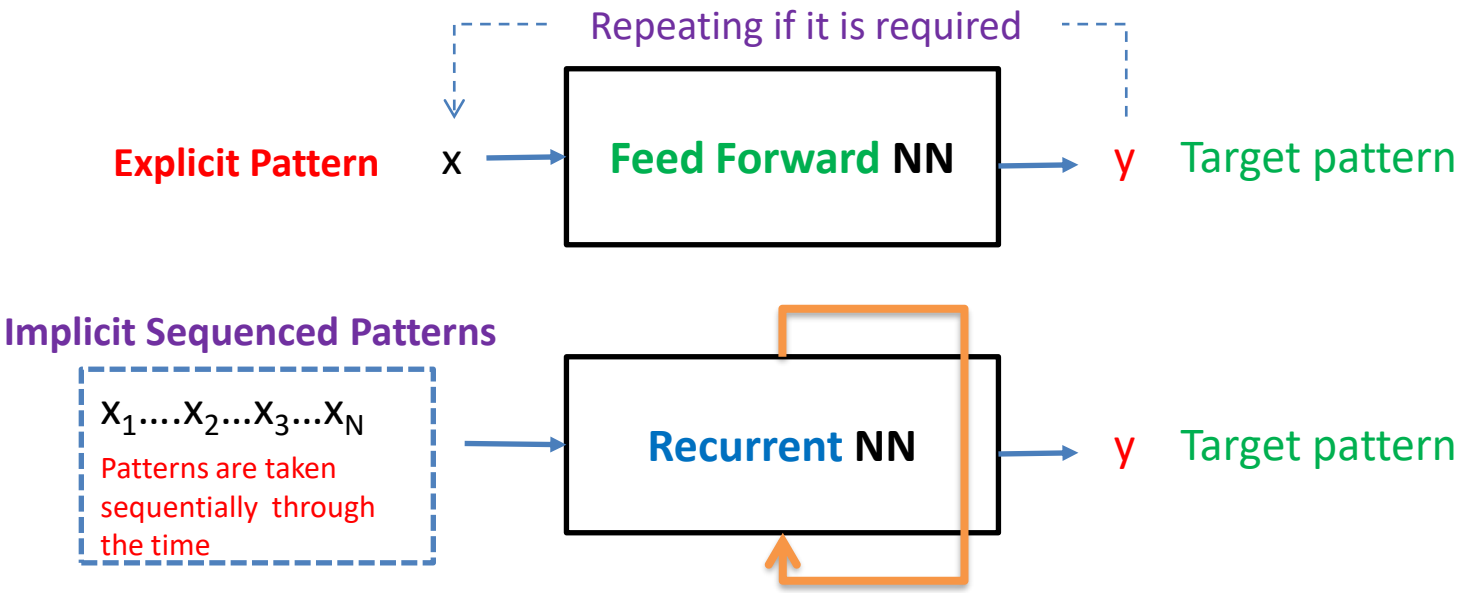
Chapter 3

Memory Neural Networks

Memory Neural networks(MNNs):

NNs which associate an explicit input pattern or Implicit sequenced patterns to a target pattern (Supervised Learning)

* such NNs make memories for pattern association.



Applications of MNNs

Natural NNs have high capability in pattern associations.

Inspiring from natural neural networks, MNNs have been utilized in:

(1) Identification

- To associate biometric signals (face/finger print/IRIS/hand geometry/body…) to man/animal/plant.
- To associate signature, handwriting or speech to certain people.

(3) Character recognition(OCR)

(5) Translation machines

(6) Prediction

1. To predict samples of temporal signal : finance sys./energy sys./natural sys,…
2. To predict words of a text in a writing process.
3. To predict a part of a speech.
4. To predict frames of a movie.

(7) Modeling walking and moving of different people/animals/phenomena

(8) Learning in Social Robots to path planning and interact with environment.

(9) Image Captioning ,Video or Image Descriptions

(10) ….

Challenges in learning of MNNs

MNNs with explicit input patterns

1. Each explicit input pattern may be deformed, or partially presented.
2. Each explicit input pattern may disturbed with noise.

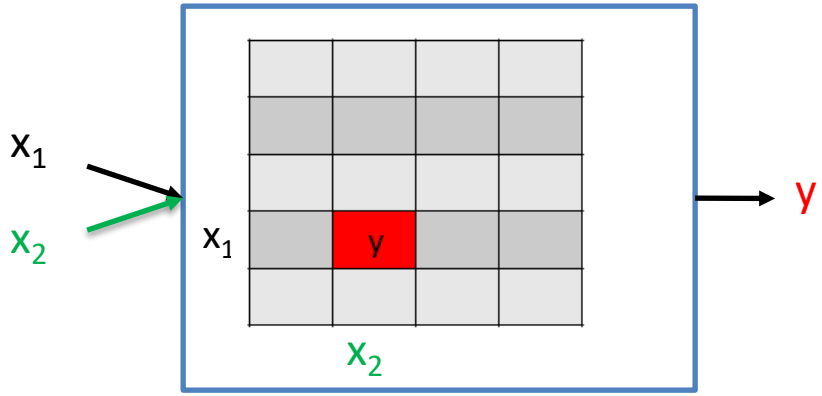
MNNs with implicit sequenced patterns

1. The implicit sequenced patterns may be presented with different lengths.
2. The implicit sequenced patterns may be combined with disturbance and non-relevant patterns.
3. Among the sequenced patterns, it may be short and long dependencies.

* MNNs make robust memories against distortion, disturbances and dimension variations of input patterns.

Interpretation 1: A MNN is like a "look-up table"

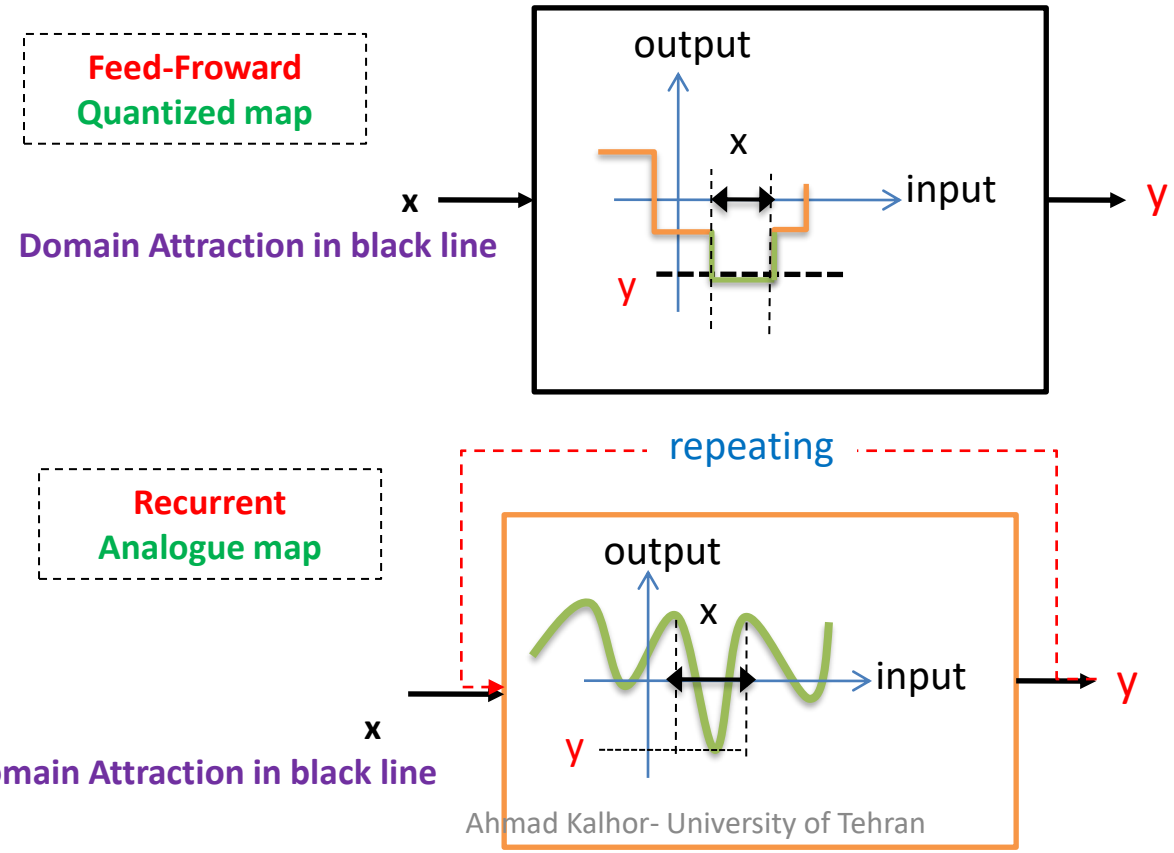
(1) A MNN acts like a "look-up table" where by applying the indices of a cell as the input pattern, the content of the cell is come out as the output pattern.



* Actually, in such look-up tables the indices are not exactly known, they may deformed, disturbed or partially presented or even the number of indices may be changed.

Interpretation 2: A MNN is like a mapping network

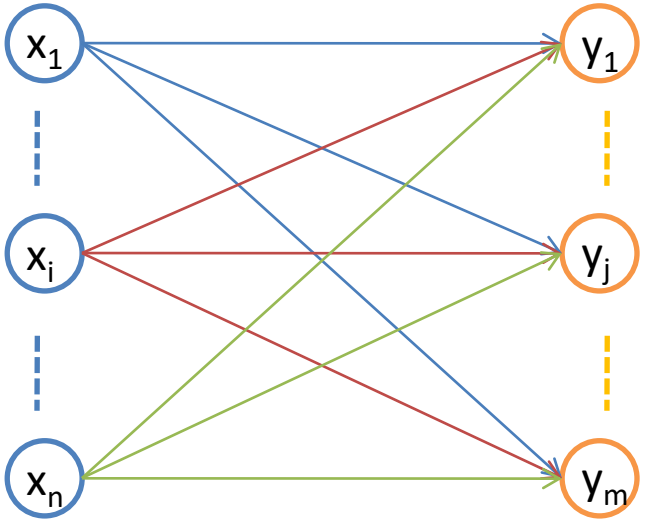
(1) A MNN makes a map in which, each output pattern is represented by a quantized level or a local extremum point and the corresponding input pattern with all possible changes makes its domain of attraction.



2.1 MNNs with explicit input patterns

- The input pattern has a known dimension and it is considered as the main input of the network. **(Quantized map)**
- Simple Feed-Forward MNN
  1. One layer Hetro-Associative and Auto-Associative Networks
  2. Heb Learning method for simple Non-Iterative MNN
- Simple Recurrent MNNs
  1. Auto-Associator With Threshold Function **(Iterative Hetro-Associative Network)**
  2. Hopfield Network **(Iterative Associative Network)**
  3. Bidirectional MNNs **(Iterative Hetro-Associative Network)**
- **MNNs by using Classifiers: MLP, DBNNs and CNNs**

One layer Feed-Forward Network for pattern association



Input pattern: $x^T = [x_1 \dots x_i \dots . x_n]$

Output pattern: $y^T = [y_1 \dots y_j \dots . y_m]$

Weight Matrix $W = [w_{ij}] = \begin{bmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nm} \end{bmatrix}$

Activation Function: **Sign Function**

Output Patterns: **Bipolar/Binary**

$$y^T = f(x^T W)$$

Auto-Associative Memory : the output patterns and the input patterns are the same in the learning phase. However, in the real cases, the input pattern is deformed or disturbed with noise.

Hetro-Associative Memory : the output patterns and the input patterns are not the same in learning phase but in real cases the input patterns are deformed or disturbed

Aim: the weight matrix W is learned in order that for each pair of input-output Patterns: (s(p),t(p)), (p=1..P), y =f(s(p)^{T}W)→t(p)

Pattern Association Algorithm by Hebbian Learning Rule

Step1: Initialize the weight matrix: $w_{ij}=0$ (i=1..n, j=1…m)

Step2: For each input-output pair ({s(p),t(p)}, p=1,2,….,P) do following steps: $s^T=[s_1 \ldots s_i \ldots s_n]$ $t^T=[t_1 \ldots . t_j \ldots . t_m]$

Step3: For i=1…n $s_i \rightarrow x_i$

Step4: For j=1…m $t_j \rightarrow y_j$

Step5: $w_{ij}^{+} := w_{ij}^{-} + x_i y_j$

End

***It can be shown that the weight matrix is sum of the outer-products of input –output patterns:**

$$W = \sum_{p=1}^{P} s(p)t(p)^T$$

Sufficient Conditions for Pattern Association by Hebbian learning rule

1. If all input patterns are orthogonal together then their target patterns will be associated by the network :

$$\forall p_1, p_2 \in \{1,..,P\} and\ (p_1 \neq p_2)\ \textbf{if}\quad \boldsymbol{s}(p_1) \perp \boldsymbol{s}(p_2)\ \textbf{then}\quad \boldsymbol{y} = f(\boldsymbol{s}(p)^T \mathrm{W}) = \boldsymbol{t}(p)$$

$$\boldsymbol{s}(p_1) \perp \boldsymbol{s}(p_2) \equiv \boldsymbol{s}(p_1)^T \boldsymbol{s}(p_2) = 0$$

2. If (1) each pair of input patterns, which have different target patterns, are orthogonal together and (2) inner product of each pair of input patterns, which have the same target patterns, is positive, then their target patterns will be associated by the network :

$$\text{If } \forall p_1, p_2 \in \{1,..,P\} \quad (1)\ \ for\ \ \boldsymbol{t}(p_1) \neq \boldsymbol{t}(p_2)\ if\ \ \boldsymbol{s}(p_1) \perp \boldsymbol{s}(p_2)$$
$$(2)\ \ for\ \ \boldsymbol{t}(p_1) = \boldsymbol{t}(p_2)\ if\ \ \boldsymbol{s}(p_1)^T \boldsymbol{s}(p_2) \geq 0$$

$$\text{then } y^T = \boldsymbol{f}(\boldsymbol{s}(\boldsymbol{p})^T \mathrm{W}) = t^T(p)$$

Input Pattern with disturbance

- If the input pattern s(p) is added by a disturbance

$$\boldsymbol{x} = \boldsymbol{s}(p) + \boldsymbol{\delta} \qquad \boldsymbol{\delta}^T = [\delta_1 \dots \delta_n]$$

- Then the output pattern is computed as follows:

