Lecture 25: Recurrent Neural Networks

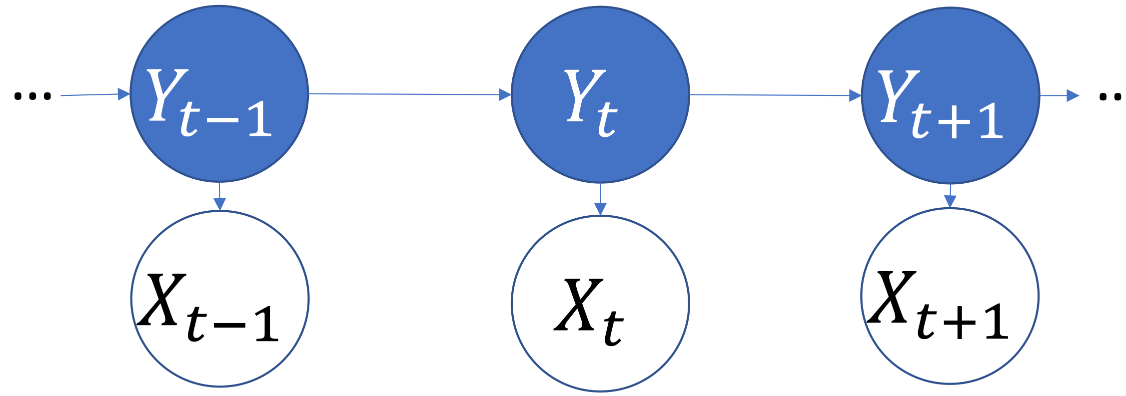
Mark Hasegawa-Johnson

3/2022

Content

- Belief propagation
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

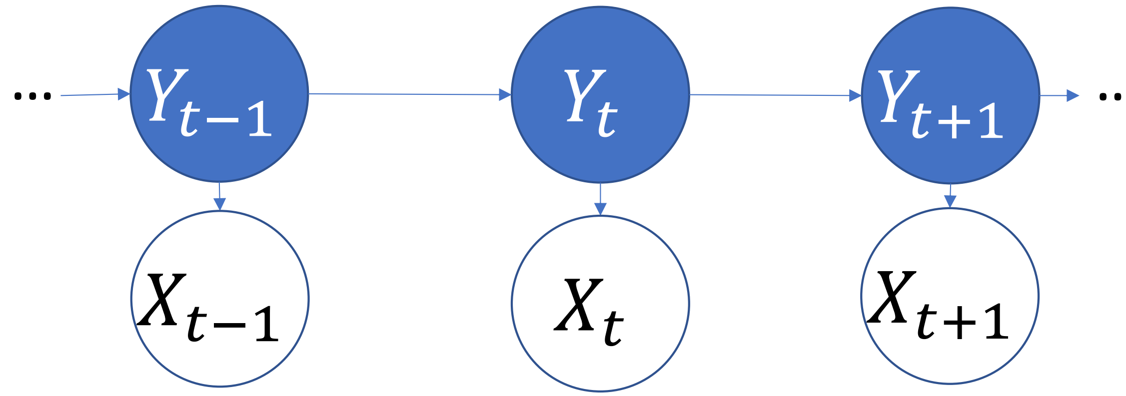
Semantics of Bayesian Networks



Remember the graph semantics of a Bayesian network: Edges denote dependence. This graph means:

$$P(Y_1, X_1, \ldots, Y_T, X_T) = \prod_{t=1}^{T} P(Y_t | Y_{t-1}) P(X_t | Y_t)$$

