1)/(k-1)$] where a is the number of bits used to represent each component of the state vector, m is the state vector dimension, k is the maximum outdegree of the trees, and h their maximum height
- Examples

  The following sets of trees cannot be codified using only four bytes (i.e. collisions occur):
  - Binary trees with $h>5$ and $m=1$
  - Binary trees with $h>10$ and $m=32$
  - Trees with outdegree equal to 5 with $h>2$ and $m=1$
  - Trees with outdegree equal to 10 with $h>1$ and $m=3$
  - Trees with outdegree equal to 10 with $h>2$ and $m=31$

Linear recursive networks – 1

- In linear recursive networks all the neurons have linear activations (calculated as products between connection weights and inputs) and a neuron output function that coincides with the identity
- Classical linear algebra tools can, therefore, be used to establish conditions on their dynamical properties and on their ability to encode and classify structured information
- Many of the detectable limits for linear networks are intrinsically related to the recursive framework and can be directly extended to the general model, definitely establishing its computational capacity and its applicability ambit

Linear recursive networks – 2

- In general, even if the number of the state neurons exponentially grows with the height of the structures to be learnt, in order to avoid collisions, however, significant classification problems on trees can be solved with the use of a “reasonable” amount of resources
- In fact, in most of the problems, we do not pretend to *distinguish all the trees*, but rather to highlight some particularly significant classes
- The root state must encode only that a certain tree belongs to a given class

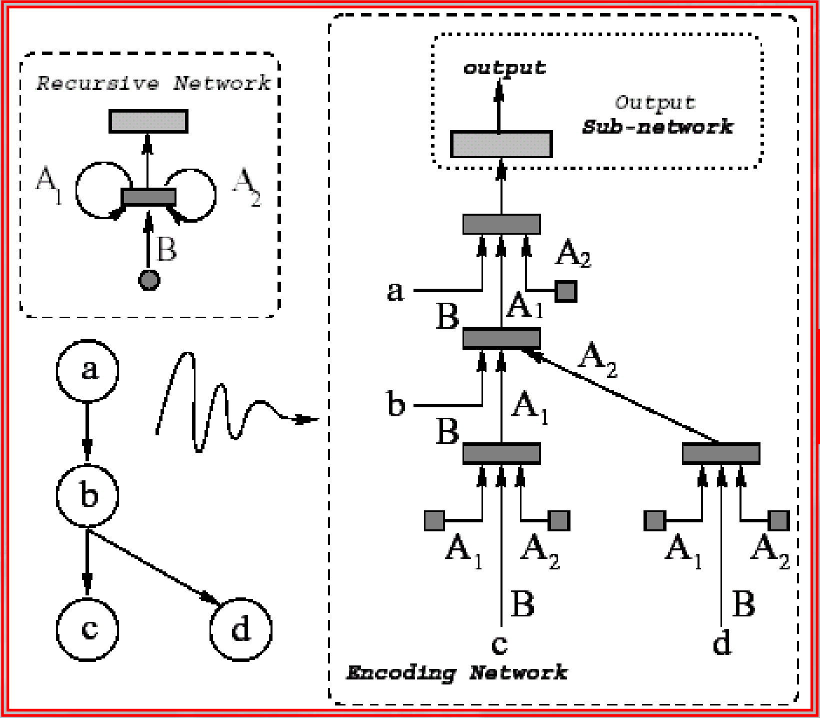
Linear recursive networks – 3

- Actually, linear recursive networks, with a limited number of parameters, can still evidence interesting properties of tree structures, distinguishing these structures according to ...
  - …the number of nodes in each level
  - …the number of leaves
  - …the number of left and right children in binary trees
  - …

Linear recursive networks – 4

- In linear recursive networks…

$$f(\mathbf{X}_1,\ldots,\mathbf{X}_k,\mathbf{U}) = \sum_{i=1}^{k} A_i\mathbf{X}_i + \mathbf{BU}$$
$$g(\mathbf{X}) = \mathbf{CX}$$



- If the frontier state is $X_0=0$, we can calculate the network output as:

$$C(A_1(A_1Bc + A_2Bd + Bb) + Ba)$$

$X_c=2X_0+Bc=Bc$ $X_d=Bd$
$X_b=A_1Bc+A_2Bd+Bb$
$X_a=A_1(A_1Bc+A_2Bd+Bb)+Ba$

Linear networks: how to avoid collisions

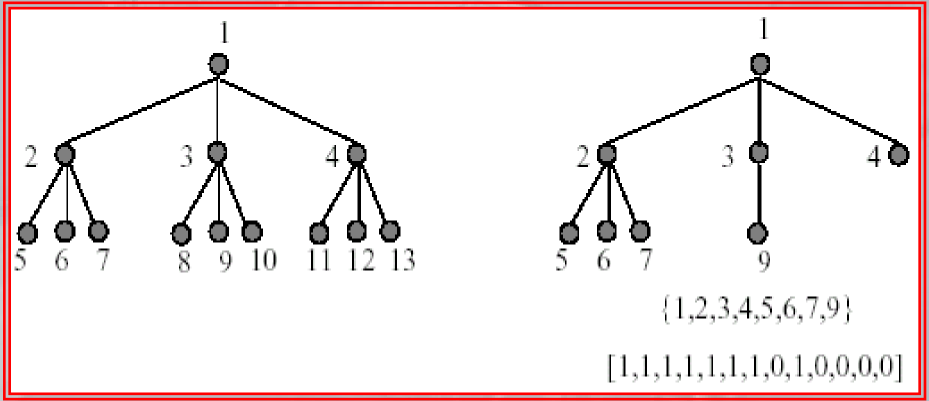
- Definition
  Let p be a natural number and let $I \subseteq \mathbb{R}$; let $\boldsymbol{T}_{p,I}$ the class of trees with height p at most, and with labels belonging to I
- Theorem 2
  Let us consider the class $\boldsymbol{T}_{p,1}$, and let $X_0=0$ be. For any p and any path enumeration of $\boldsymbol{T}_{p,1}$, a recursive network exists for which no collisions occur; moreover:

  1. The state $X_s \in \mathbb{R}^n$, with $$\boxed{n = \frac{k^{p+1}-1}{k-1}}$$
  2. The recursive network calculates

$$\boxed{\mathbf{X}_s[i] = \begin{cases} 1 & \textit{if the input tree contains the i–th path} \\ 0 & \textit{otherwise} \end{cases}}$$

Path enumeration

- A way to achieve an enumeration of the paths of a tree is that of ordering the nodes of the complete tree of height p



  - Each tree can be represented by the set of its nodes ({1,2,3,4,5,6,7,9})
  - An alternative representation uses a binary vector ([1,1,1,1,1,1,1,0,1,0,0,0,0]), having the i–th element equal to 1 if the tree contains the i–th path, and 0 otherwise

In general… –1

- For $I \equiv \mathbb{R}^+$, and for any p, a recursive network exists, for which no collisions occur, with $(k^{p+1}-1)/(k-1)$ state neurons, that calculates

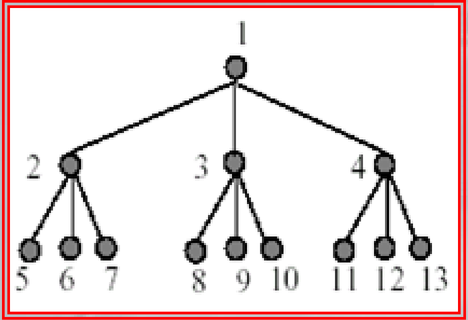
$$\mathbf{X}_s[i] = \begin{cases} U_{v_i} & \textit{if the input tree contains the } i\textit{–th path} \\ 0 & \textit{otherwise} \end{cases}$$

- Proof
  - By construction, each element $a_{i,j}^k$ must be equal to 1 iff the i–th path is composed by the arc between the root and its k–th child together with the j–th path within the k–th subtree, 0 otherwise
  - The entries b_i will be equal to 1 iff the i–th path is empty, i.e. it contains only the root, 0 otherwise (namely only b_1 is equal to 1)

In general… –2

- Example (cont.)