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}^{\mathrm{T}}\mathrm{W})$$

- where

$$\boldsymbol{x}^{\mathrm{T}}\mathrm{W} = \boldsymbol{s}(p)^{\mathrm{T}}\mathrm{W} + \underbrace{\delta^{\mathrm{T}}\mathrm{W}}_{\boldsymbol{d}} = \|\boldsymbol{s}(p)\|^2\boldsymbol{t}(p)^T + \boldsymbol{d} \qquad \boldsymbol{d} = [d_1 \dots d_m]$$

- To associate true output patter it is enough:

$$\boldsymbol{y_j} = \boldsymbol{f}(\|\boldsymbol{s}(p)\|^2 t_j(p) + d_j)$$

$$\forall j,p \quad sign\big(\|\boldsymbol{s}(p)\|^2 t_j(\mathrm{p}) + d_j\big) = sign\big(\|\boldsymbol{s}(p)\|^2 t_j(\mathrm{p})\big)$$

- **A sufficient condition for output-disturbance can be computed as follows:**

$$\max_j |d_j| < \min_p \|\boldsymbol{s}(p)\|^2$$

- A **sufficient condition for input-disturbance can be computed as follows** :

$$\|\delta\| < \bar{\delta} = \frac{\min_p \|\boldsymbol{s}(p)\|^2}{\left\|\sum_{p=1}^{P} |s(p)|\right\|}$$

- As a result:

$$\forall \mathrm{p}_1 \quad \text{if} \quad \|\mathrm{x} - \mathrm{s}(\mathrm{p}_1)\| < \bar{\delta} \quad then \quad f(x^T W) = t(p_1)$$

Example(1)

P=4

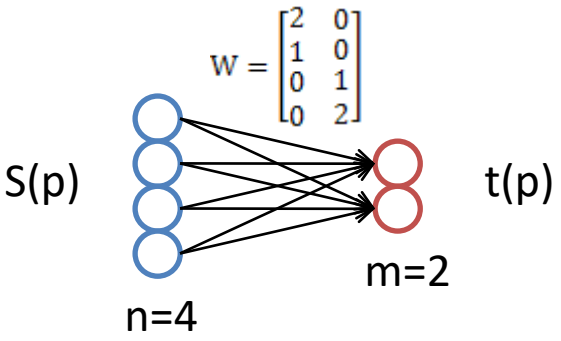
$S(1)^T=[1\ 0\ 0\ 0] \rightarrow t(1)^T=[1\ 0]$

$S(2)^T=[1\ 1\ 0\ 0] \rightarrow t(2)^T=[1\ 0]$

$S(3)^T=[0\ 0\ 0\ 1] \rightarrow t(3)^T=[0\ 1]$

$S(4)^T=[0\ 0\ 1\ 1] \rightarrow t(4)^T=[0\ 1]$

$W=s(1)\ t(1)^T+s(2)\ t(2)^T+s(3)\ t(3)^T+s(4)\ t(4)^T$



$$f(s(1)^T W) = f\left([1\ 0\ 0\ 0]\begin{bmatrix}2 & 0\\ 1 & 0\\ 0 & 1\\ 0 & 2\end{bmatrix}\right) = f([2\ 0]) = [1\ \ 0] = t(1)$$

$$f(s(2)^T W) = f\left([1\ 1\ 0\ 0]\begin{bmatrix}2 & 0\\ 1 & 0\\ 0 & 1\\ 0 & 2\end{bmatrix}\right) = f([3\ 0]) = [1\ \ 0] = t(2)$$

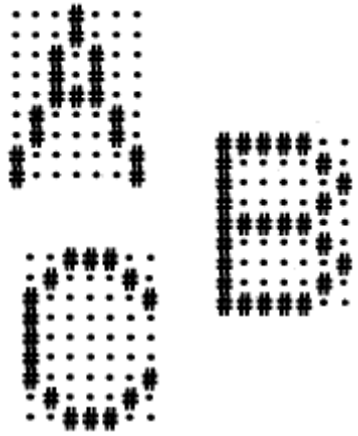
$$f(s(3)^T W) = f\left([0\ 0\ 0\ 1]\begin{bmatrix}2 & 0\\ 1 & 0\\ 0 & 1\\ 0 & 2\end{bmatrix}\right) = f([0\ 2]) = [0\ \ 1] = t(3)$$

$$f(s(4)^T W) = f\left([0\ 0\ 1\ 1]\begin{bmatrix}2 & 0\\ 1 & 0\\ 0 & 1\\ 0 & 2\end{bmatrix}\right) = f([0\ 3]) = [0\ \ 1] = t(4)$$

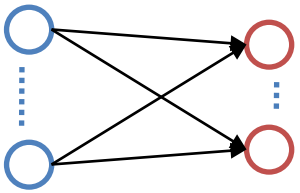
$$f(x) = \begin{cases}1, & x > 0\\ 0, & x \le 0\end{cases}$$

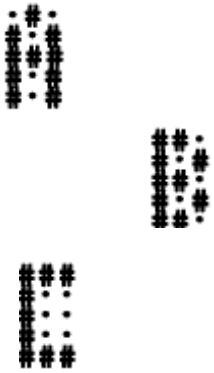
Example(2)

character Recognition



$$f(x) = \begin{cases} 1, & x > 0 \\ -1, & x \le 0 \end{cases}$$





$S(A)^T=[-1\ -1\ -1\ 1\ -1 \ldots .-1\text{-}1\ 1]_{63*1}$ ⟶ $t(A)^T=[-1\ 1\ -1\ \ldots .1\text{-}1\ 1]_{15*1}$

$S(B)$ ⟶ $t(B)$

$S(C)$ ⟶ $t(C)$

$$W = S(A)t(A)^T + S(B)t(B)^T + S(C)t(C)^T$$

It works for three characters

$$f\left(s(A)^T W\right) = t(A)$$

$$f\left(s(B)^T W\right) = t(B)$$

$$f\left(s(C)^T W\right) = t(C)$$

Network Evaluation with disturbed input patterns

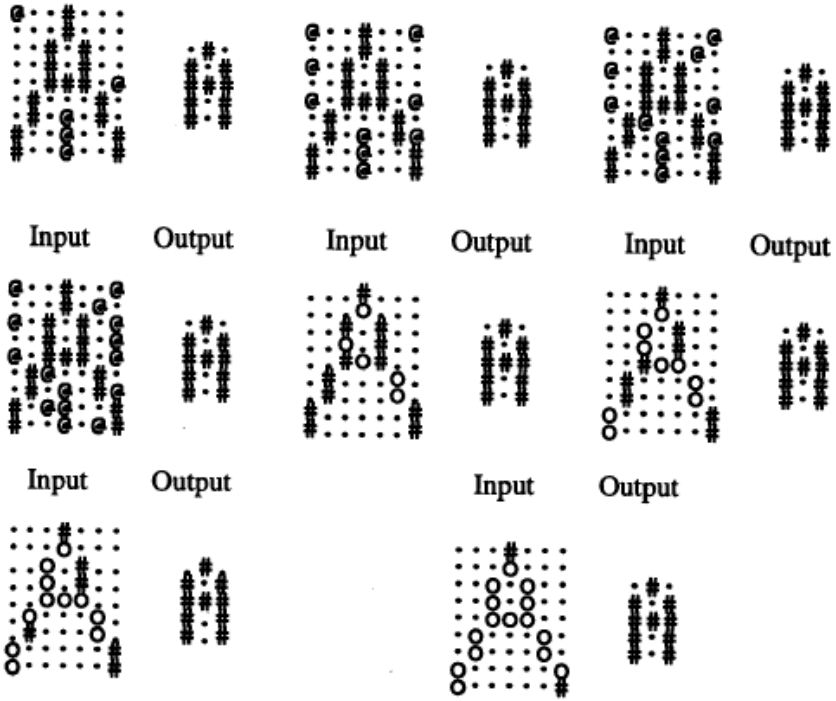


Figure 3.4 Response of heteroassociative net to several noisy versions of pattern A.

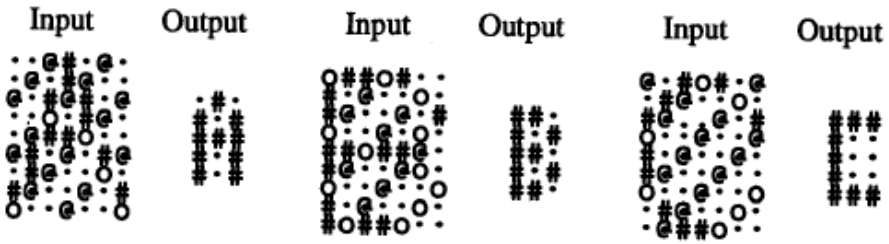
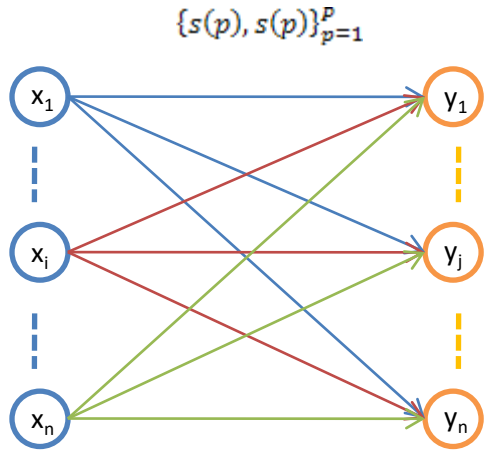


Figure 3.5 Response of heteroassociative net to patterns A, B, and C with mistakes in 1/3 of the components.

Auto-Associative Memory Network



Input pattern: $x^T = [x_1 \dots x_i \dots x_n]$

Output pattern: $y^T = [y_1 .. y_j \dots y_n]$

Weight Matrix $W = [w_{ij}]_{n \times n}$

Activation Function: **Sign Function**

Output Patterns: **Bipolar/Binary**

$$y^T = f(x^T W)$$

• Such Networks are similar to Auto Encoders but dimensionality reduction is not the main purpose. Here, retrieving an original pattern from its disturbed or deformed form is the main goal.

Modified Hebbian learning rule for Auto-associative Networks

- Hebbian learning rule

$$W = \sum_{p=1}^{P} s(p)s(p)^T$$

Such weigh matrix (p.s.d (positive semi definite) and symmetric) causes that a self-associative network can make memory for "p " orthogonal binary or bipolar patterns:

$$\forall p_1 \neq p_2 \in \{1,..,P\} \;\; if \;\; \boldsymbol{s}(p_1) \perp \boldsymbol{s}(p_2) \quad \text{then } y^T = \boldsymbol{f}(\boldsymbol{s}(\boldsymbol{p})^T W) = s^T(p) \qquad \{s(p)\}_{p=1}^{P}$$

If "P" is increased then it can be shown that for bipolar patterns the weigh matrix becomes nearer to an Identify matrix. Hence, the capability of the network in associating the disturbed or deformed memorized patterns to their original forms is reduced. The domain of attraction about a certain pattern will be recued.

- The modified Hebbian learning rule

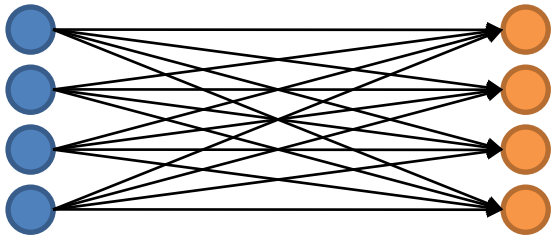
$$W = \sum_{p=1}^{P} s(p)s(p)^T - PI_n$$

It can be shown that the modified Heb learning method can still associate the disturbed or deformed orthogonal patterns to their original ones.

Example1:

Store the bipolar pattern s^T=[1 1 1 -1]

Activation function is bipolar "sign" function.

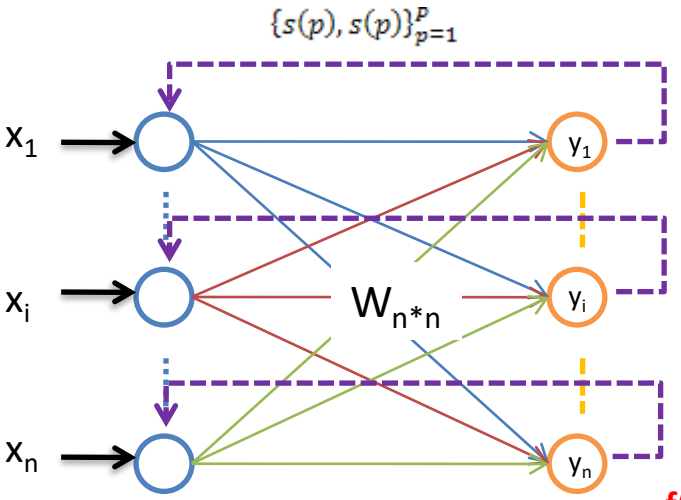


$$W = SS^T = \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

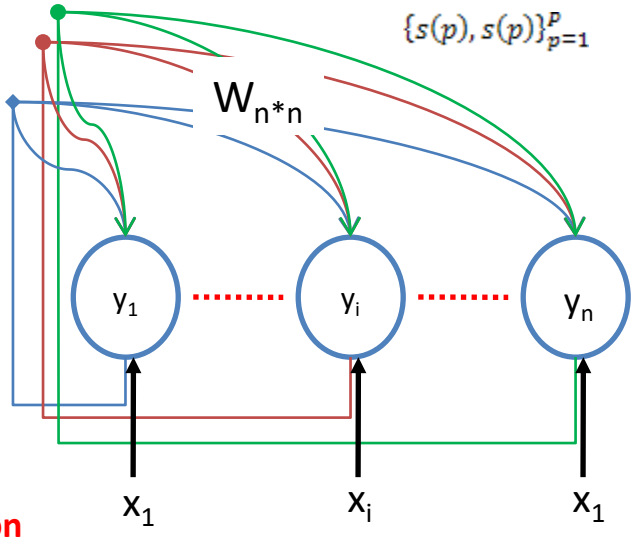
$$f\left([1\,1\,1\,-1]\begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}\right) = f([4\,4\,4\,-4]) = [1\,1\,1\,-1]$$

Input with one mistake components	Input with two missing components	Input with two mistake components
$f([-1\,1\,1\,-1]W) = [1\,1\,1\,-1]$ ok	$f([0\,0\,1\,-1]W) = [1\,1\,1\,-1]$ ok	$f([-1\,-1\,1\,-1]W) = [0\,0\,00]$ Not Ok
$f([1\,-1\,1\,-1]W) = [1\,1\,1\,-1]$ ok	$f([0\,1\,0\,-1]W) = [1\,1\,1\,-1]$ ok	
$f([1\,1\,-1\,\,-1]W) = [1\,1\,1\,-1]$ ok	$f([1\,1\,0\,0]W) = [1\,1\,1\,-1]$ ok	
$f([1\,1\,11]W) = [1\,1\,1\,-1]$ ok		

Recurrent Auto-Associative NET



OR



f(): Activation function

The iterations

(1) **x**:=disturbed or deformed **s**(p)

(2) $\mathbf{y}_1^T = f(\mathbf{x}^T W)$

(3) $\mathbf{y}_2^T = f(\mathbf{y}_1^T W)$

(4) $\mathbf{y}_3^T = f(\mathbf{y}_2^T W)$

.....

up to stop threshold

$x \rightarrow y_1 \rightarrow y_2 \rightarrow y_3 \rightarrow .... \rightarrow y_{final}$

$y_j = [y_{1j}, .., y_{ij}, .., y_{nj}]^T$

(Recurrent) Auto-Associative with threshold Function

Algorithm:

Step 0. Initialize weights to store patterns. (Use Hebbian learning.)

$$W = \sum_{p=1}^{P} s(p)s(p)^T - PI_n$$

Step 1. For each testing input vector, do Steps 2–5.