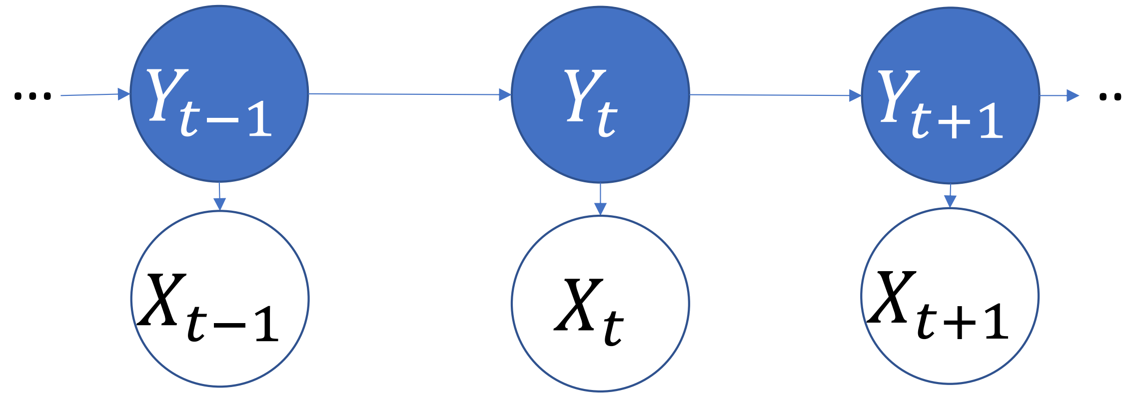
Belief Propagation and Viterbi Algorithm



So far, we've discussed two types of inference.

- Belief propagation computes $P\left(Y_{\text{query}}, X_1, \ldots, X_T\right)$ by repeating the following two steps for every t:
  - Multiply: $P(\ldots, Y_{t-1}=i, X_{t-1}, Y_t=j, X_t) = P(\ldots, Y_{t-1}=i, X_{t-1})P(Y_t=j|Y_{t-1}=i)P(X_t|Y_t=j)$
  - If $t-1 \neq$ query, then Add: $P(\ldots, Y_t=j, X_t) = \sum_i P(\ldots, Y_{t-1}=i, X_{t-1}, Y_t=j, X_t)$
- The Viterbi algorithm finds the most probable sequence $Y_1, \ldots, Y_T$ given $X_1, \ldots, X_T$ by repeating the following two steps for every t:
  - Multiply: $P(\ldots, Y_{t-1}=i, X_{t-1}, Y_t=j, X_t) = v_{i,t-1}P(Y_t=j|Y_{t-1}=i)P(X_t|Y_t=j)$
  - Take the maximum: $v_{j,t} = \max_i P(\ldots, Y_{t-1}=i, X_{t-1}, Y_t=j, X_t)$

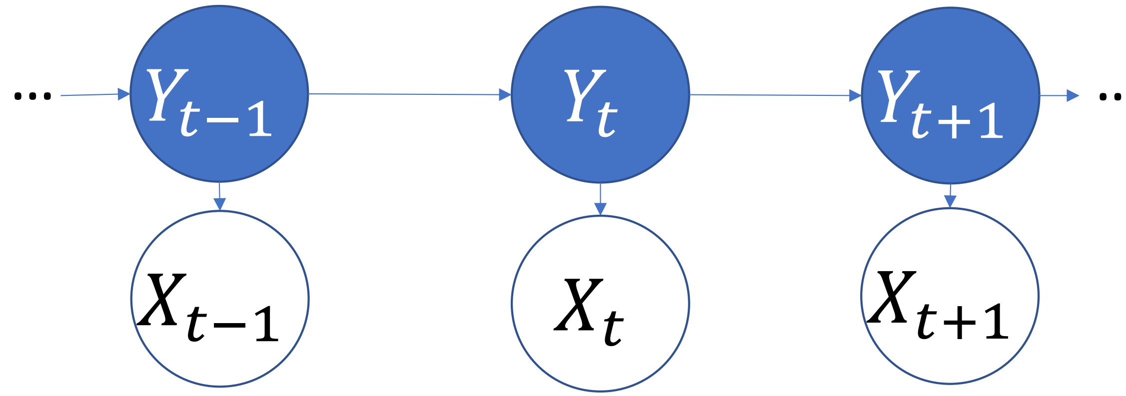
The 3 types of parameters



Both Belief Propagation and Viterbi depend on three index variables:

- j is the index of Y_t. We're trying to compute $P(\dots, Y_t = j, X_t)$
- i is the index of Y_{t-1}. We know that Y_t depends on Y_{t-1} by way of $P(Y_t = j | Y_{t-1} = i)$
- k is the index of X_t. X_t and Y_t are related by $P(X_t = k | Y_t = j)$