  Posing:



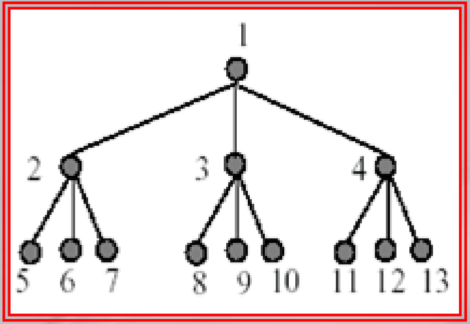
$$\begin{array}{lll} a^1_{2,1}=1, & a^2_{3,1}=1, & a^3_{4,1}=1 \\ a^1_{5,2}=1, & a^2_{8,2}=1, & a^3_{11,2}=1 \\ a^1_{6,3}=1, & a^2_{9,3}=1, & a^3_{12,3}=1 \\ a^1_{7,4}=1, & a^2_{10,4}=1, & a^3_{13,4}=1 \end{array}$$

$$b_1 = 1$$

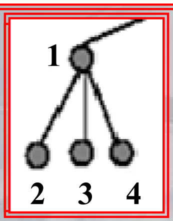
with A_k, k=1,2,3, 13×13 matrices and b vector of length 13, we obtain the "binary" representation for ternary trees of height 2 at most

In general… –3

- Example
  To calculate $a^k_{i,j}$

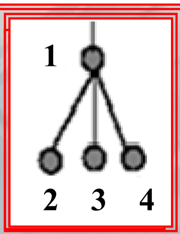


$a^1_{2,1} = 1$



The second path (i=2) connects the root with its first child (k=1) and, w.r.t. the leftmost subtree – renumbering its nodes – the node with label 2 becomes the first one (j=1)

$a^2_{10,4} = 1$



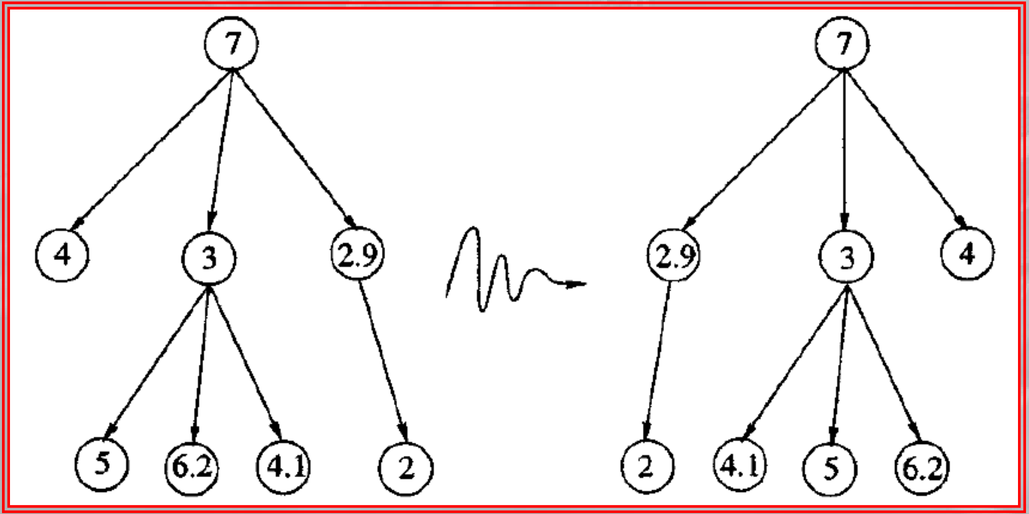
The tenth path (i=10) connects the root with its second child (k= 2) and, renumbering the central subtree, the node with label 10 becomes the fourth one (j=4)

- Choosing an appropriate output function g, such a network can approximate any injective function on trees (with positive real labels), with any degree of accuracy

Recursive models for non positional graphs – 1

- In DAGs (*Directed Acyclic Graphs*) the position of the descendants of each node is not significant
- DAGs can model many different kinds of information, from gene interaction maps to combinational logic blocks in electronic circuit design
- To properly process DAGs using a recursive model, if $G_1 =_{DAG} G_2$ then

$$(g \circ \tilde{f})(G_1) = (g \circ \tilde{f})(G_2)$$



Recursive models for non positional graphs – 2

- Therefore, the recursive model must satisfy the following equation

$$\boxed{f(\mathbf{X}_{v_1},\ldots,\mathbf{X}_{v_k},\mathbf{U}_v,\theta_f)=f(\mathbf{X}_{v_{\pi(1)}},\ldots,\mathbf{X}_{v_{\pi(k)}},\mathbf{U}_v,\theta_f)}$$

  for each permutation $\pi:\{1,\ldots,k\}\rightarrow\{1,\ldots,k\}$ and for any possible set of weigths and labels

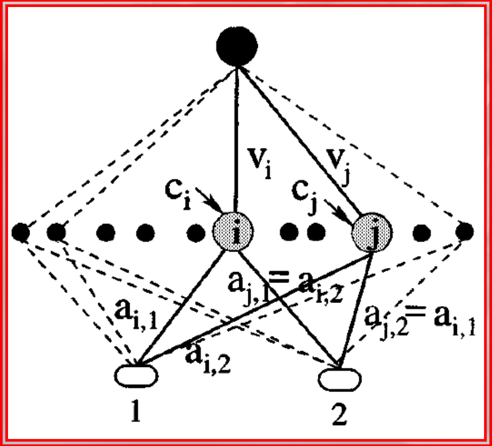
- f is insensitive to any reorganization of the descen-dants of each node
  - Function f can be learnt from examples by a classical recursive model (without constraints)
  - In practice, this training would require a very extens-ive learning set and should be time–prohibitive
  - Vice versa, f can be implemented through a model that naturally realizes (via constraints on the weights) insensitivity to permutations

Recursive models for non positional graphs – 3

- Example
  - If the state transition function is realized by a two–layer neural network, with two inputs, $2q$ hidden neurons and a unique output neuron
  - Let x_1, x_2 be the inputs and let $a_{i,1}$, $a_{i,2}$, c_i, v_i be the network parameters w.r.t. the i–th hidden neuron; for the j–th hidden neuron, let $a_{j,1}=a_{i,2}$, $a_{j,2}=a_{i,1}$, and $c_j = c_i$, $v_j = v_i$ be
  - The contribution to the output due to i and j is:

$$\boxed{h_i(x_1,x_2) = v_i\sigma(a_{i,1}x_1+a_{i,2}x_2+c_i)+v_i\sigma(a_{i,2}x_1+a_{i,1}x_2+c_i)}$$

with $h_i(x_1,x_2)=h_i(x_2,x_1)$



Recursive models for non positional graphs – 4

- Remark
  In the general case of networks with any number of inputs and outputs, the hidden layer must contain q sets of units, each of which consists of a number of neurons equal to all possible permutations on the inputs: $o!$, if o is the maximum outdegree of the structures to be learnt
- The original time complexity is moved to space complexity (a very large architecture, anyway diffi-cult to train)

Recursive models for cyclic graphs – 1

- The general recursive model, in which the state updating is carried out by

$$\mathbf{X}_v = \sigma\left(\sum_{k=1}^{o} \mathbf{A}_k \cdot \mathbf{X}_{\mathrm{ch}_k[v]} + \mathbf{B} \cdot \mathbf{U}_v\right)$$

cannot be used for cyclic structures

- Actually, in this case, it would produce a sort of recursion: the state X_v at node v, involved in a cycle, depends on the same X_v calculated at some previous pseudo–time, given that v is a descendant of itself

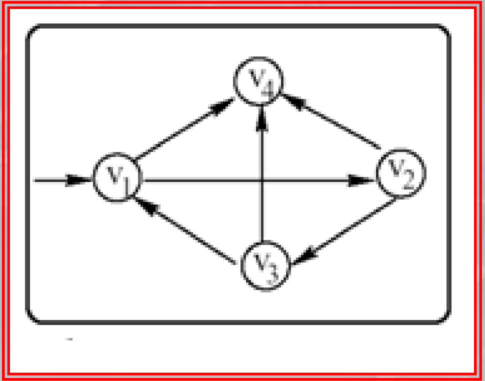
Recursive models for cyclic graphs – 2

- The recursive network represents a dynamic system, whose equilibrium points are the solutions to the state update equation
- How can we face this issue?
  - Collapse the cycle into a single node, summarizing the overall information in its label
- Problem: The operation can be carried out with graph–clustering techniques, but some information will be lost