Step 2. Set activations **x**.

Step 3. While the stopping condition is false, repeat Steps 4 and 5.

Step 4. Update activations of all units (the threshold, θ_i, is usually taken to be zero):

$$x_i = \begin{cases} 1 & \text{if } \sum_j x_j w_{ij} > \theta_i \\ x_i & \text{if } \sum_j x_j w_{ij} = \theta_i \\ -1 & \text{if } \sum_j x_j w_{ij} < \theta_i. \end{cases}$$

Step 5. Test stopping condition: the net is allowed to iterate until the correct vector **x** matches a stored vector, or **x** matches a previous vector **x**, or the maximum number of iterations allowed is reached.

* Here, Hebbian Learning is the same modified version.

Example 1: store the vector (1,1,1,-1)

- The **Weight Matrix** by Modified Hebbian Learning Method

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

First input vector, (1, 0, 0, 0) **Three missing components**

Step 4: $(1, 0, 0, 0)\cdot\mathbf{W} = (0, 1, 1, -1)$.
Step 5: $(0, 1, 1, -1)$ is neither the stored vector nor an activation vector produced previously (since this is the first iteration), so we allow the activations to be updated again.
Step 4: $(0, 1, 1, -1)\cdot\mathbf{W} = (3, 2, 2, -2) \rightarrow (1, 1, 1, -1)$.
Step 5: $(1, 1, 1, -1)$ is the stored vector, so we stop.

OK

Second input vector, (0, 1, 0, 0) **Three missing components**

Step 4: $(0, 1, 0, 0)\cdot\mathbf{W} = (1, 0, 1, -1)$.
Step 5: $(1, 0, 1, -1)$ is not the stored vector or a previous activation vector, so we iterate.
Step 4: $(1, 0, 1, -1)\cdot\mathbf{W} = (2, 3, 2, -2) \rightarrow (1, 1, 1, -1)$.
Step 5: $(1, 1, 1, -1)$ is the stored vector, so we stop.

OK

Third input vector, (0, 0, 1, 0) Three missing components

Step 4: (0, 0, 1, 0)·W = (1, 1, 0, –1).
Step 5: (1, 1, 0, –1) is neither the stored vector nor a previous activation vector, so we iterate.
Step 4: (1, 1, 0, –1)·W = (2, 2, 3, –2) → (1, 1, 1, –1).
Step 5: (1, 1, 1, –1) is the stored vector, so we stop.

OK

Fourth input vector, (0, 0, 0, –1) Three missing components

Step 4: (0, 0, 0, –1)·**W** = (1, 1, 1, 0)
Step 5: Iterate.
Step 4: (1, 1, 1, 0)·**W** = (2, 2, 2, –3) → (1, 1, 1, , –1).
Step 5: (1, 1, 1, –1) is the stored vector, so we stop.

OK

For input vector (–1, –1, 1, 1). Three mistake components

Step 4: (–1, –1, 1, 1) . W = (–1, –1, –1, 1)
Step 5: Iterate.
Step 4: (–1, –1, – 1, 1) . W = (–1, –1, –1, 1)
Step 5: Since this is the input vector repeated, stop.

Not OK

Discrete Hopfield Net (Hopfield 1982/1984)

Two Differences with the former Algorithm

1. only one unit updates its activation at a time (based on the signal it receives from each other unit) and
2. each unit continues to receive an external signal in addition to the signal from the other units in the net.

Weight Matrix Initialization (Modified Hebbian Learning Method)

To store a set of binary patterns $\mathbf{s}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p)),$$

the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p [2s_i(p) - 1][2s_j(p) - 1] \qquad \text{for } i \neq j$$

and

$$w_{ii} = 0.$$

To store a set of bipolar patterns $\mathbf{s}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p)),$$

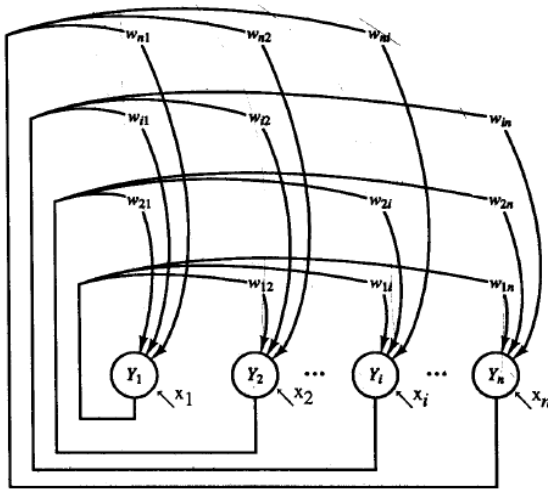
the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p s_i(p)s_j(p) \qquad \text{for } i \neq j$$

and

$$w_{ii} = 0.$$

The application algorithm is stated for binary patterns; the activation function can be modified easily to accommodate bipolar patterns.



Algorithm

Application Algorithm for the Discrete Hopfield Net

Step 0. Initialize weights to store patterns.
(Use Hebb rule.)

While activations of the net are not converged, do Steps 1–7.

Step 1. For each input vector **x**, do Steps 2–6.

Step 2. Set initial activations of net equal to the external input vector **x**:

$$y_i = x_i, (i = 1, \ldots n)$$

Step 3. Do Steps 4–6 for each unit Y_i.
(Units should be updated in random order.)

Step 4. Compute net input:

$$y\_in_i = x_i + \sum_j y_j w_{ji}.$$

Step 5. Determine activation (output signal):

$$y_i = \begin{cases} 1 & \text{if } y\_in_i > \theta_i \\ y_i & \text{if } y\_in_i = \theta_i \\ 0 & \text{if } y\_in_i < \theta_i. \end{cases}$$

Step 6. Broadcast the value of y_i to all other units.
(This updates the activation vector.)

Step 7. Test for convergence.

Example

Original (binary) stored pattern $(1, 1, 1, 0)$

Mistakes in the **first** and **second** components of the stored vector $(0, 0, 1, 0)$

Step 0. Initialize weights to store patterns:

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}.$$

Random Orders for computations: 1,4,3,2

Step 1. The input vector is $\mathbf{x} = (0, 0, 1, 0)$. For this vector,

Step 2. $\mathbf{y} = (0, 0, 1, 0)$.

Step 3. Choose unit Y_1 to update its activation:

Step 4. $y\_in_1 = x_1 + \sum_j y_j w_{j1} = 0 + 1.$

Step 5. $y\_in_1 > 0 \rightarrow y_1 = 1.$

Step 6. $\mathbf{y} = (1, 0, 1, 0)$.

Step 3. Choose unit Y_4 to update its activation:

Step 4. $y\_in_4 = x_4 + \sum_j y_j w_{j4} = 0 + (-2).$

Step 5. $y\_in_4 < 0 \rightarrow y_4 = 0.$
Step 6. $\mathbf{y} = (1, 0, 1, 0).$

Step 3. Choose unit Y_3 to update its activation:
Step 4. $y\_in_3 = x_3 + \sum_j y_j w_{j3} = 1 + 1.$
Step 5. $y\_in_3 > 0 \rightarrow y_3 = 1.$
Step 6. $\mathbf{y} = (1, 0, 1, 0).$

Step 3. Choose unit Y_2 to update its activation:
Step 4. $y\_in_2 = x_2 + \sum_j y_j w_{j2} = 0 + 2.$
Step 5. $y\_in_2 > 0 = y_2 = 1.$
Step 6. $y = (1, 1, 1, 0).$

Step 7. Test for convergence.

further iterations do not change the activation of any unit

Important Notes

1. Hopfield has utilized the gradient of a quadratic function as recurrent rules for output units of the network.
2. Repeating such recurrent rules causes that the energy function converges to a local minimum point and the output pattern will converge to a certain pattern.

Proof:

Consider following Energy Function

The Energy Function has a limited lower bound

$$E = -0.5y^T W y - x^T y + \theta^T y \qquad \theta^T = [\theta_1, \theta_2, \ldots, \theta_n]$$

$$E = -.5 \sum_{i \neq j} \sum_{j} y_i y_j w_{ij} - \sum_{i} x_i y_i + \sum_{i} \theta_i y_i.$$

$$\Delta E = E(y_i + \Delta y_i) - E(y_i)$$

$$\Delta E = -\left[\sum_{j} y_j w_{ij} + x_i - \theta_i\right] \Delta y_i$$

a change Δy_i will occur

If y_i is positive, it will change to zero if

$$x_i + \sum_{j} y_j w_{ji} < \theta_i.$$

This gives a negative change for y_i. In this case, $\Delta E < 0$.

If y_i is zero, it will change to positive if

$$x_i + \sum_{j} y_j w_{ji} > \theta_i.$$

This gives a positive change for y_i. In this case, $\Delta E < 0$.

Thus Δy_i is positive only if $[\sum_j y_j w_{ji} + x_i - \theta_i]$ is positive, and Δy_i is negative only if this same quantity is negative. Therefore, the energy cannot increase. Hence, since the energy is bounded, the net must reach a stable equilibrium such that the energy does not change with further iteration.

Storage Capacity. Hopfield found experimentally that the number of binary patterns that can be stored and recalled in a net with reasonable accuracy, is given approximately by

$$P \approx 0.15n,$$

where n is the number of neurons in the net.

Abu-Mostafa and St Jacques (1985) have performed a detailed theoretical analysis of the information capacity of a Hopfield net. For a similar net using bipolar patterns, McEliece, Posner, Rodemich, and Venkatesh (1987) found that

$$P \approx \frac{n}{2 \log_2 n}.$$

How to utilize Hebbian Memory Networks for **(1) non-orthogonal, (2) analogue input patterns (3)** with high enough capacity?

For Arbitrary number of non-orthogonal analogue inputs:

(1) Provide a transformation function to generate bipolar patterns from input analogue patterns.
The dimension of the bipolar patterns must be more than or equal to the number of input patterns
The generated bipolar patterns must be linearly independent form each other.
the order of distances of any analogue input pattern with all other input patterns must be seen in bipolar patterns, too.

→

(2) Provide a transformation matrix to get orthogonal bipolar form of the former generated bipolar patterns.

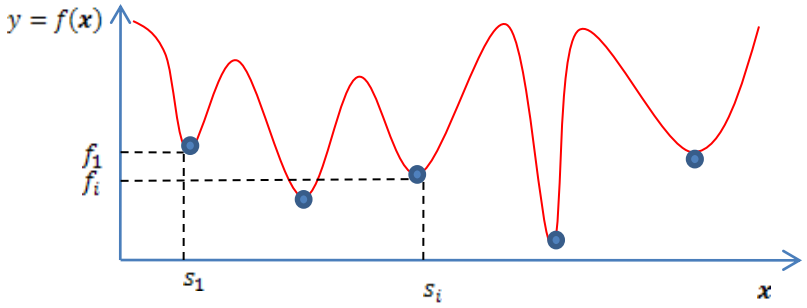
→

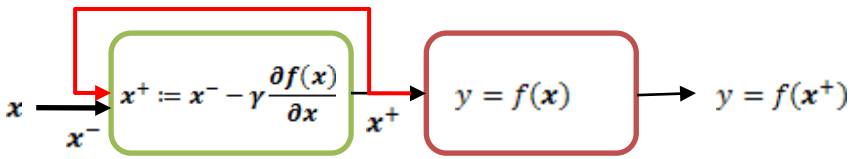
(3) Use Hebbian Learning Rule to store the required patterns

Restoring arbitrary number of patterns

without dimension extending and orthogonal transformation

1. **Assume it is aimed to train a network which associate different input patterns $\{S_i \ i=1,2..,P\}$ to target patterns $\{t_i \ i=1,2..,P\}$.**
2. **Define a nonlinear scalar function with n-dimensional input variable: y=f(x). Such function must have at least "P" local minimum points :$\{y=f_i \ i=1,2..,P\}$ at $\{x=S_i \ i=1,2..,P\}$.**
3. **Assume that each local minimum point $(y=f_i)$ denotes the output patterns (t_i) and for two equal (not equal) output patterns two corresponding levels are equal (not equal) .**
4. **Now define a network which has two sequenced parts. At the First part the Network in a recurrent structure updates the input pattern by Gradient descent of the defined function. At the second part the Network produce the output of the function based on the updated input (by the first part).**
5. **It is expected that the network at first part converges to an input pattern (S_j) and in the second part (f_j) will be appeared which is considered as a label for (t_j).**





Capacity of one layer MNN in restoring input patterns

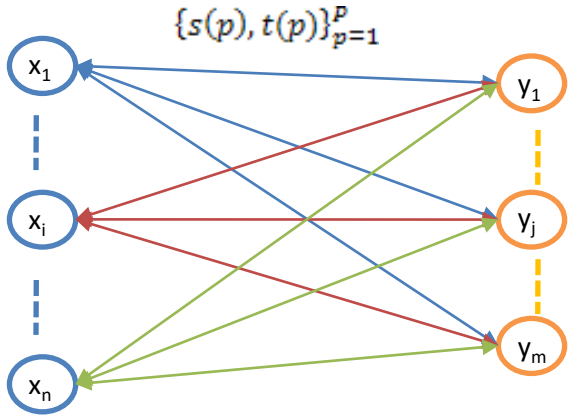
For n-dimensional orthogonal input patterns

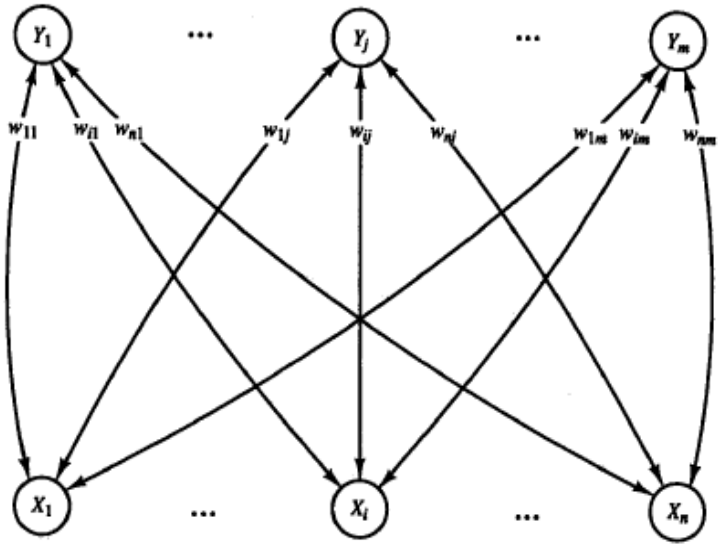
Weight Matrix	Hetro-Associative	Auto-Associative Bipolar patterns
Hebbian Matrix	"Capacity =n"	"Capacity =n"
Modified Hebbian Matrix	-	"Capacity =n-1"

Two important notes

1. For Auto-associative MNNs each stored vector is an Eigen-vector of the weight matrix of the network.
2. If additive disturbance of an input pattern (with arbitrary norm) are defined in nullity space of the weight matrix, the network will have unlimited robustness against it. Hence, it is suggested to use a low percentage of the whole capacity for restoring patterns.