Writing them as vectors and matrices



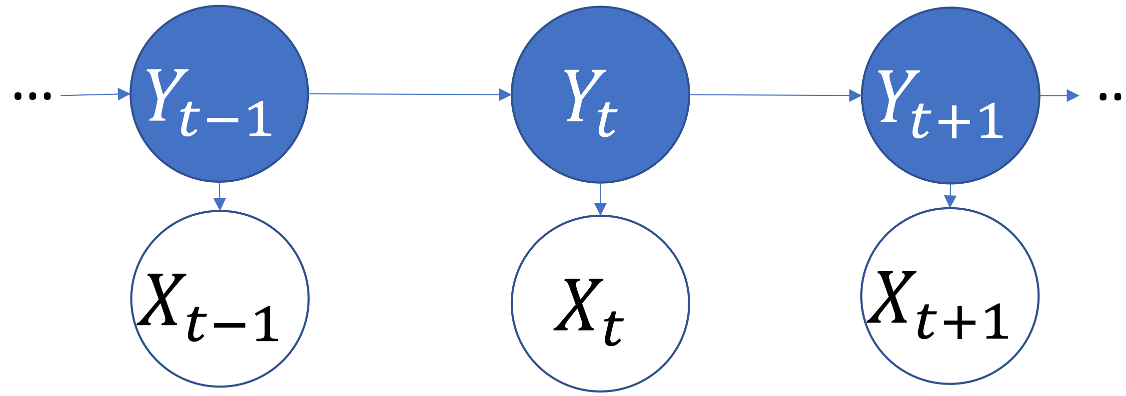
Let’s write these as vectors and matrices:

$$\vec{h}_t = \begin{bmatrix} h_{1,t} \\ \vdots \\ h_{N,t} \end{bmatrix}, h_{j,t} = P(\ldots, Y_t = j, X_t)$$

$$U = \begin{bmatrix} u_{1,1} & \cdots & u_{1,N} \\ \vdots & \ddots & \vdots \\ u_{N,1} & \cdots & u_{N,N} \end{bmatrix}, u_{j,i} = P(Y_t = j | Y_{t-1} = i)$$

$$W = \begin{bmatrix} w_{1,1} & \cdots & w_{1,V+1} \\ \vdots & \ddots & \vdots \\ w_{N,1} & \cdots & w_{N,V+1} \end{bmatrix}, w_{j,k} = P(X_t = k | Y_t = j)$$

Log Belief Propagation



Now, if we write out the whole logarithm of belief propagation:

$$\ln P(\dots, Y_t = j, X_t) = \ln \sum_i P(\dots, Y_{t-1} = i, X_{t-1}) P(Y_t = j | Y_{t-1} = i) + \ln P(X_t = k | Y_t = j)$$

… we discover that we can write it as:

$$\vec{h}_t = \exp\left(\ln U \vec{h}_{t-1} + \ln W \vec{x}_t\right)$$

…where we've defined:

$$\vec{x}_t = \begin{bmatrix} x_{1,t} \\ \vdots \\ x_{V+1,t} \end{bmatrix}, x_{k,t} = \begin{cases} 1 & X_t = k \\ 0 & \text{otherwise} \end{cases}$$