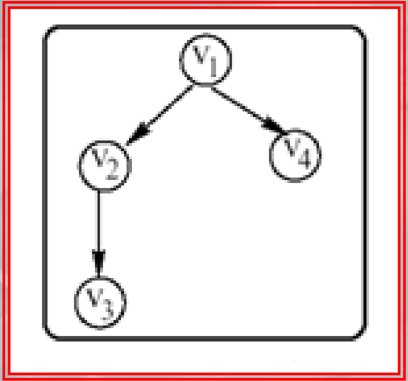
Recursive models for cyclic graphs – 3

- Alternative solution: Let us represent cyclic graphs using *recursive–equivalent trees*
- Let $G=(V,E)$ be a directed, cyclic graph, having s as its supersource; the tree $\boldsymbol{T}_r=(V_r,E_r)$, recursive–equivalent to G, can be constructed based on the following algorithm:
    - Visit G, starting from s, calculating a *spanning tree* $\boldsymbol{T}_c=(V,E_c)$, with root s; initially, let $\boldsymbol{T}_r=\boldsymbol{T}_c$
    - For each edge $(v_1,v_2)\in E\backslash E_c$, a clone of v_2, v_2^{new}, is added in $\boldsymbol{T}_r$, i.e., we update $V_r=V_r\cup v_2^{new}$ and $E_r=E_r\cup(v_1,v_2^{new})$

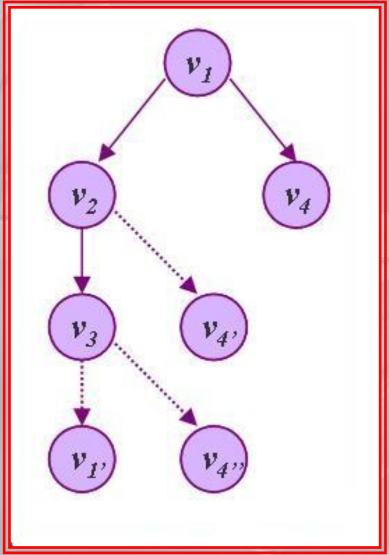
Recursive models for cyclic graphs – 4



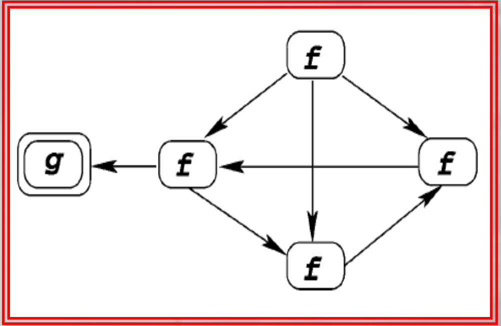
Rooted directed, cyclic graph



Spanning tree



Recursive–equivalent tree



Encoding and output networks

Recursive models for cyclic graphs – 5

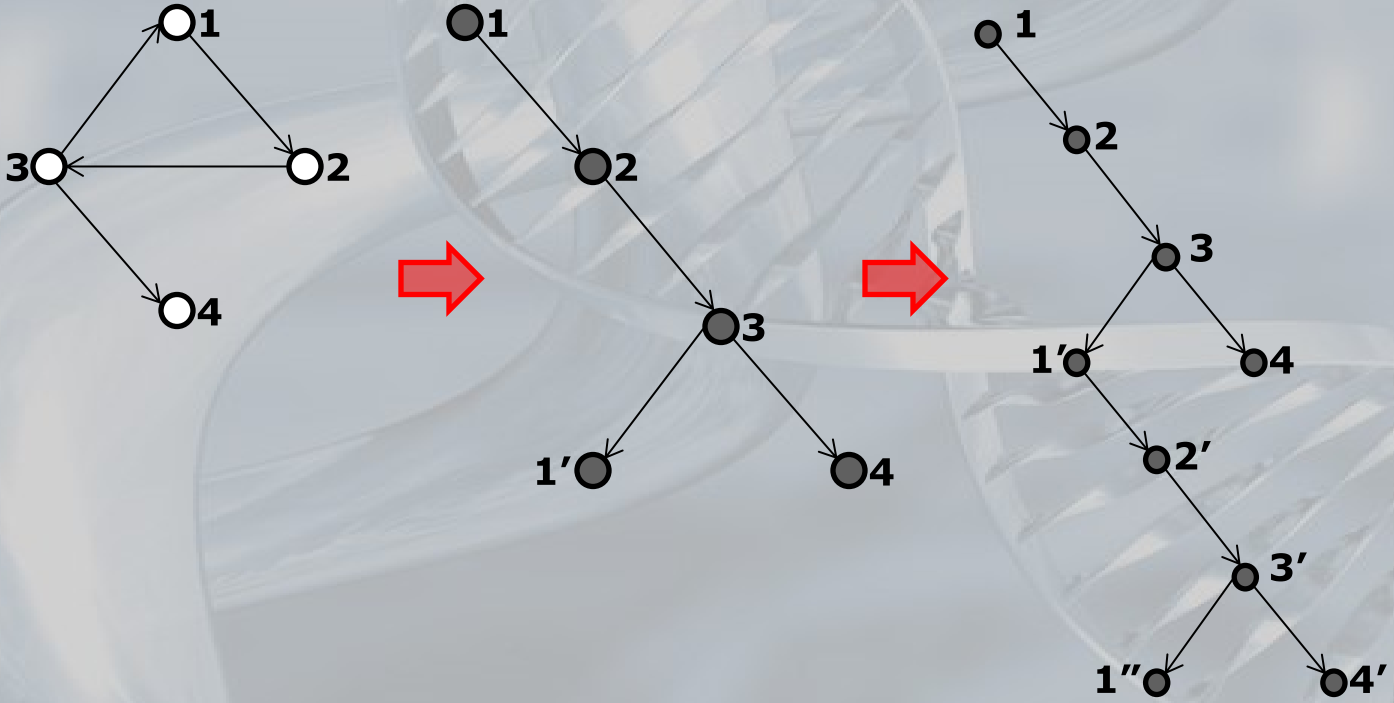
- Theorem 3

  Let $G=(V,E)$ be a directed, positional cyclic graph, having a supersource s; let $\boldsymbol{T}_r=(V_r,E_r)$ be its recursive–equivalent tree, with $|E_r|=|E|$; G can be uniquely reconstructed starting from $\boldsymbol{T}_r$

  - Cyclic graphs can be recursively processed after being preprocessed in order to extract the related recursive–equivalent tree $\boldsymbol{T}_r$
  - Results and model limitations derived for trees can be equivalently applied to cyclic graphs

Recursive models for cyclic graphs – 6

- More generally, each cycle can be unfolded, starting from the supersource, to form tree structures of varying depth, obtained through multiple visits to the nodes of the cycle



Recursive models for cyclic graphs – 7

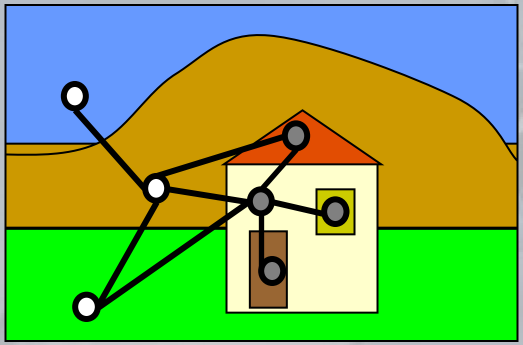
- Remarks
  - Recursive networks can actually be applied to cyclic structures, at least after an appropriate preprocessing
  - The phenomenon of *vanishing errors* for deep backpropagations, in this particular case, en-sures that it is not necessary to unfold the structure “too much” in order to stabilize the learning procedure

A general solution: GNNs – 1

- Aim: Model and learn a function
  $$\varphi_w: \mathcal{G}\times\mathcal{N}\rightarrow\mathbb{R}^n$$
  where $\mathcal{G}$ is a set of graphs, $\mathcal{N}$ represents the set of nodes and R^n is the n–dimensional Euclidean space
- Function φ_w
  - accepts a graph G and a node v as its input and calculates a real vector
  - is a parametric function: w collects its parameters
- Graph Neural Networks can face two main types of problems:
  - *Node–focused*
  - *Graph–focused*