Bidirectional Associative Memory (**BAM**)





$$x = \underset{\text{deformed}}{\overset{\text{disturbed}}{(s(p))}} \qquad y = \underset{\text{deformed}}{\overset{\text{disturbed}}{(t(p))}}$$

$$x^T = f(y^T W^T) \longleftrightarrow y^T = f(x^T W)$$

Repeat up to
Stop Condition

1. This Network is a recurrent form of Hetro-Associative Network
2. At First, two disturbed and deformed patterns of s(p) and t(p) are assigned to “x” and “y” as two entrances of the network.
3. Using a bidirectional pattern association finally two input and output patterns converged to certain pair of patterns.
4. It is expected that s(p) and t(p) being final retrieved patterns. Other wise the network has failed to associate to the restored original patterns.

Setting the weights

Setting the Weights. The weight matrix to store a set of input and target vectors $\mathbf{s}(p):\mathbf{t}(p)$, $p = 1, \ldots, P$, where

$$\mathbf{s}(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$

and

$$\mathbf{t}(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p)),$$

can be determined by the Hebb rule. The formulas for the entries depend on whether the training vectors are binary or bipolar. For binary input vectors, the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p (2s_i(p) - 1)(2t_j(p) - 1).$$

For bipolar input vectors, the weight matrix $\mathbf{W} = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_p s_i(p)t_j(p).$$

Activation Function

Activation Functions. The activation function for the discrete BAM is the appropriate step function, depending on whether binary or bipolar vectors are used.

For binary input vectors, the activation function for the Y-layer is

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > 0 \\ y_j & \text{if } y\_in_j = 0 \\ 0 & \text{if } y\_in_j < 0, \end{cases}$$

and the activation function for the X-layer is

$$x_i = \begin{cases} 1 & \text{if } x\_in_i > 0 \\ x_i & \text{if } x\_in_i = 0 \\ 0 & \text{if } x\_in_i < 0. \end{cases}$$

For bipolar input vectors, the activation function for the Y-layer is

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > \theta_j \\ y_j & \text{if } y\_in_j = \theta_j \\ -1 & \text{if } y\_in_j < \theta_j, \end{cases}$$

and the activation function for the X-layer is

$$x_i = \begin{cases} 1 & \text{if } x\_in_i > \theta_i \\ x_i & \text{if } x\_in_i = \theta_i \\ -1 & \text{if } x\_in_i < \theta_i. \end{cases}$$

Algorithm

Algorithm.

Step 0. Initialize the weights to store a set of P vectors;
initialize all activations to 0.

Step 1. For each testing input, do Steps 2–6.

Step 2a. Present input pattern **x** to the X-layer
(i.e., set activations of X-layer to current input pattern).

Step 2b. Present input pattern **y** to the Y-layer.
(Either of the input patterns may be the zero vector.)

Step 3. While activations are not converged, do Steps 4–6.

Step 4. Update activations of units in Y-layer.
Compute net inputs:

$$y\_in_j = \sum_i w_{ij} x_i.$$

Compute activations:

$$y_j = f(y\_in_j).$$

Send signal to X-layer.

Step 5. Update activations of units in X-layer.
Compute net inputs:

$$x\_in_i = \sum_j w_{ij} y_j.$$

Compute activations:

$$x_i = f(x\_in_i).$$

Send signal to Y-layer.

Step 6. Test for convergence:
If the activation vectors **x** and **y** have reached equilibrium, then stop; otherwise, continue.

Example

Example 3.23 A BAM net to associate letters with simple bipolar codes

Consider the possibility of using a (discrete) BAM network (with bipolar vectors) to map two simple letters (given by 5 × 3 patterns) to the following bipolar codes:

(-1, 1)

(1, 1)

The weight matrices are:

(to store $A \rightarrow -11$)

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ -1 & 1 \end{bmatrix}$$

($C \rightarrow 11$)

$$\begin{bmatrix} -1 & -1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

(W, to store both)

$$\begin{bmatrix} 0 & -2 \\ 0 & 2 \\ 2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ -2 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix}$$

To illustrate the use of a BAM, we first demonstrate that the net gives the correct Y vector when presented with the x vector for either the pattern A or the pattern C:

Verification for original patterns (former example)

INPUT PATTERN *A*

$$(-1\ 1\ -1\ 1\ -1\ 1\ 1\ 1\ 1\ 1\ -1\ 1\ 1\ -1\ 1)\mathbf{W} = (-14, 16) \rightarrow (-1, 1).$$

INPUT PATTERN *C*

$$(-1\ 1\ 1\ 1\ -1\ -1\ 1\ -1\ -1\ 1\ -1\ -1\ -1\ 1\ 1)\mathbf{W} = (14, 16) \rightarrow (1, 1).$$

To see the bidirectional nature of the net, observe that the Y vectors can also be used as input. For signals sent from the Y-layer to the X-layer, the weight matrix is the transpose of the matrix $\mathbf{W}$, i.e.,

$$\mathbf{W}^T = \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}.$$

For the input vector associated with pattern A, namely, $(-1, 1)$, we have

$$(-1, 1)\mathbf{W}^T =$$

$$(-1, 1) \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ -2\ \ 2\ \ -2\ \ 2\ \ 2\ \ 2\ \ 2\ \ 2\ \ -2\ \ 2\ \ 2\ \ -2\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ 1\ \ -1\ \ 1).$$

This is pattern A.

Similarly, if we input the vector associated with pattern C, namely, $(1, 1)$, we obtain

$$(1, 1)\mathbf{W}^T =$$

$$(1, 1) \begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ 2\ \ 2\ \ -2\ \ -2\ \ 2\ \ -2\ \ -2\ \ 2\ \ -2\ \ -2\ \ -2\ \ 2\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ 1\ \ 1\ \ -1\ \ -1\ \ 1\ \ -1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ 1\ \ 1),$$

which is pattern C.

test for noisy patterns
(former example)

Example 3.24 Testing a BAM net with noisy input

In this example, the net is given a y vector as input that is a noisy version of one of the training y vectors and no information about the corresponding x vector (i.e., the x vector is identically 0). The input vector is (0, 1); the response of the net is

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

$$(0, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2 \quad 2 \quad 0 \quad 2 \quad -2 \quad 0 \quad 2 \quad 0 \quad 0 \quad 2 \quad -2 \quad 0 \quad 0 \quad 0 \quad 2)$$

$$\rightarrow (-1 \quad 1 \quad 0 \quad 1 \quad -1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad -1 \quad 0 \quad 0 \quad 0 \quad 1).$$

Note that the units receiving 0 net input have their activations left at that value, since the initial x vector is 0. This x vector is then sent back to the *Y*-layer, using the weight matrix **W**:

$$(-1 \quad 1 \quad 0 \quad 1 \quad -1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad -1 \quad 0 \quad 0 \quad 0 \quad 1)\begin{bmatrix} 0 & -2 \\ 0 & 2 \\ 2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ -2 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\rightarrow (0 \ 1).$$

Converged to non-original patterns

test for noisy patterns with modifications (former example)

This result is not too surprising, since the net had no information to give it a preference for either A or C. The net has converged (since, obviously, no further changes in the activations will take place) to a spurious stable state, i.e., the solution is not one of the stored pattern pairs.

If, on the other hand, the net was given both the input vector $\mathbf{y}$, as before, and some information about the vector $\mathbf{x}$, for example,

$$\mathbf{y} = (0\ \ 1),\ \mathbf{x} = (0\ \ 0\ \ -1\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ -1\ \ 0),$$

the net would be able to reach a stable set of activations corresponding to one of the stored pattern pairs.

Note that the $\mathbf{x}$ vector is a noisy version of

$$A = (-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ 1\ \ -1\ \ 1),$$

where the nonzero components are those that distinguish A from

$$C = (-1\ \ 1\ \ 1\ \ 1\ \ -1\ \ -1\ \ 1\ \ -1\ \ -1\ \ 1\ \ -1\ \ -1\ \ -1\ \ 1\ \ 1).$$

Now, since the algorithm specifies that if the net input to a unit is zero, the activation of that unit remains unchanged, we get

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

$$(0, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ 0\ \ 2\ \ -2\ \ 0\ \ 2\ \ 0\ \ 0\ \ 2\ \ -2\ \ 0\ \ 0\ \ 0\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ -1\ \ 1\ \ 1\ \ -1\ \ 1),$$

which is pattern A.

Since this example is fairly extreme, i.e., every component that distinguishes A from C was given an input value for A, let us try something with less information given concerning $\mathbf{x}$.

For example, let $\mathbf{y} = (0\ 1)$ and $\mathbf{x} = (0\ 0\ -1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$. Then

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

$$(0, 1)\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & -2 & 0 & -2 & -2 & 0 & 0 & -2 & -2 & 2 & 0 \\ -2 & 2 & 0 & 2 & -2 & 0 & 2 & 0 & 0 & 2 & -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$= (-2\ \ 2\ \ 0\ \ 2\ \ -2\ \ 0\ \ 2\ \ 0\ \ 0\ \ 2\ \ -2\ \ 0\ \ 0\ \ 0\ \ 2)$$

$$\rightarrow (-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ -1\ \ 0\ \ 0\ \ 0\ \ 1),$$

which is not quite pattern A.

So we try iterating, sending the $\mathbf{x}$ vector back to the Y-layer using the weight matrix W:

$$(-1\ \ 1\ \ -1\ \ 1\ \ -1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ -1\ \ 0\ \ 0\ \ 0\ \ 1)\begin{bmatrix} 0 & -2 \\ 0 & 2 \\ 2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ -2 & 0 \\ -2 & 0 \\ 2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$\rightarrow (-6, 10) \rightarrow (-1, 1).$$

If this pattern is fed back to the X-layer one more time, the pattern A will be produced.

Two notes

Hamming distance

The number of different bits in two binary or bipolar vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ is called the *Hamming distance* between the vectors and is denoted by $H[\mathbf{x}_1, \mathbf{x}_2]$. The average Hamming distance between the vectors is $\frac{1}{n}H[\mathbf{x}_1, \mathbf{x}_2]$, where n is the number of components in each vector. The **x** vectors in Examples 3.23 and 3.24, namely,

differ in the 3rd, 6th, 8th, 9th, 12th, 13th, and 14th positions. This gives an average Hamming distance between these vectors of 7/15. The average Hamming distance between the corresponding **y** vectors is 1/2.

Energy function

The convergence of a BAM net can be proved using an energy or Lyapunov function, in a manner similar to that described for the Hopfield net. A Lyapunov function must be decreasing and bounded. For a BAM net, an appropriate function is the average of the signal energy for a forward and backward pass:

$$L = -0.5\,(\mathbf{x}W\mathbf{y}^T + \mathbf{y}W^T\mathbf{x}^T).$$

However, since $\mathbf{x}W\mathbf{y}^T$ and $\mathbf{y}W^T\mathbf{x}^T$ are scalars, and the transpose of a scalar is a scalar, the preceding expression can be simplified to

$$L = -\mathbf{x}W\mathbf{y}^T$$

$$= -\sum_{j=1}^{m}\sum_{i=1}^{n} x_i w_{ij} y_j.$$

For binary or bipolar step functions, the Lyapunov function is clearly bounded below by

$$-\sum_{j=1}^{m}\sum_{i=1}^{n} |\, w_{ij}\,|.$$

2.1 MNNs with Implicit input patterns

- Some known Recurrent NNs
  - Recurrent Neural Networks(RNNs)
  - Long-Short Term Memory (LSTM)
  - Gated Recurrent Unit (GRU)

- Such Networks make nonlinear non-homogenous difference Equations. By solving such equations, the nonlinear functions, which generate the output patterns based on the implicit sequenced input patterns, are revealed.
- Recurrent structures will make a plenty of memories utilized in pattern associations between implicit inputs and target outputs.

- Pattern Association (Memories) types.
  - **One to one:** many classification problems and identification by fingerprint/biometric signals/signature/…
  - **Many to one:** voice/handwriting/walking in identification
  - **One to Many-:** image captioning/video captioning
  - **Many to Many** -- Translation machines...text to picture/Video

Some Applications of RNNs

1. Language Modeling and Generating Text

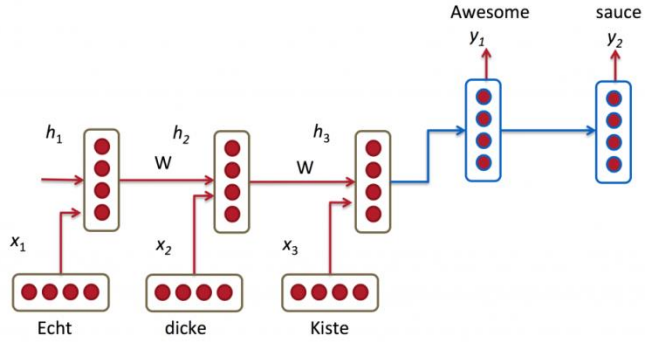
Given a sequence of words we want to predict the probability of each word given the previous words. Language Models allow us to measure how likely a sentence is, which is an important input for Machine Translation (since high-probability sentences are typically correct).