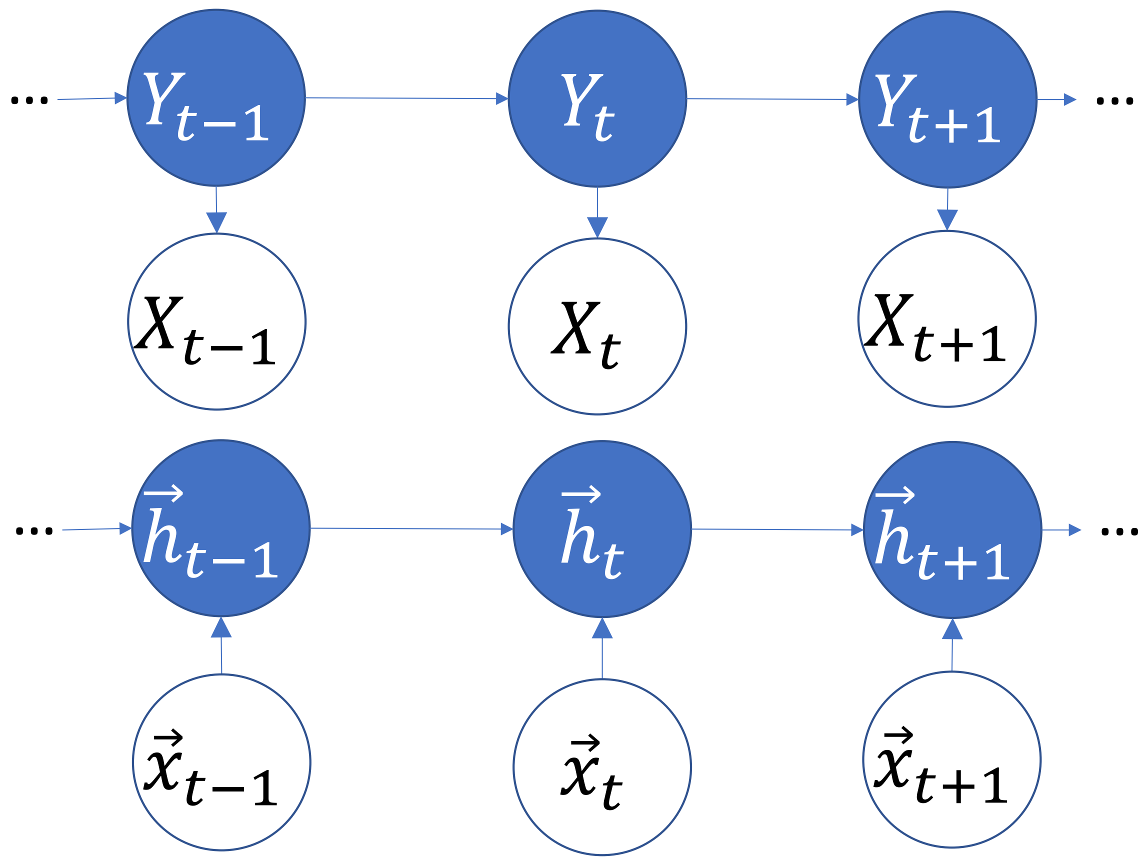
Recurrent neural network

We have turned a Bayesian network with this dependency structure:

$$P(\ldots, Y_{t-1}, X_{t-1}, Y_t, X_t) \\ = P(\ldots, Y_{t-1}, X_{t-1}) P(Y_t | Y_{t-1}) P(X_t | Y_t)$$

Into a neural network with this flowgraph:

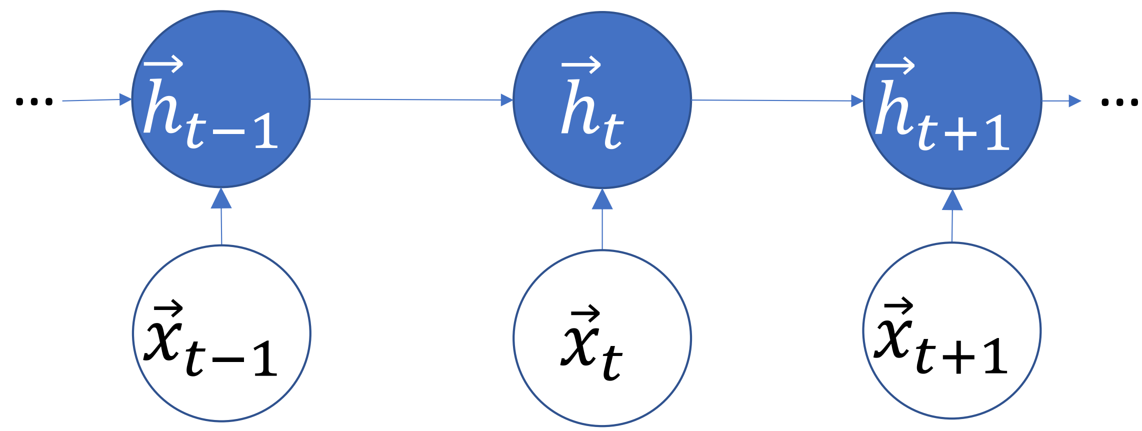
$$\vec{h}_t = \exp\left(\ln U \vec{h}_{t-1} + \ln W \vec{x}_t\right)$$



Content

- Belief propagation
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Recurrent neural network (RNN)