A general solution: GNNs – 2

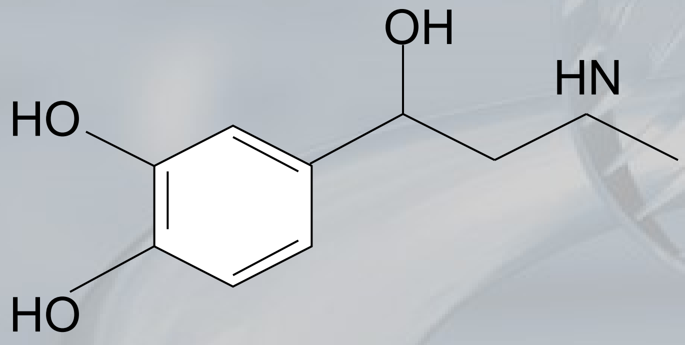
- Example 1
  In node–focused problems, $\varphi_w(G,v)$ depends both on the whole graph and on a particular node



- Images can be represented by Region Adjacency Graphs (RAGs), where:
  - nodes represent homogenous regions and are labelled by visual features (colour, texture, area)
  - edges define adjacency relationships
- Object localization in images is a node–focused problem
  - The network can answer the question:
    Does node v belong to the house?

A general solution: GNNs – 3

- Example 2
  In graph–focused problems $\varphi_w(G)$ depends only on the graph G



- Molecules are represented by indirected and cyclic graphs:
  - nodes represent atoms and small molecules
  - edges stand for chemical bonds

- The network can answer the question:
  Is the chemical compound an active drug against cancer proliferation?

A general solution: GNNs – 4

- Example 3

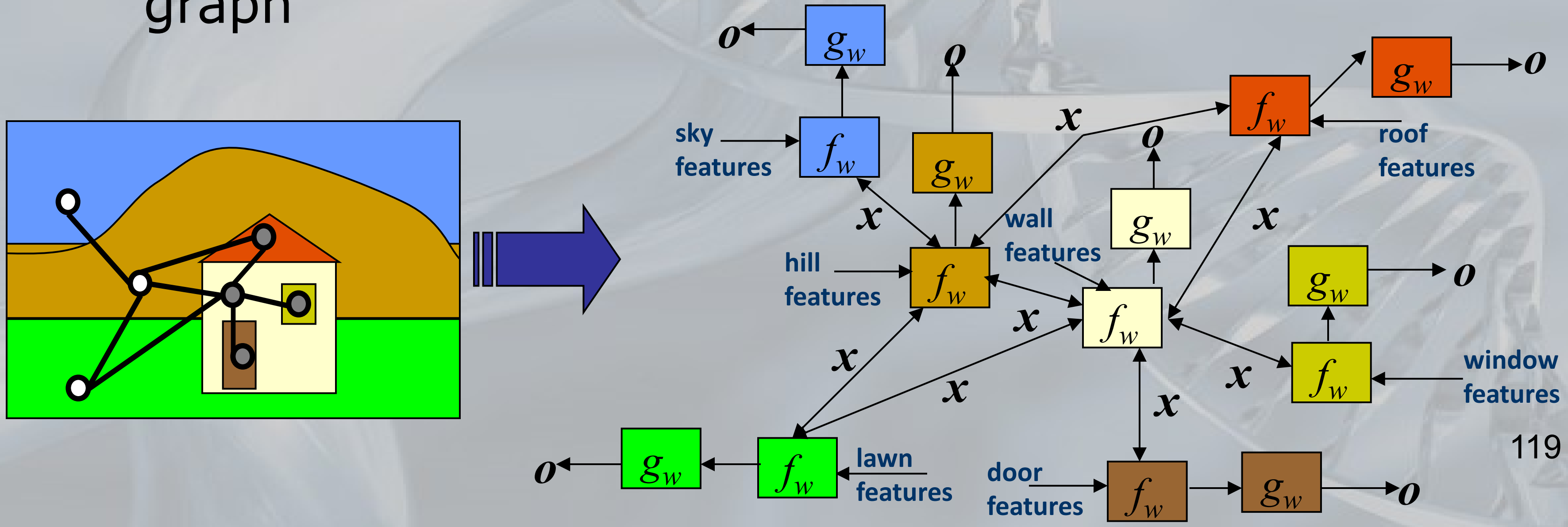
  Actually a third type of problem to be solved on graphs exists, which is called *edge–focused*
- Edge–focused problems concern tasks in which the targets are associated to the edges: the GNN must classify, cluster, or even predict the existence of relationships between patterns
- Predicting the nature of chemical bonds between atoms or small molecules represents an edge–focused task

Graph Neural Networks – 1

- At each node v, a state x_v, and (possibly) the relative output o_v, are calculated
- f_w, g_w are feedforward networks that process local information to v

$$\boxed{x_v = f_w(l_v, x_{ne_1[v]}, \ldots, x_{ne_k[v]}) \quad o_v = g_w(l_v, x_v)}$$

- The GNN shares the same topology of the input graph



Graph Neural Networks – 2

- Note: The state transition function f_w may depend also on the labels of the neighboring nodes of v and on the labels of the edges originating from v (if any)
- In order to ensure a correct processing mode for GNNs, it must be guaranteed that, for the state update equations

$$
\begin{array}{l} x_v = f_w(l_v, x_{ne_1[v]}, \dots, x_{ne_k[v]}) \\ o_v = g_w(l_v, x_v) \end{array}
\quad \Longleftrightarrow \quad
\begin{array}{l} X = F_w(L, X) \\ O = G_w(L, X) \end{array}
$$

  a unique solution exists
- We also need to:
  - provide an efficient method for calculating the states
  - define an efficient learning algorithm for the parameter optimization

Graph Neural Networks – 3

- The choice of the transition function in GNNs is based on the Banach Theorem, to ensure the existence and the uniqueness of the solution
- Actually, the Fixed Point Theorem guarantees that the state update equations admit a unique solution if and only if F_w is a contraction with respect to X

Graph Neural Networks – 4

- Definition
  Let (X,d) be a metric space. Then a map F: $X \rightarrow X$ is called a contraction mapping on X if there exists $q\in[0, 1)$ such that
  $$d(F(x),F(y))\leq qd(x,y)$$
  for all $x, y\in X$
- Banach Fixed Point Theorem
  Let (X,d) be a non–empty complete metric space with a contraction mapping F: $X \rightarrow X$; then F admits a unique fixed–point x^* in X (i.e. $F(x^*) = x^*$); furthermore, x^* can be found as follows: start with an arbitrary element x_0 in X and define a sequence $\{x_n\}$ by $x_n = F(x_{n-1})$, then $x_n \rightarrow x^*$

GNNs: State calculation – 1

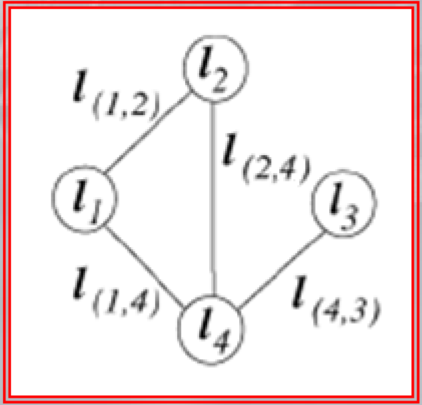
- The states of all the nodes are initialized with a default value; they are iteratively updated until reaching an equilibrium point
- The Banach Theorem ensures the convergence of the iterative procedure with exponential speed (and regardless of the initial state value)