- Recurrent neural network based language model
- Extensions of Recurrent neural network based language model
- Generating Text with Recurrent Neural Networks

2. Machine Translation

- A Recursive Recurrent Neural Network for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Joint Language and Translation Modeling with Recurrent Neural Networks

Machine Translation is similar to language modeling in that our input is a sequence of words in our source language (e.g. German). We want to output a sequence of words in our target language (e.g. English).



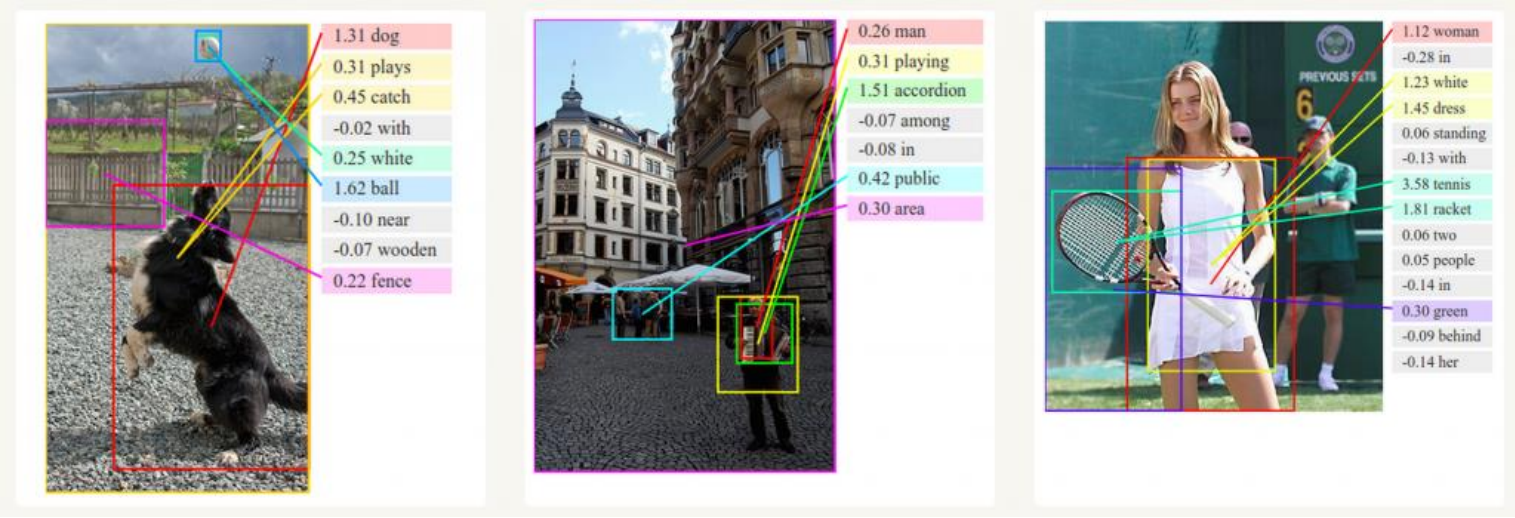
3. Speech Recognition

Given an input sequence of acoustic signals from a sound wave, we can predict a sequence of phonetic segments together with their probabilities.

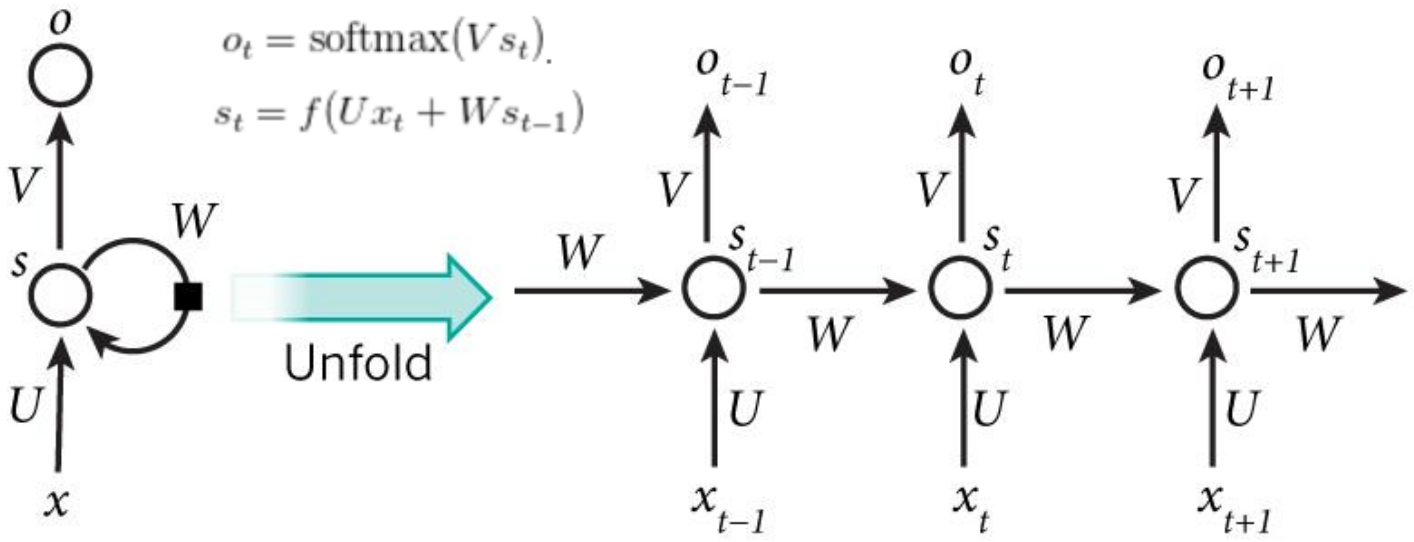
1. Towards End-to-End Speech Recognition with Recurrent Neural Networks

4. Generating Image Descriptions

- Together with convolutional Neural Networks, RNNs have been used as part of a model to generate descriptions for unlabeled images



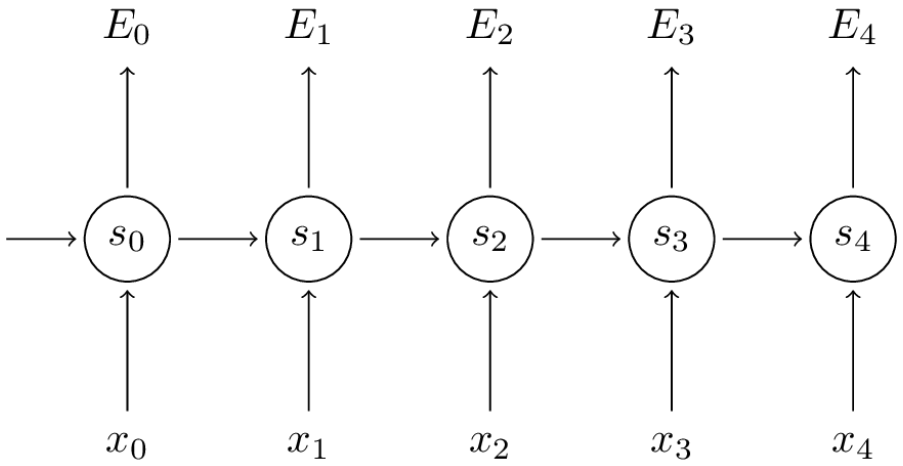
A Typical RNN



- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. s_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function f usually is a nonlinearity such as tanh or ReLU. s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- o_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. $o_t = \text{softmax}(Vs_t)$.

Training RNNs

Back Propagation Through Time (BPTT)



$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

Loss function

$$E_t(y_t, \hat{y}_t) = \underbrace{0.5(y_t - \hat{y}_t)^2}_{Squared\ Error}$$

or

$$E_t(y_t, \hat{y}_t) = \underbrace{-y_t \log(\hat{y}_t)}_{Cross\ entropy}$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

Updating Rules (for t=3)

$$W^+ = W^- - \eta \frac{\partial E_3}{\partial W}\Big|_{U^-, V^-, W^-}$$

$$U^+ = U^- - \eta \frac{\partial E_3}{\partial U}\Big|_{U^-, V^-, W^-}$$

$$V^+ = V^- - \eta \frac{\partial E_3}{\partial V}\Big|_{U^-, V^-, W^-}$$

Back Propagation Through Time (BPTT)

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \text{softmax}(Vs_t)$$

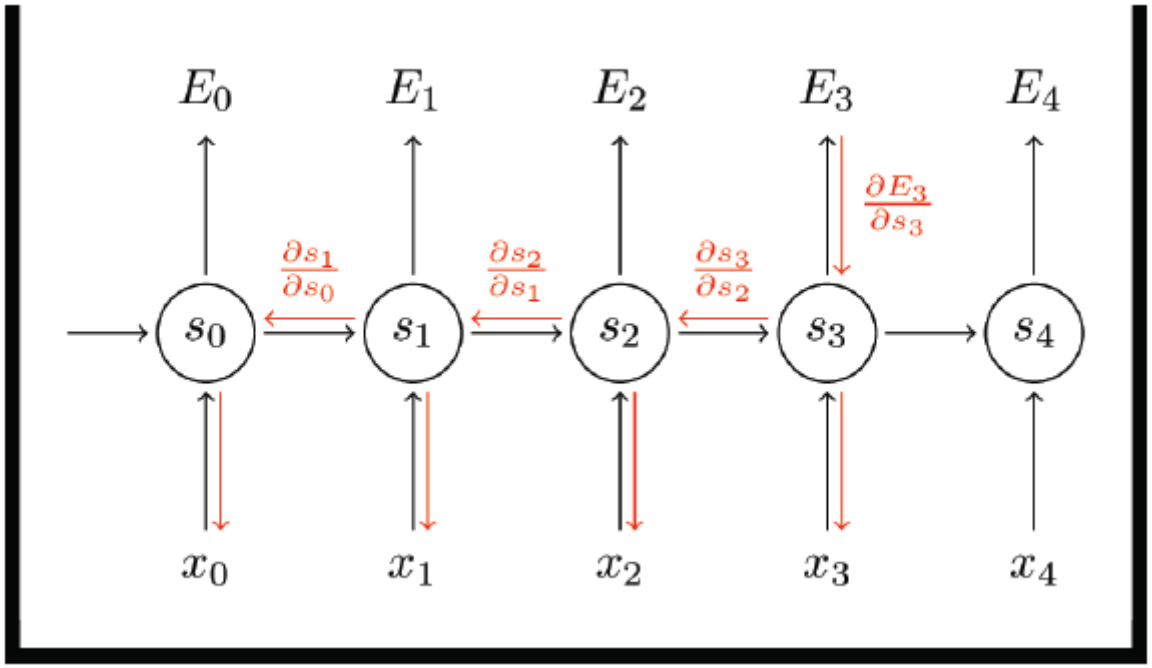
$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W}$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3}\frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

$$\frac{\partial E_3}{\partial U} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial U}$$

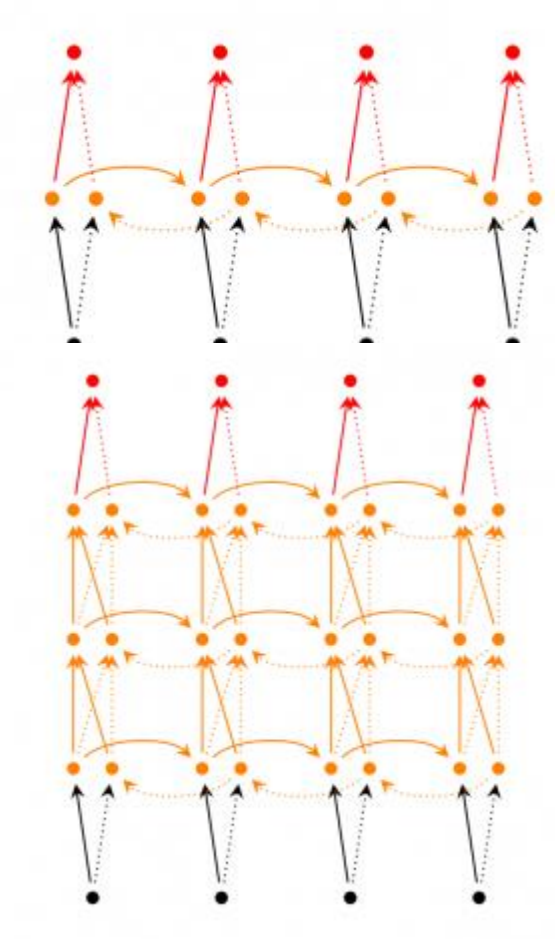
$$\frac{\partial E_3}{\partial U} = \sum_{k=0}^{3}\frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial U}$$

$$\frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V}$$
$$= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V}$$
$$= (\hat{y}_3 - y_3) \otimes s_3$$



RNN Extensions

- **Bidirectional RNNs** are based on the idea that the output at time may not only depend on the previous elements in the sequence, but also future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple. They are just two RNNs stacked on top of each other. The output is then computed based on the hidden state of both RNNs.
- **Deep (Bidirectional) RNNs** are similar to Bidirectional RNNs, only that we now have multiple layers per time step. In practice this gives us a higher learning capacity (but we also need a lot of training data)



Main limitation in RNN

derivative of sigmoid functions is less than one.

$$\frac{\partial \tanh(x)}{\partial x} < 1 \ , \frac{\partial \text{sigm}(x)}{\partial x} < 1$$

In "BPTT" when the RNN is unfolded for many times the back-propagated gradient coefficient is vanished for the inputs taken at older times.

$$\text{In computing } \left(\frac{\partial E_t}{\partial W}\right) \text{ or } \left(\frac{\partial E_t}{\partial U}\right) \text{ for } d \gg 1:$$

$$\left\| \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial s_t} \frac{\partial s_t}{\partial s_{t-1}} \dots \frac{\partial s_{t-d+1}}{\partial s_{t-d}} \right\| \to 0$$

The learning for long dependencies is not effective any more.