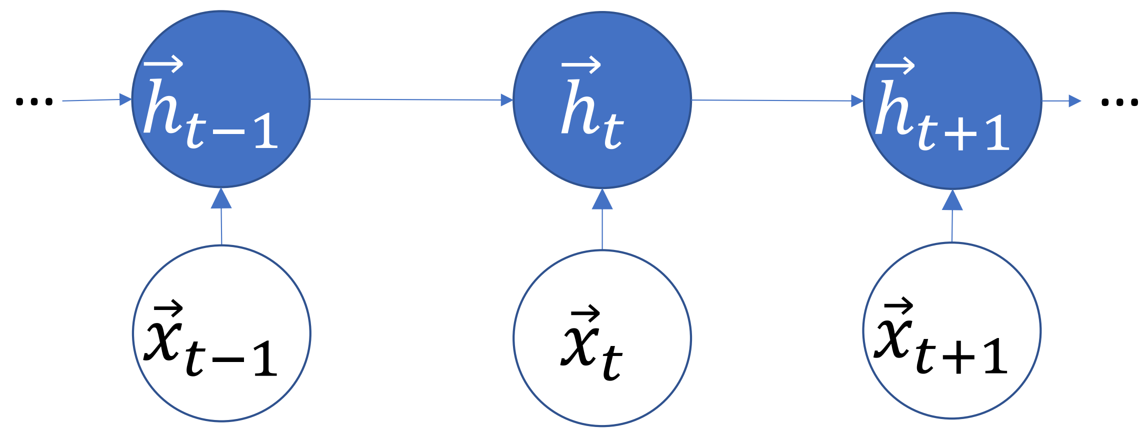


A recurrent neural network (RNN) is a network in which the hidden nodes at time t depend on the input at time t, and on the hidden nodes at time t-1:

$$\vec{h}_t = g\left(U\vec{h}_{t-1}, W\vec{x}_t\right)$$

...where U and W are weight matrices, and g() is some kind of scalar nonlinearity.

Recurrent neural network (RNN)

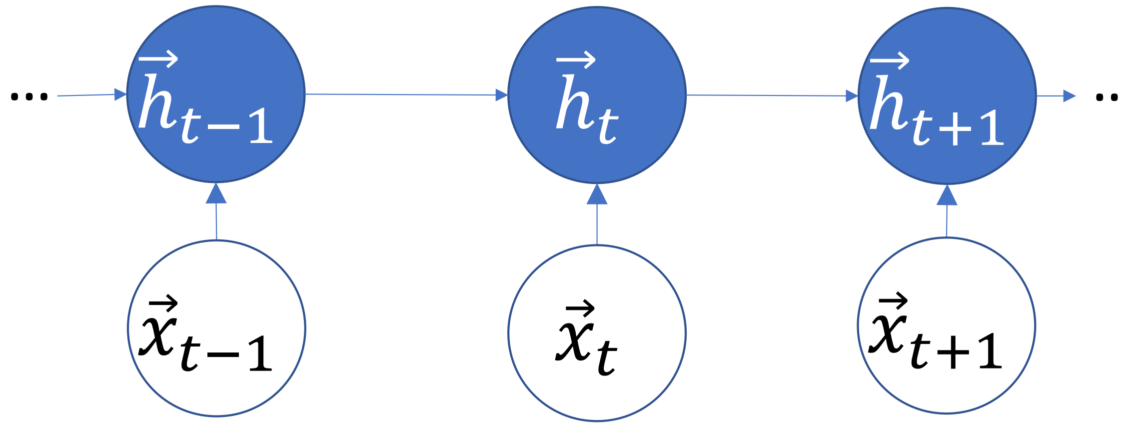


For example, suppose that we have the sentence

"John hit the ball"

... and we want to find each word's part of speech.