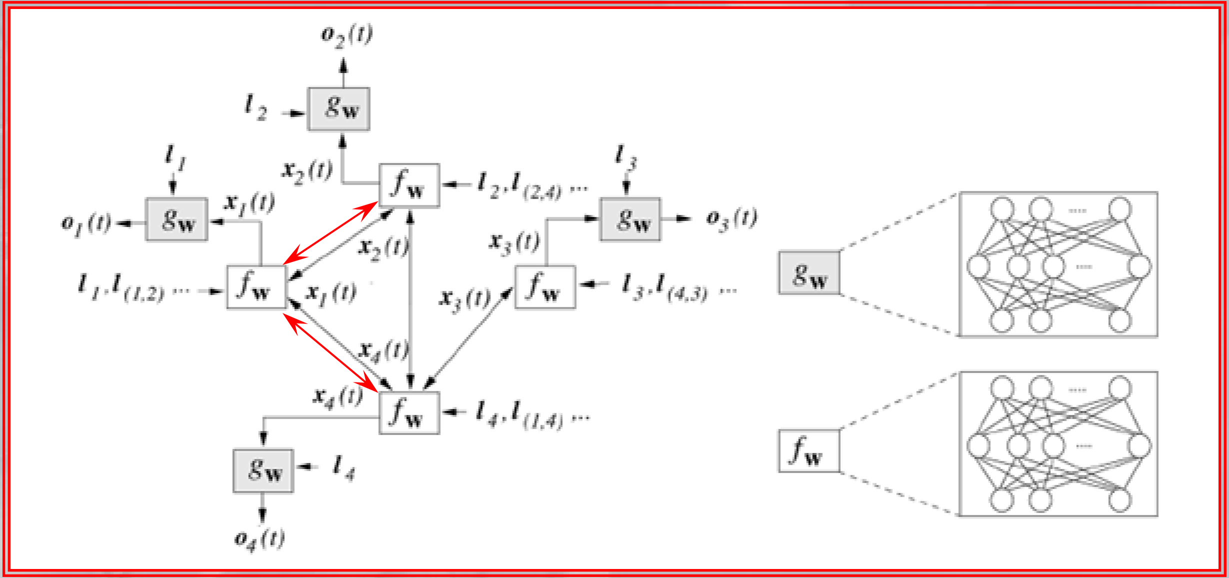
$$x_v(t+1) = f_w(l_v, x_{ne_1[v]}(t), .., x_{ne_k[v]}(t))$$
$$o_v(t+1) = g_w(l_v, x_v(t+1))$$

$$\Longleftrightarrow$$

$$X(t+1) = F(L, X(t))$$
$$O(t+1) = G(L, X(t+1))$$

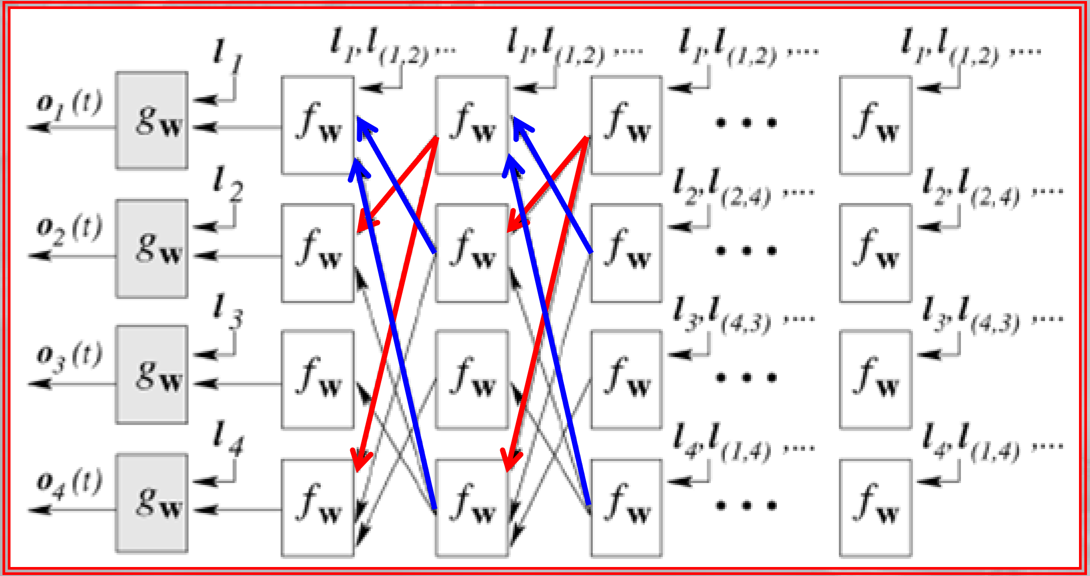
GNNs: State calculation – 2





When the state transition and the output functions are implemented by MLPs, the encoding network is a recursive network

In the unfolding network, each layer corresponds to a time instant and contains a copy of all the units of the encoding network; connections between layers depend on the encoding network connectivity



GNNs: State calculation – 3

- In detail:
  - Each iteration produces a "synchronous activation" of the encoding network
  - …which corresponds to an iteration of the Jacobi algorithm for the solution of nonlinear systems
  - The Jacobi algorithm is easily adapted to large systems (millions of equations) and it is also used for the calculation of the Google PageRank

$$\boxed{\begin{array}{rcl}
f^1(x_1^{k+1}, x_2^k, \dots x_{n-1}^k, x_n^k) & = & 0 \\
f^2(x_1^k, x_2^{k+1}, \dots x_{n-1}^k, x_n^k) & = & 0 \\
 & . & \\
 & . & \\
 & . & \\
f^n(x_1^k, x_2^k, \dots x_{n-1}^k, x_n^{k+1}) & = & 0
\end{array}}$$

GNNs: The learning algorithm

- A gradient–descent strategy is used in order to minimize the error function

$$\boxed{e_w = \sum_{i,k} (t_{i,k} - \varphi_w(\mathbf{G}_k, v_{i,k}))^2}$$

- Learning proceeds through the repetition of the steps…
  - …for updating the states $x_n(t)$ until convergence is reached
  - ...for calculating the gradient $\frac{\partial e_w}{\partial w}$, based on updated states and weights
- The gradient calculation is performed by combining the Almeida–Pineda algorithm (a particular version of BPTT) and BPTS

GNNs: Universal approximation

- So as recursive models (in terms of their particular scope), GNNs are (*almost*) universal approximators, i.e., they can approximate in probability, and up to any degree of precision, all the "practically useful" functions on the graph space

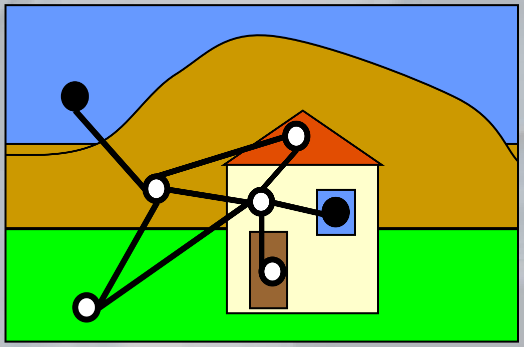
Long–term dependencies in GNNs – 1

- Practical difficulties have been reported in training dynamical networks to perform tasks in which spatio–temporal contingencies present in the input structures span long intervals
- In other words… gradient based learning al-gorithms face an increasingly difficult problem as the duration of the dependencies to be captured increases
  - There is a trade–off between efficient learning by gradient descent and latching of information for long “periods”

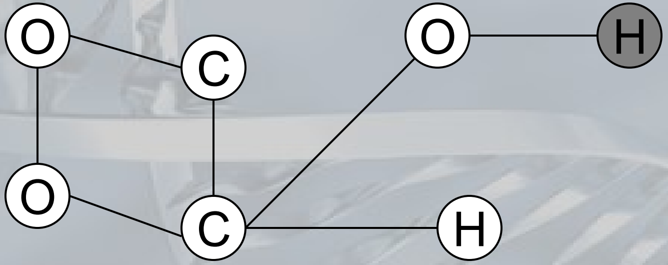
Long–term dependencies in GNNs – 2

- In GNNs, the long–term dependency problem is observed when the output on a node depends on far nodes (i.e. neurons connected by long paths)

Localize the objects having the same color



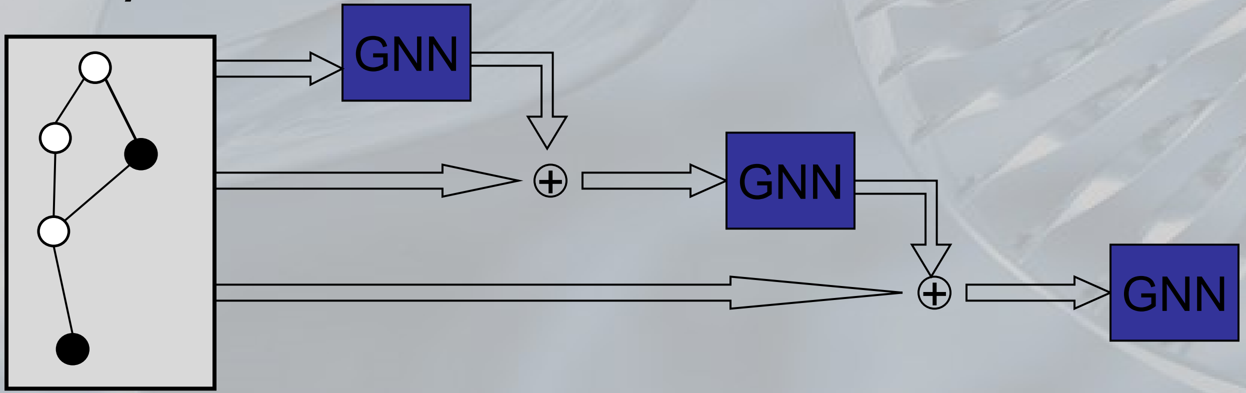
Is this molecule mutagenic?