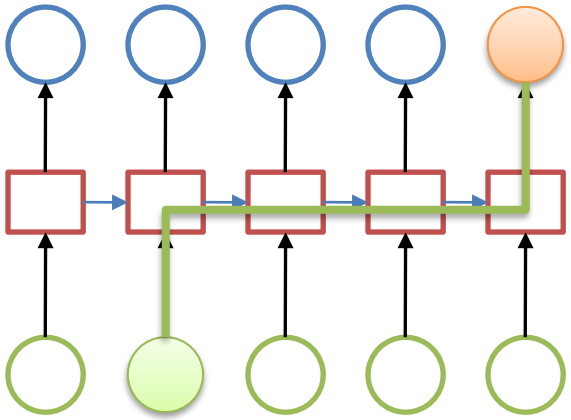
The Vanishing Gradient Problem

Error gradients vanish exponentially quickly with the size of the time lag between important events

RNNs are good to make Short Term Memory

The clouds are in the SKY

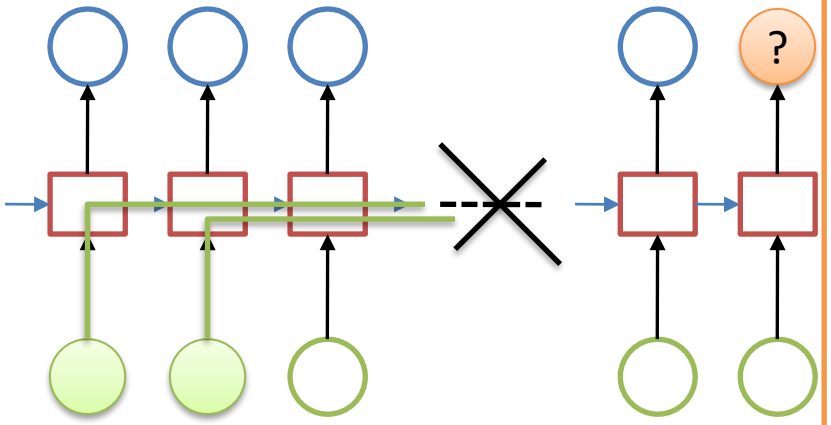
implicit input → target



RNNs are not good to make Long Term Memory

I grew up in France……I speak fluent French

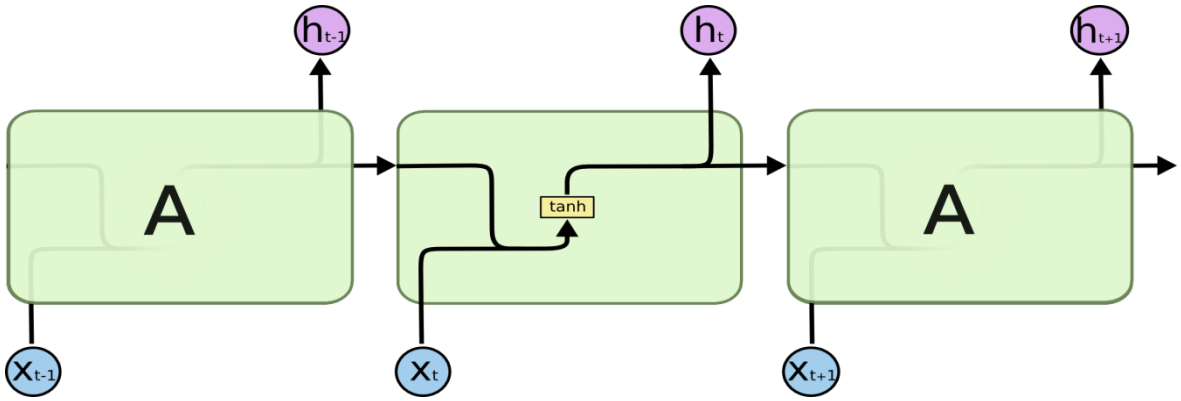
implicit input →? target



LSTM (long short term memory)

Hochreitor & Shmidhuber 1997

Module in RNN

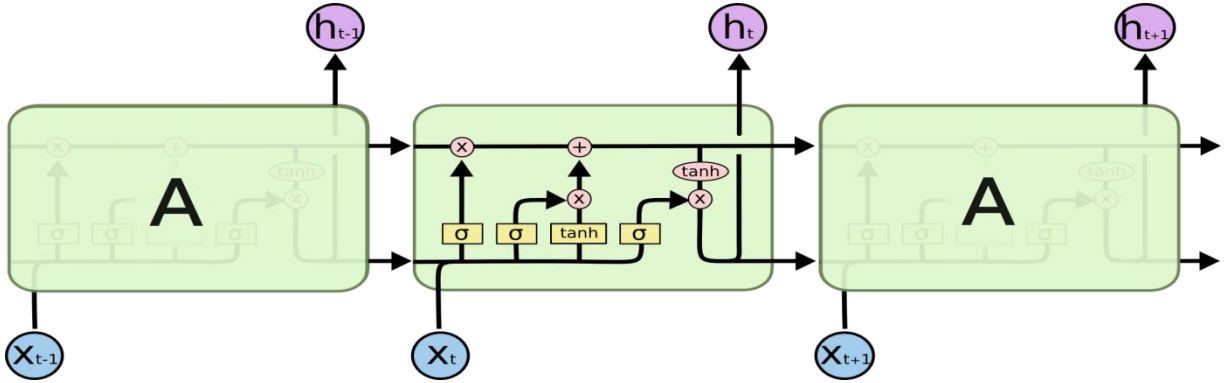


($\mathbf{h_t}$ is the same $\mathbf{s_t}$)

$\mathbf{h_t} = \tanh(Ux_t + W\,\mathbf{h_{t-1}})$

The repeating module in a standard RNN contains a single layer.

Module in LSTM

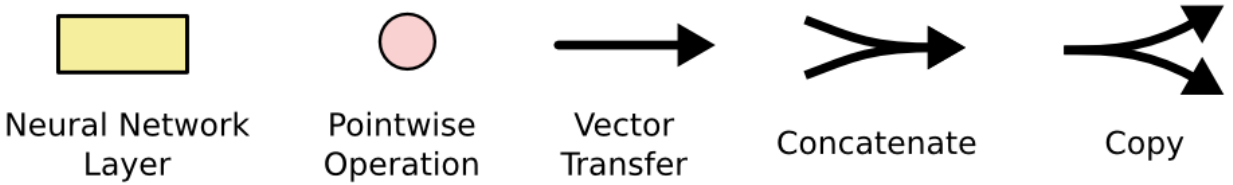


The repeating module in an LSTM contains four interacting layers.

the LSTM can read, write and delete information from its memory.

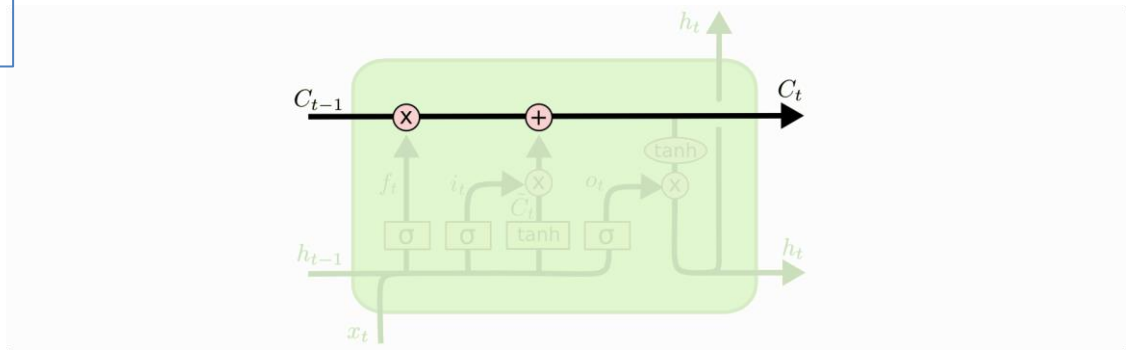
Some Concepts

The notation

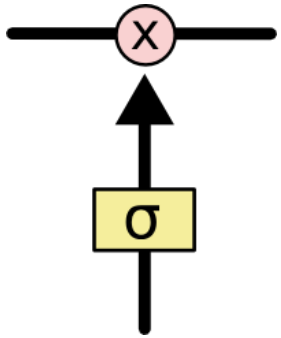


Two key concepts in LSTMs

1. Cell state



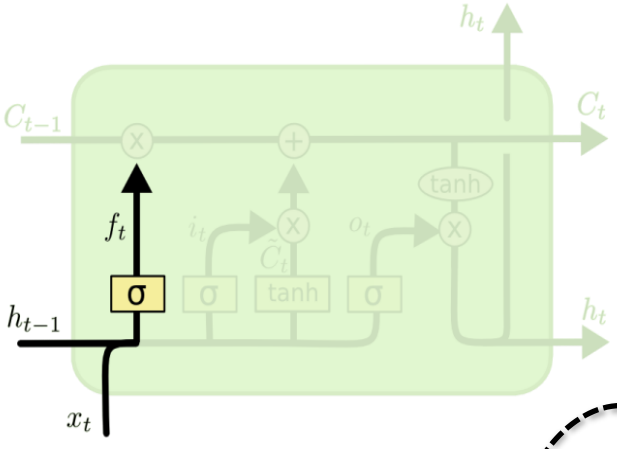
2. Gate



Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

Step-by-Step LSTM Walk Through

Step1: to decide what information we're going to throw away from the cell state.

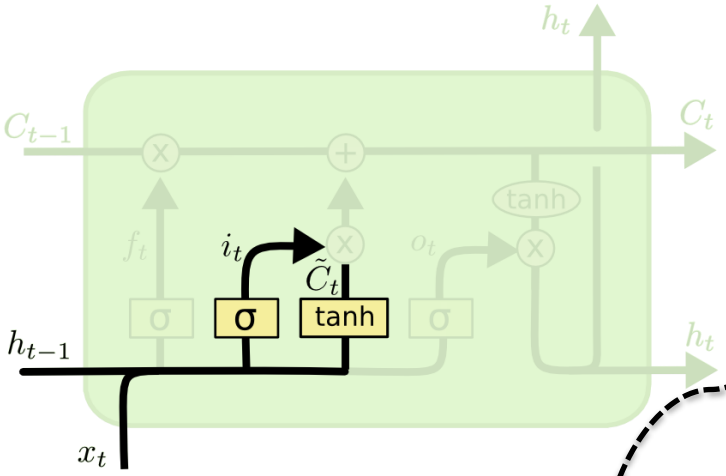


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Forget Gate

A Gate for Saving/Deleting Memory

Step2: to decide what new information we're going to store in the cell state.



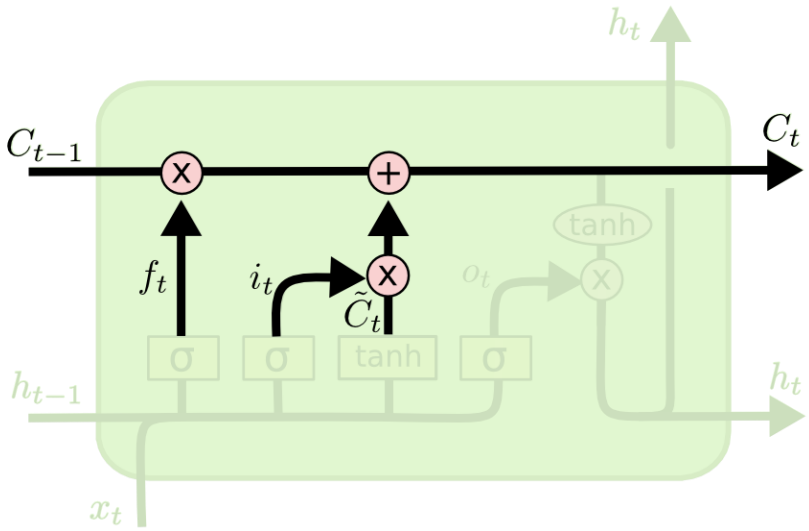
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

Input Gate

A Gate for Writing to Memory

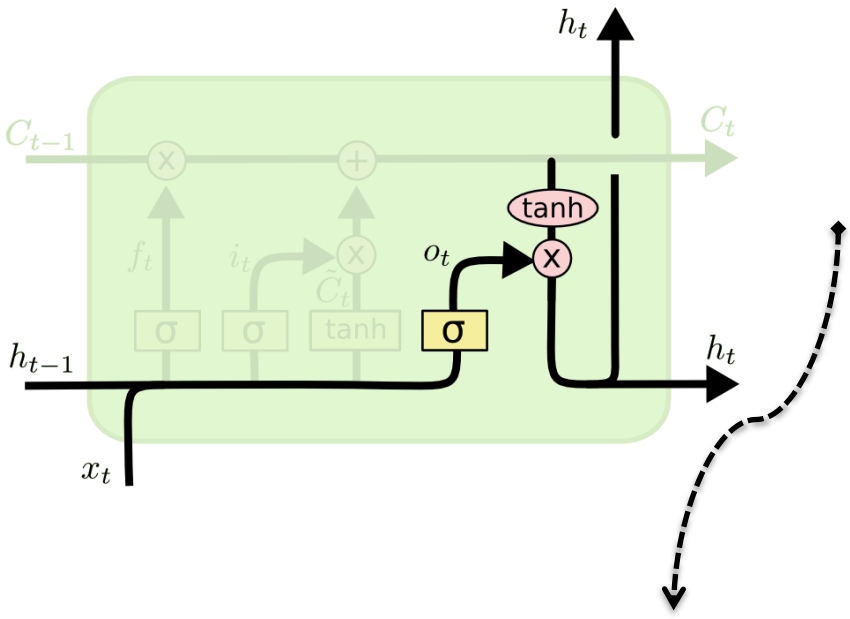
Step3: drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

Deleting/Saving and Writing to Memory

Step4: to decide what we're going to output



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

$$h_t = o_t * \tanh\left(C_t\right)$$

Output Gate

Reading from Memory

Compact Form Equations

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

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

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

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

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

$$h_t = o_t * \tanh\left(C_t\right)$$

$$\hat{y}_t = f(V.h_t + b_v)$$

$x_t \epsilon \mathbf{R}^n$

$W_{f,i,c,o} \epsilon \mathbf{R}^{h \times (h+n)}$

$b_{f,i,c,o} \epsilon \mathbf{R}^{h \times 1}$

$f_t, i_t, o_t, h_t. c_t \epsilon \mathbf{R}^h$

$V \epsilon \mathbf{R}^{m \times h}$ $\hat{y}_t, y_t \epsilon \mathbf{R}^m$

Parameters: $\{(W_i, b_i), (W_f, b_f), (W_C, b_c), (W_o, b_o), (V, b_V)\}$

$$\text{Loss Function: } E_t = \sum_t E(y_t - \hat{y}_t)$$

$$\text{Parameters}^{+} = \text{Parameters}^{-} - \gamma \frac{\partial E_t}{\partial \text{Parameters}}$$

- Parameters are updated by “BPTT”
  All computed gradients of parameters are added together through the time

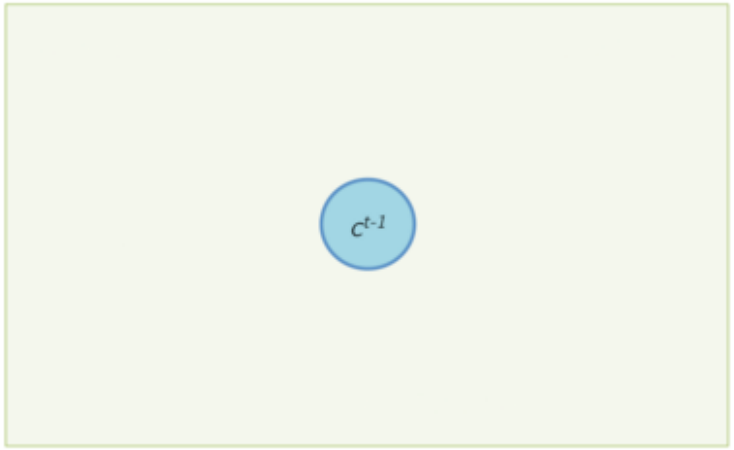
LSTM Learning methods

1. **Like RNN, "BPTT" can be performed to learn the weights and biases of a LSTM module. Unlike standard RNNs, the error remains in the unit's memory.**
2. LSTM can also be trained by a combination of artificial evolution for weights to the hidden units, and pseudo-inverse or support vector machines for weights to the output units.
3. In reinforcement learning applications LSTM can be trained by policy gradient methods, evolution strategies or genetic algorithms.

LSTM Forward and Backward Pass, Arun Mallya

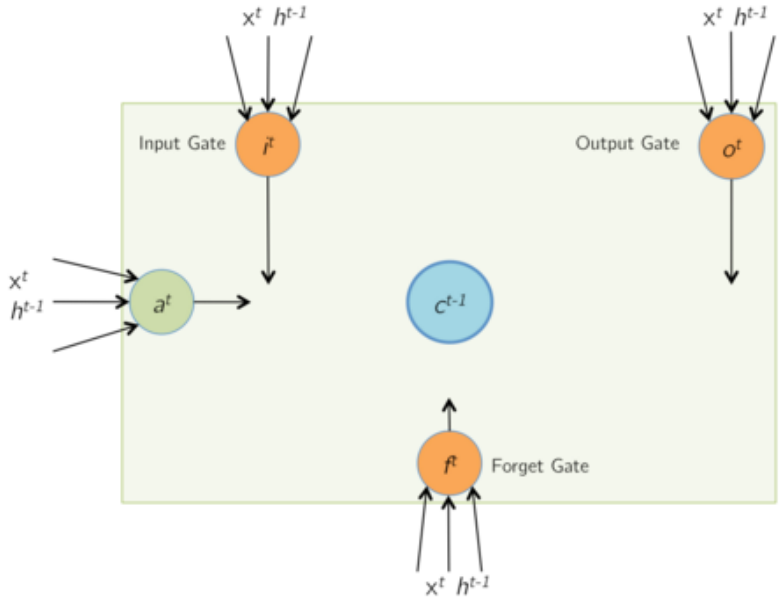
Forward Pass: Initial State

Initially, at time t, the memory cells of the LSTM contain values from the previous iteration at time (t − 1).



Forward Pass: Input and Gate Computation

At time t, The LSTM receives a new input vector x^t (including the bias term), as well as a vector of its output at the previous timestep, h^{t-1}.



$$
\begin{aligned}
a^t &= \tanh(W_c x^t + U_c h^{t-1}) = \tanh(\hat{a}^t) \\
i^t &= \sigma(W_i x^t + U_i h^{t-1}) = \sigma(\hat{i}^t) \\
f^t &= \sigma(W_f x^t + U_f h^{t-1}) = \sigma(\hat{f}^t) \\
o^t &= \sigma(W_o x^t + U_o h^{t-1}) = \sigma(\hat{o}^t)
\end{aligned}
$$

Ignoring the non-linearities,

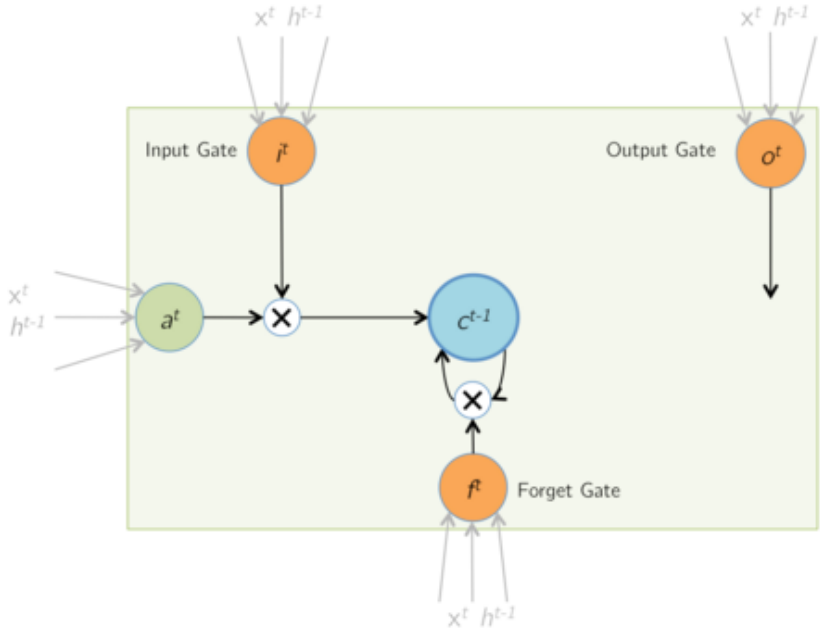
$$
z^t = \begin{bmatrix} \hat{a}^t \\ \hat{i}^t \\ \hat{f}^t \\ \hat{o}^t \end{bmatrix} = \begin{bmatrix} W^c & U^c \\ W^i & U^i \\ W^f & U^f \\ W^o & U^o \end{bmatrix} \times \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix}
$$

$$
= W \times I^t
$$

If the input x^t is of size $n \times 1$, and we have d memory cells, then the size of each of W_* and U_* is $d \times n$, and and $d \times d$ resp. The size of W will then be $4d \times (n + d)$. Note that each one of the d memory cells has its own weights W_* and U_*, and that the only time memory cell values are shared with other LSTM units is during the product with U_*.

Forward Pass: Memory Cell Update

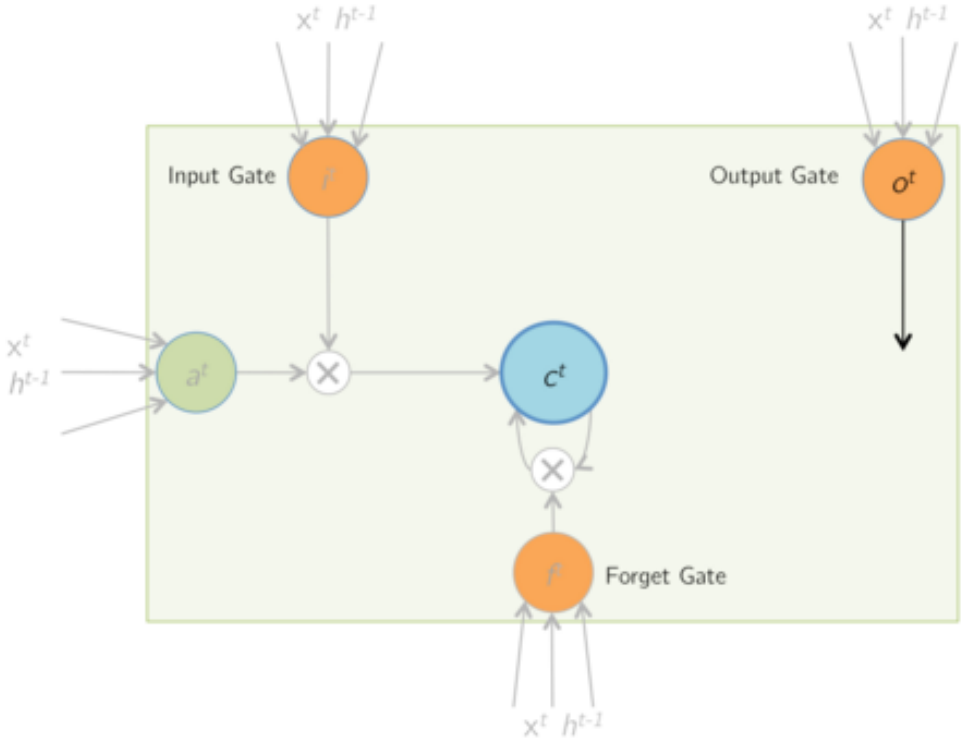
During this step, the values of the memory cells are updated with a combination of a^t, and the previous cell contents c^{t-1}. The combination is based on the magnitudes of the input gate i^t and the forget gate f^t. $\odot$ denotes elementwise product (Hadamard product).



$$c^t = i^t \odot a^t + f^t \odot c^{t-1}$$

Forward Pass: Updated Memory Cells

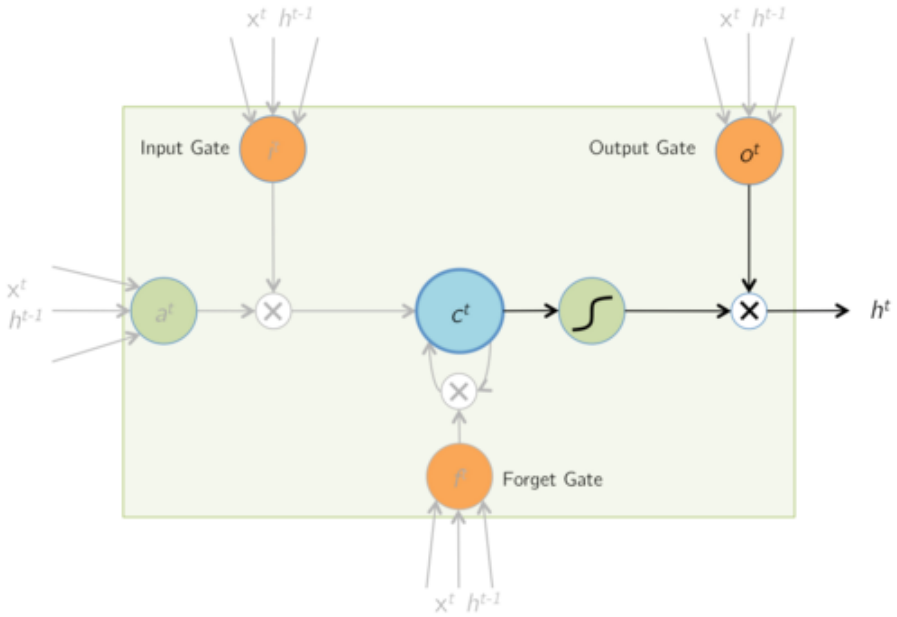
The contents of the memory cells are updated to the latest values.



$$c^{t-1} \rightarrow c^t$$

Forward Pass: Output

Finally, the LSTM cell computes an output value by passing the updated (and current) cell value through a non-linearity. The output gate determines how much of this computed output is actually passed out of the cell as the final output h^t.



$$h^t = o^t \odot \tanh(c^t)$$

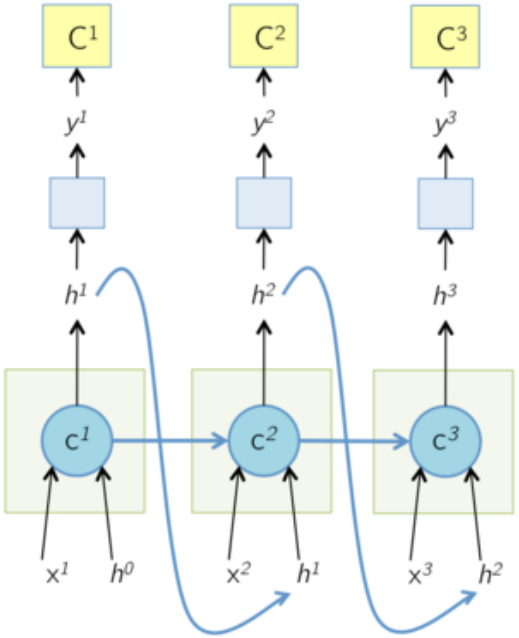
$$\hat{y}_t = f(V^t h^t)$$

$$E = \sum_{\tau=1}^{t} C^t(\hat{y}_t, y_t)$$

$$E(W)$$

Forward Pass: Unrolled Network

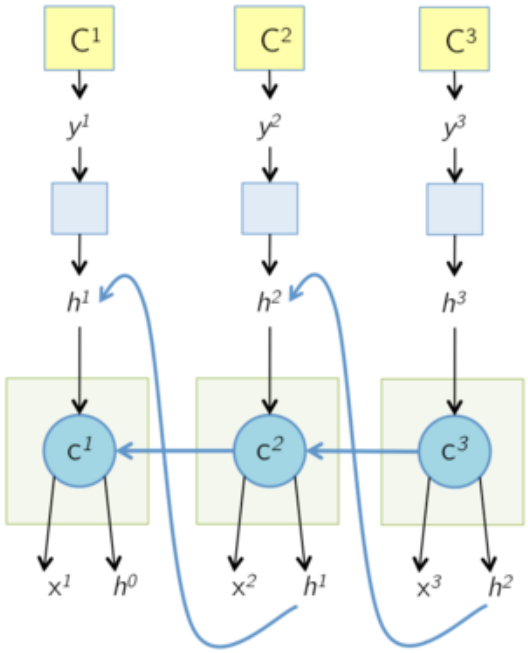
The unrolled network during the forward pass is shown below. Note that the gates have not been shown for brevity. An interesting point to note here is that in the computational graph below, the cell state at time T, c^T is responsible for computing h^T as well as the next cell state c^{T+1}. At each time step, the cell output h^T is shown to be passed to some more layers on which a cost function C^T is computed, as the way an LSTM would be used in a typical application like captioning or language modeling.