A mutagenic compound is properly an “agent” capable of inducing mutations in a single gene, chromosome or genome

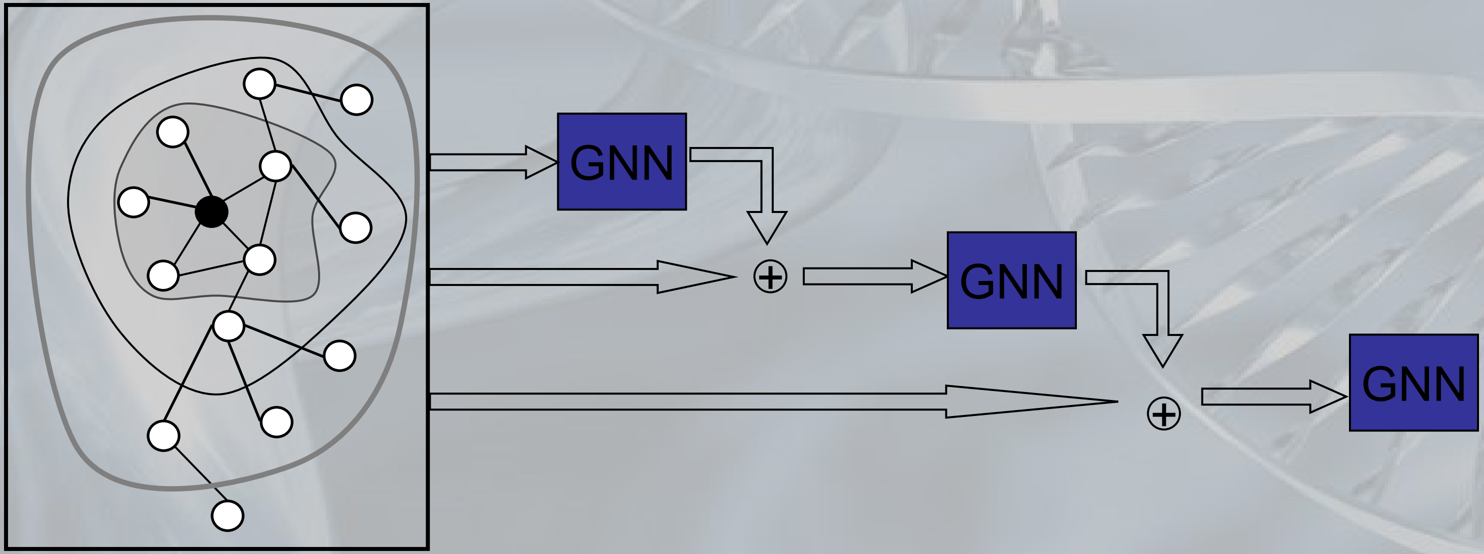
Layered GNNs

- GNNs can be *cascaded*
- In each layer, the $(i+1)$–th GNN takes a graph in input
    - with the same connectivity of the original input graph
    - with node labels “enriched” by the information produced at the previous layer, for instance:
        - the output(s) of the i–th GNN
        - the state(s) of the i–th GNN
        - both of them
- Intuitively… each GNN solves the original problem, but it can make use of the expertise acquired in the previous layers



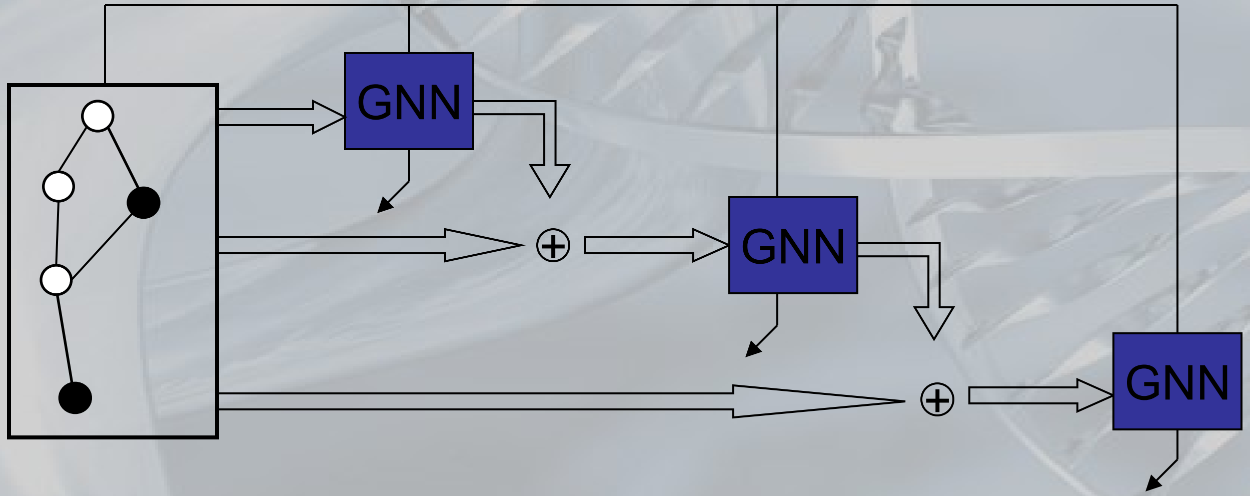
Can layered GNNs help with long–term dependencies?

- LGNNs can incrementally incorporate the dependencies into the labels
  - The output of a given node can collect information extracted from its neighborhood
  - At each layer, the label contains information about a larger neighborhood
- From a different point of view, the $(i+1)$–th GNN can concentrate its efforts only on those patterns incorrectly classified by the previous GNNs



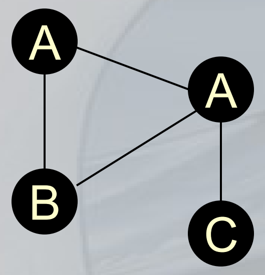
Training layered GNNs

- Training LGNNs using a BP–like algorithm would reintroduce long–term dependencies
- Other solutions?
  - The training phase is carried out layer by layer
  - Each layer is trained using the original target

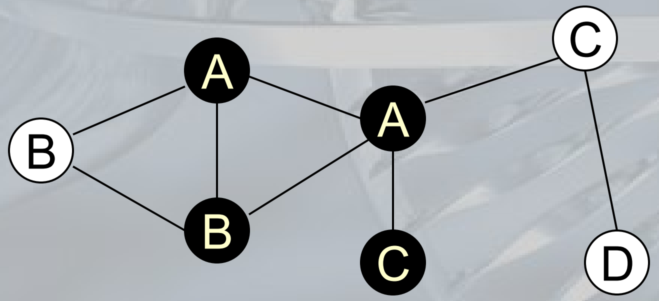


Experiments on four datasets – 1

- Subgraph localization (artificial dataset)
  - The GNN takes in input a graph G and, on each node v, returns –1/1 according to whether v belongs or not to a particular subgraph S
  - The subgraph S in unknown: it should be learnt by examples
  - The dataset contains 1000 random graphs, 15 nodes each; the subgraphs are constituted by 7 nodes



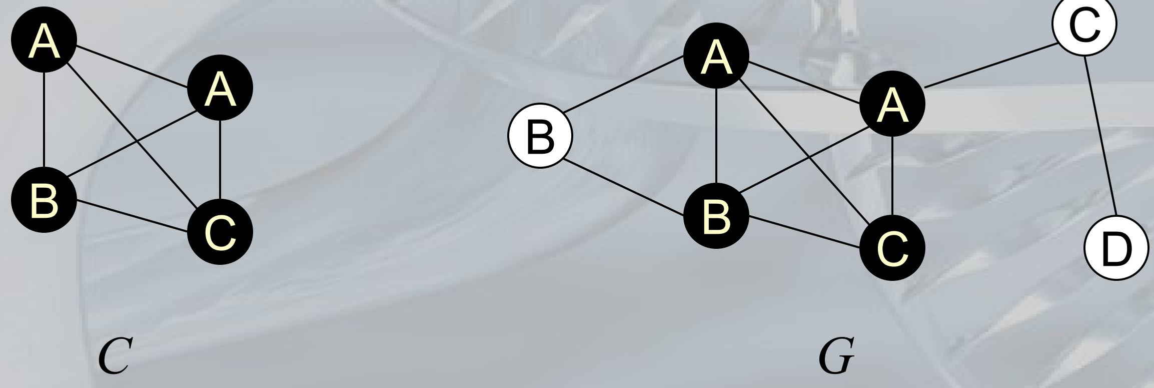
S



G

Experiments on four datasets – 2

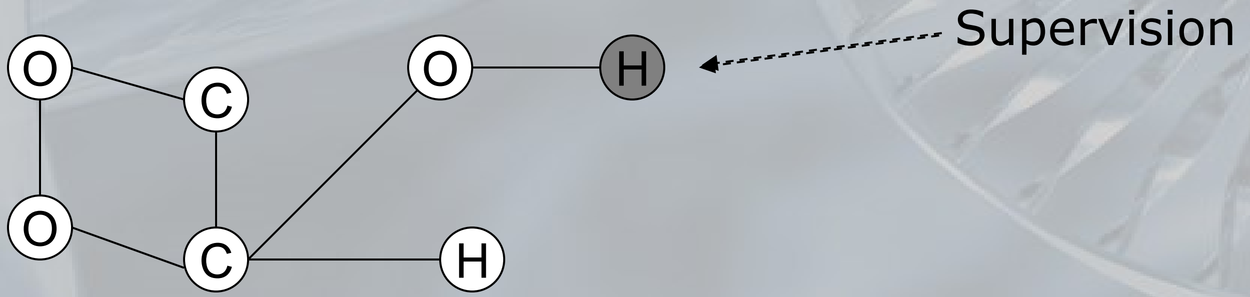
- Clique localization (artificial dataset)
  - Just the same problem, except that all the cliques (fully connected graphs) of a given size must be localized
  - The dataset contains 1000 random graphs, 15 nodes each; the cliques are constituted by 5 nodes



Experiments on four datasets – 3

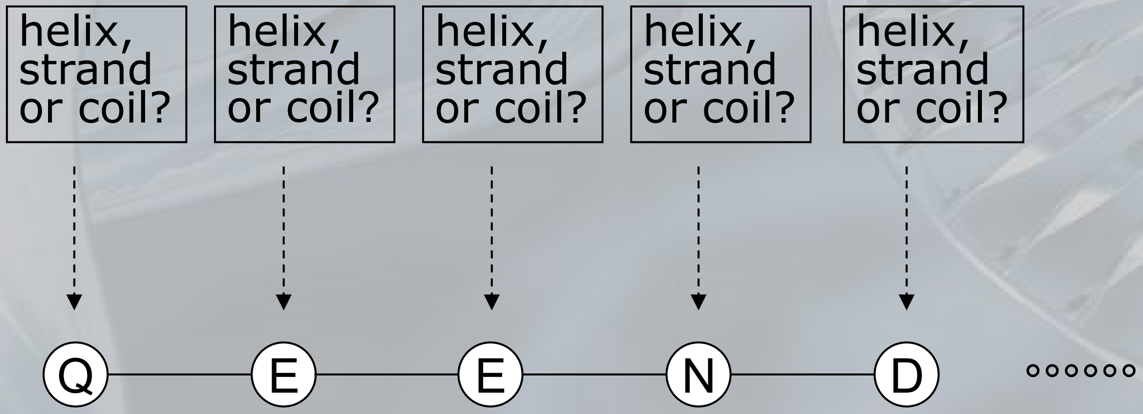
- Classification of mutagenic molecules (publicly available dataset)
  - The goal is that of predicting whether a molecule is mutagenic
  - The molecule is represented by a graph where nodes stand for atoms, and edges denote chemical bonds
  - Node/edge labels collect properties of atoms/bonds
  - A unique supervised node, since mutagenicity is a property of the whole molecule
  - The dataset contains 230 molecules

Is this molecule mutagenic?



Experiments on four datasets – 4

- Prediction of the secondary structure of proteins (publicly available dataset)
  - The goal is that of predicting, for each amino acid, if it belongs to a particular secondary structure (α–helix, β–sheet or random coil)
  - The protein is represented by its primary structure (a sequence of amino acids)
  - Node labels contain amino acid features
  - The dataset contains 2171 proteins, constituted by 344653 amino acids