There are two parallel **Forward** passes

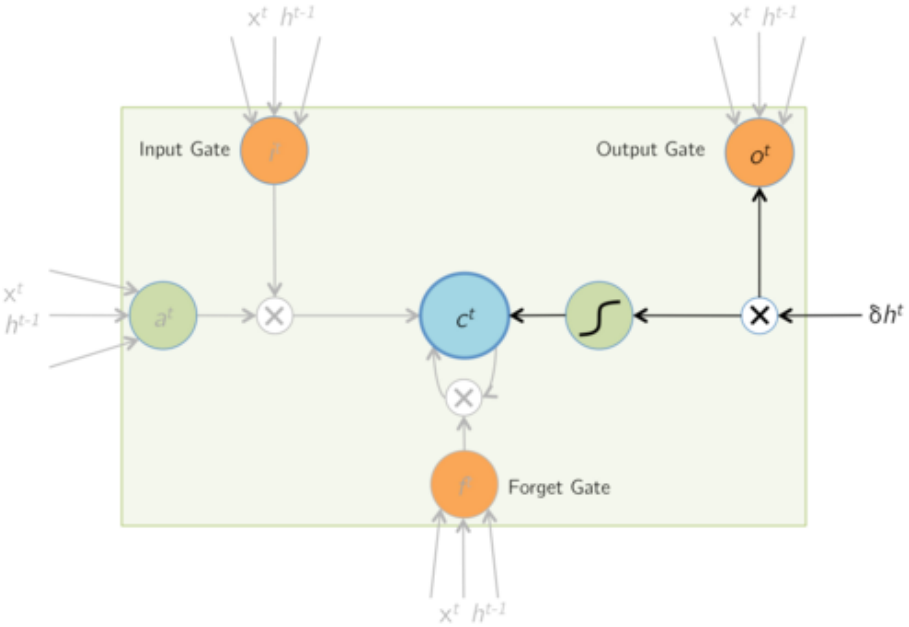
Backward Pass: Unrolled Network

The unrolled network during the backward pass is shown below. All the arrows in the previous slide have now changed their direction. The cell state at time T, c^T receives gradients from h^T as well as the next cell state c^{T+1} . The next few slides focus on computing these two gradients. At any time step T, these two gradients are accumulated before being backpropagated to the layers below the cell and the previous time steps.



There are two parallel **backward** passes

Backward Pass: Output



Forward Pass: $h^t = o^t \odot \tanh(c^t)$

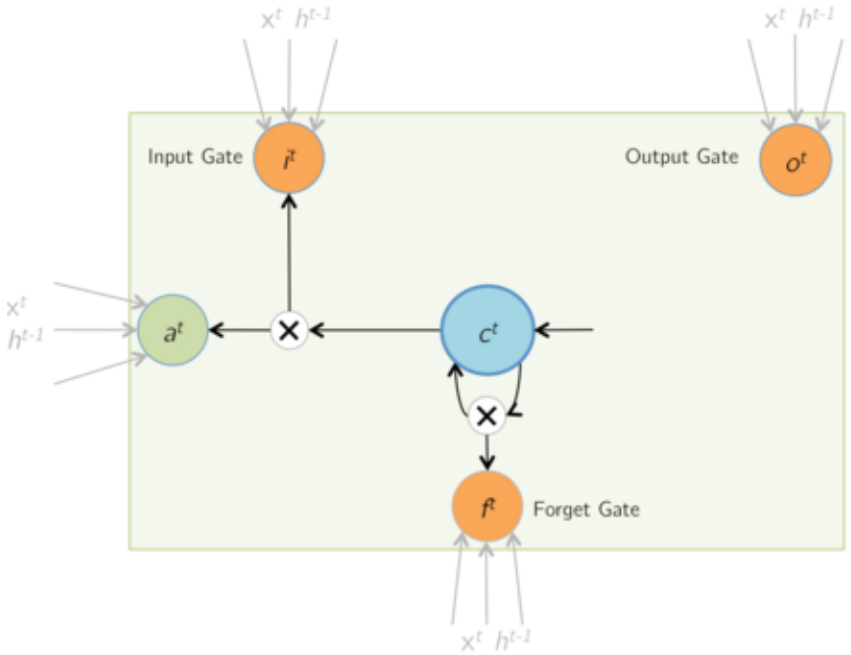
Given $\delta h^t = \dfrac{\partial E}{\partial h^t}$, find $\delta o^t, \delta c^t$

$$\frac{\partial E}{\partial o_i^t} = \frac{\partial E}{\partial h_i^t} \cdot \frac{\partial h_i^t}{\partial o_i^t}$$
$$= \delta h_i^t \cdot \tanh(c_i^t)$$
$$\therefore \delta o^t = \delta h^t \odot \tanh(c^t)$$

$$\frac{\partial E}{\partial c_i^t} = \frac{\partial E}{\partial h_i^t} \cdot \frac{\partial h_i^t}{\partial c_i^t}$$
$$= \delta h_i^t \cdot o_i^t \cdot (1 - \tanh^2(c_i^t))$$
$$\therefore \delta c^t \mathrel{+}= \delta h^t \odot o^t \odot (1 - \tanh^2(c^t))$$

Note that the $+=$ above is so that this gradient is added to gradient from time step $(t+1)$ (calculated on next slide, refer to the gradient accumulation mentioned in the previous slide)

Backward Pass: LSTM Memory Cell Update



Forward Pass: $c^t = i^t \odot a^t + f^t \odot c^{t-1}$

Given $\delta c^t = \frac{\partial E}{\partial c^t}$, find $\delta i^t, \delta a^t, \delta f^t, \delta c^{t-1}$

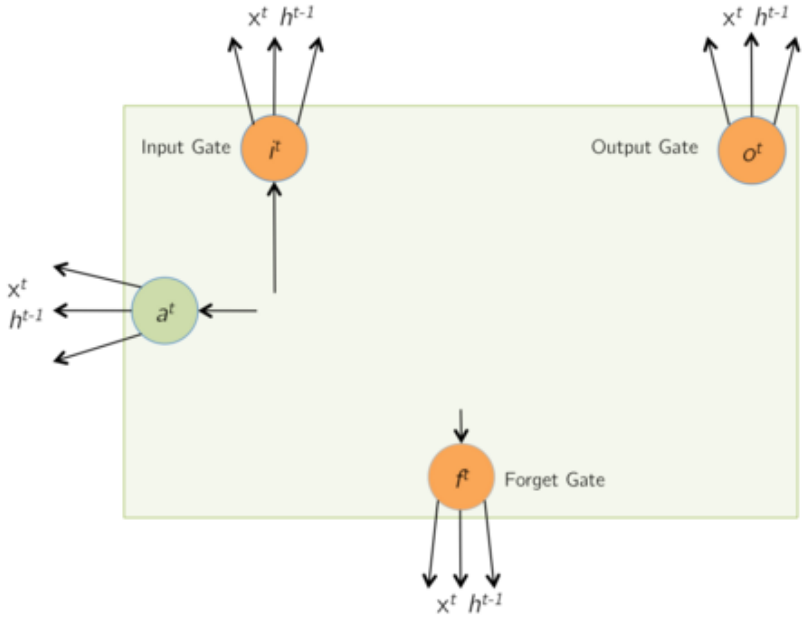
$$\frac{\partial E}{\partial i_i^t} = \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial i_i^t}$$
$$= \delta c_i^t \cdot a_i^t$$
$$\therefore \delta i^t = \delta c^t \odot a^t$$

$$\frac{\partial E}{\partial f_i^t} = \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial f_i^t}$$
$$= \delta c_i^t \cdot c_i^{t-1}$$
$$\therefore \delta f^t = \delta c^t \odot c^{t-1}$$

$$\frac{\partial E}{\partial a_i^t} = \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial a_i^t}$$
$$= \delta c_i^t \cdot i_i^t$$
$$\therefore \delta a^t = \delta c^t \odot i^t$$

$$\frac{\partial E}{\partial c_i^{t-1}} = \frac{\partial E}{\partial c_i^t} \cdot \frac{\partial c_i^t}{\partial c_i^{t-1}}$$
$$= \delta c_i^t \cdot f_i^t$$
$$\therefore \delta c^{t-1} = \delta c^t \odot f^t$$

Backward Pass: Input and Gate Computation - I

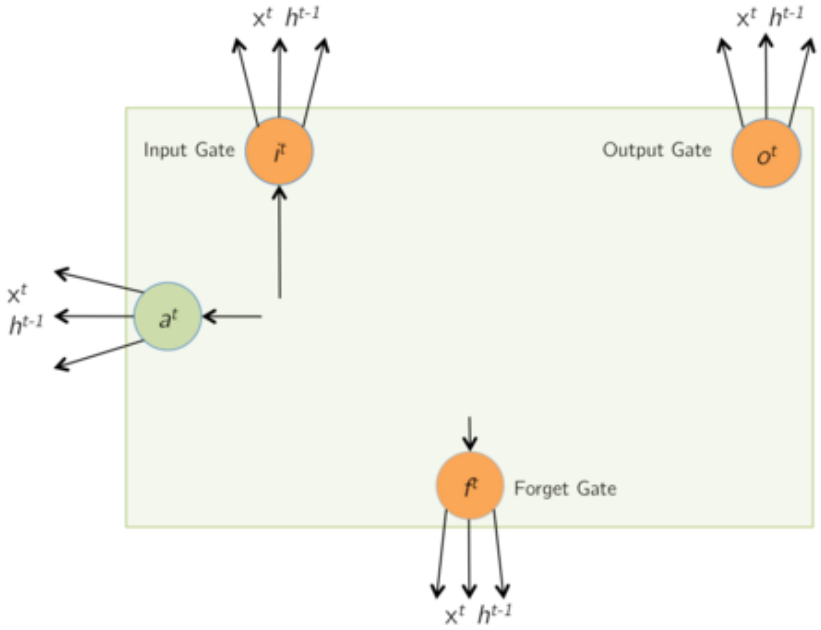


$$\text{Forward Pass: } z^t = \begin{bmatrix} \hat{a}^t \\ \hat{i}^t \\ \hat{f}^t \\ \hat{o}^t \end{bmatrix} = W \times I^t$$

$$\text{Given } \delta a^t, \delta i^t, \delta f^t, \delta o^t, \text{ find } \delta z^t$$

$$\delta \hat{a}^t = \delta a^t \odot (1 - \tanh^2(\hat{a}^t))$$
$$\delta \hat{i}^t = \delta i^t \odot i^t \odot (1 - i^t)$$
$$\delta \hat{f}^t = \delta f^t \odot f^t \odot (1 - f^t)$$
$$\delta \hat{o}^t = \delta o^t \odot o^t \odot (1 - o^t)$$
$$\delta z^t = \left[\delta \hat{a}^t, \delta \hat{i}^t, \delta \hat{f}^t, \delta \hat{o}^t\right]^T$$

Backward Pass: Input and Gate Computation - II



Forward Pass: $z^t = W \times I^t$

Given δz^t, find $\delta W^t, \delta h^{t-1}$

$$\delta I^t = W^T \times \delta z^t$$

$$\text{As } I^t = \begin{bmatrix} x^t \\ h^{t-1} \end{bmatrix},$$

δh^{t-1} can be retrieved from δI^t

$$\delta W^t = \delta z^t \times (I^t)^T$$

Parameter Update

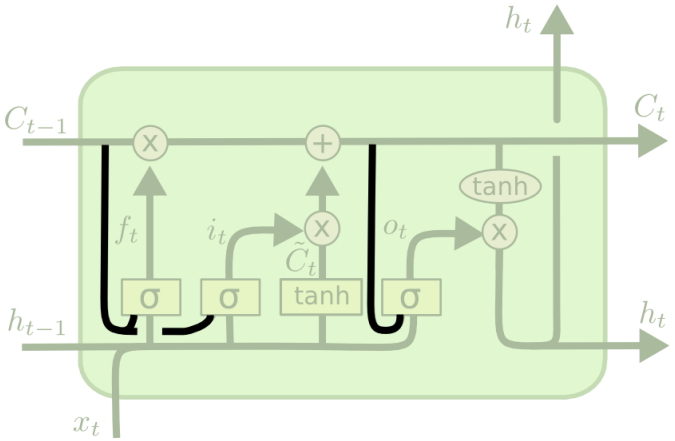
If input x has T time-steps, i.e. $x = [x^1, x^2, \cdots, x^T]$, then

$$\delta W = \sum_{t=1}^{T} \delta W^t$$

W is then updated using an appropriate Stochastic Gradient Descent solver.

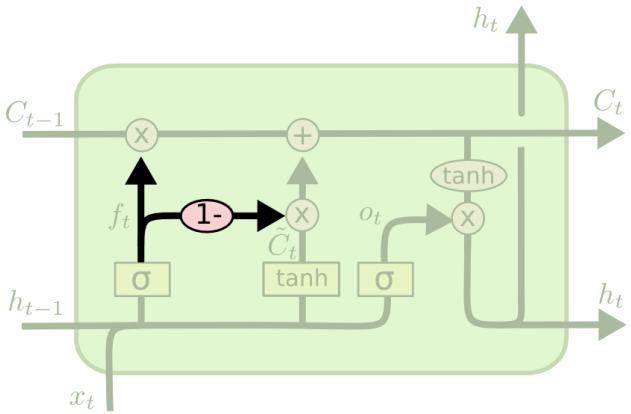
Extended Versions

1. One popular LSTM variant, introduced by Gers & Schmidhuber (2000), is adding "peephole connections." This means that we let the gate layers look at the cell state.



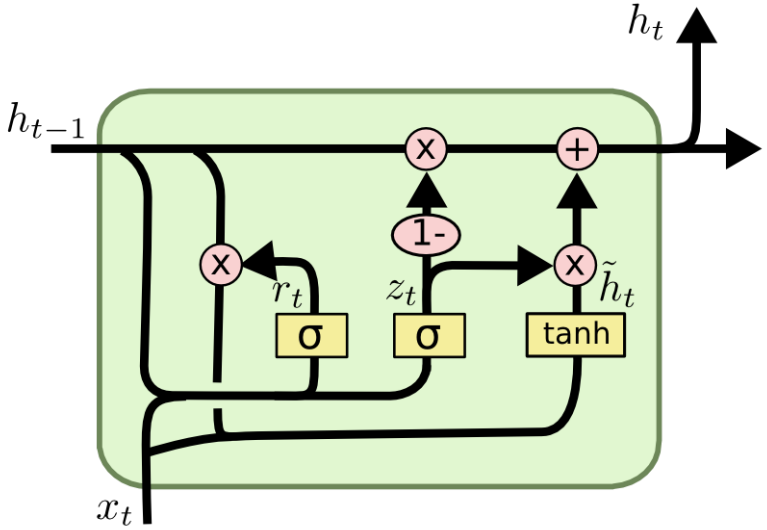
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

2. Another variation is to use coupled forget and input gates. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.



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

3. A slightly more dramatic variation on the LSTM is the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014). It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Thank you

End of Chapter 3