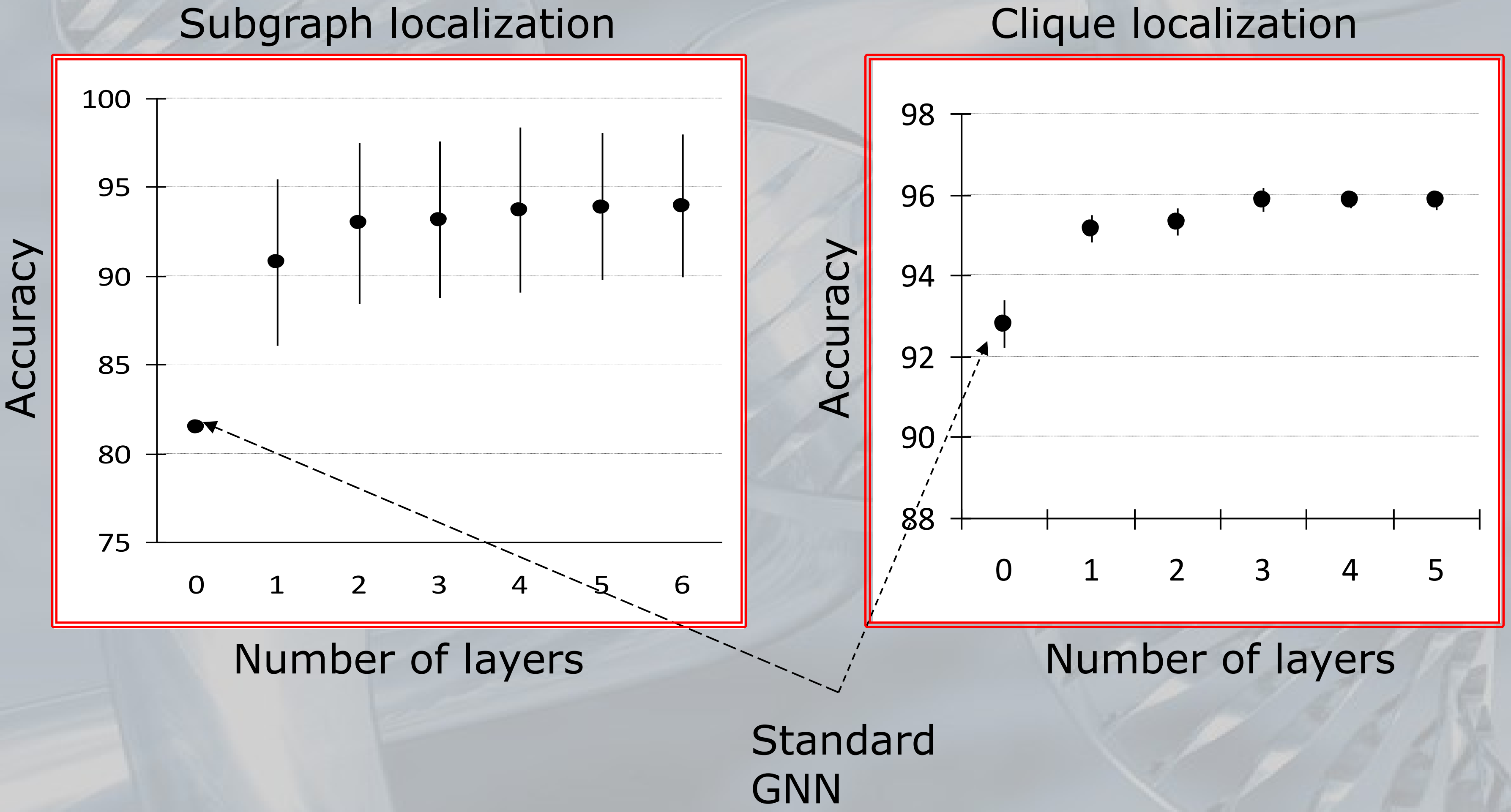


Accuracy results

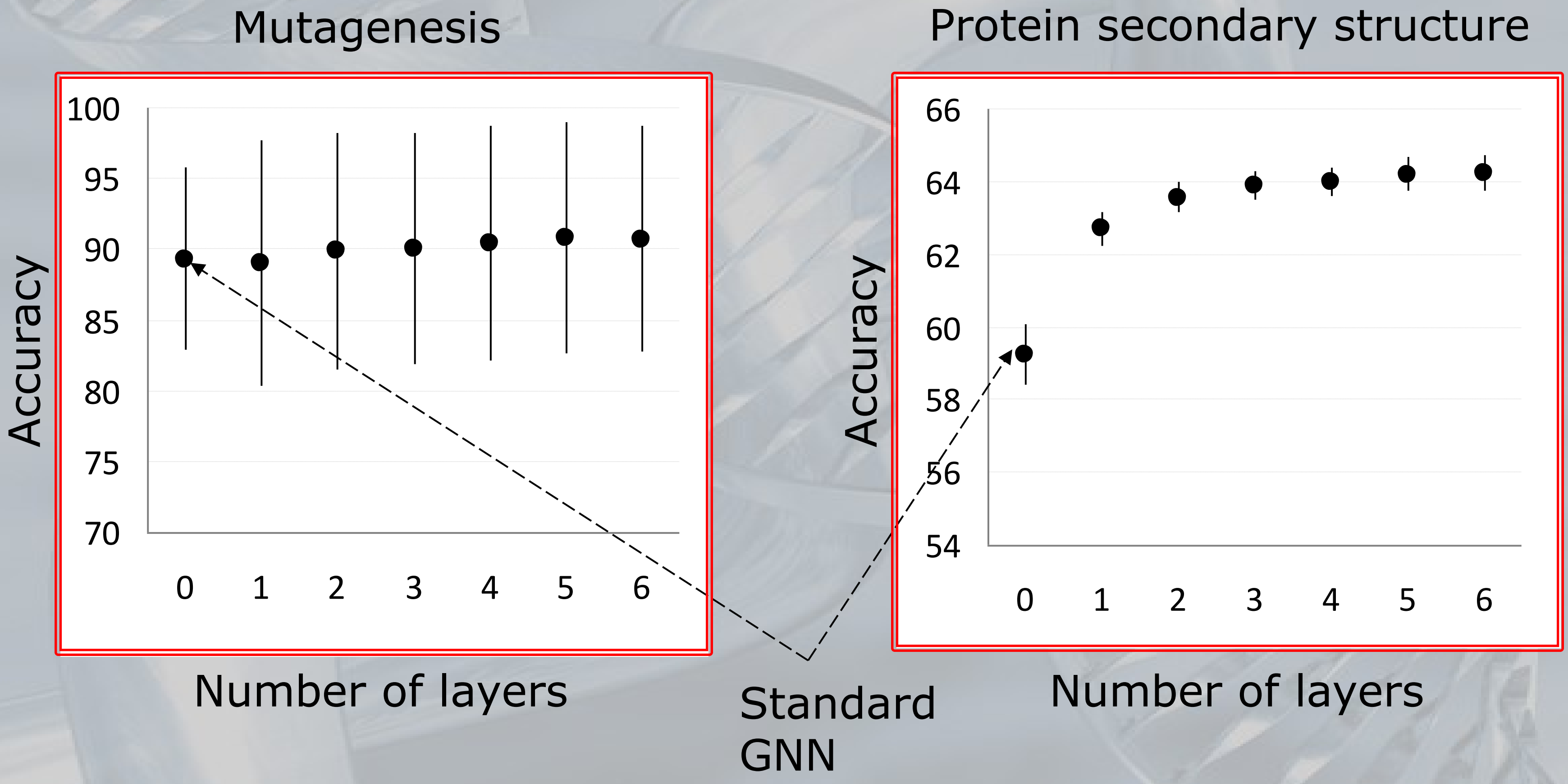
- Comparative results on average accuracy over 5 runs

Dataset	Standard GNN	Layered GNN
Subgraph localization	81.4%	93.9%
Clique localization	88.2%	96.0%
Mutagenesis	89.5%	92.2%
Protein secondary structure	60.8%	64.2%

Number of layers – 1

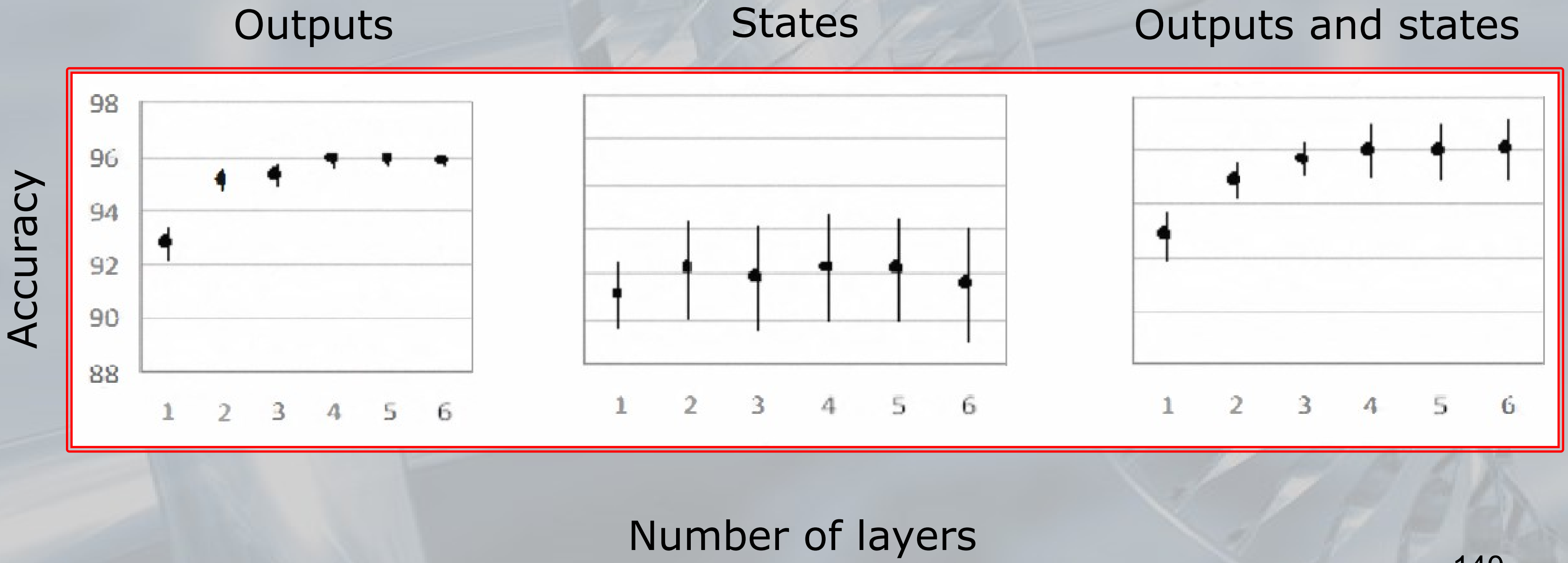


Number of layers – 2



Adjoint label content

- Comparative accuracy and standard deviation, varying both the type of adjoint label information and the number of layers, for the clique localization problem


Outputs
States
Outputs and states
Accuracy
Number of layers