Recurrent neural network (RNN)



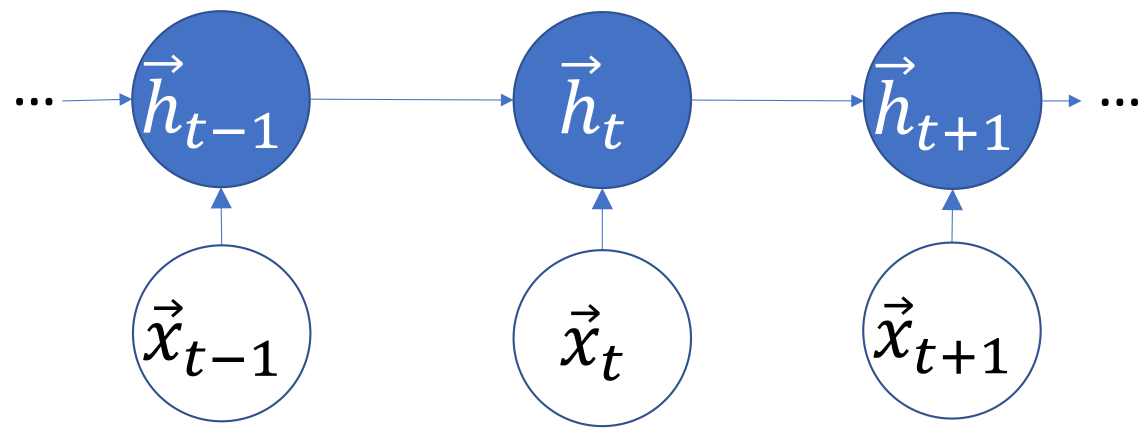
Let's define

$$\vec{x}_t = \begin{bmatrix} 1 \text{ if } X_t = \text{ball} \\ 1 \text{ if } X_t = \text{hit} \\ 1 \text{ if } X_t = \text{John} \\ 1 \text{ if } X_t = \text{the} \end{bmatrix}$$

…so the observation sequence is…

$$\vec{x}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \vec{x}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \vec{x}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \vec{x}_4 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Recurrent neural network (RNN)



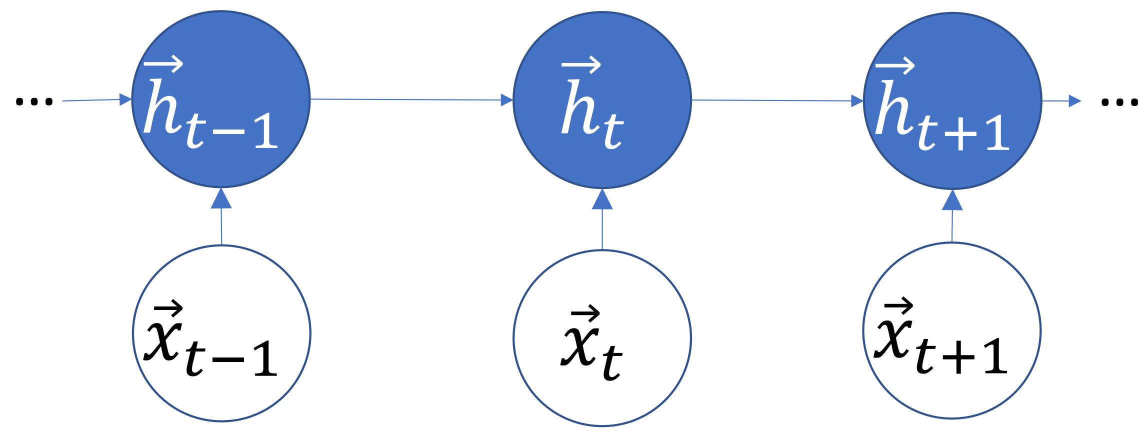
Let's define

$$\vec{h}_t = g\left(U\vec{h}_{t-1} + W\vec{x}_t\right) \approx \begin{bmatrix} P(Y_t = \mathrm{Det}|X_1, \dots, X_t) \\ P(Y_t = \mathrm{Noun}|X_1, \dots, X_t) \\ P(Y_t = \mathrm{Verb}|X_1, \dots, X_t) \end{bmatrix}$$

The approximation is not too bad if we use the following nonlinearity:

$$g(\vec{\xi}) = \begin{bmatrix} \mathrm{softmax}_1(\vec{\xi} - 1) \\ \mathrm{softmax}_2(\vec{\xi} - 1) \\ \mathrm{softmax}_3(\vec{\xi} - 1) \end{bmatrix}, \mathrm{softmax}_j(\vec{\xi} - 1) = \frac{e^{\xi_j - 1}}{\sum_{i=1}^{N} e^{\xi_i - 1}}$$

Recurrent neural network (RNN)

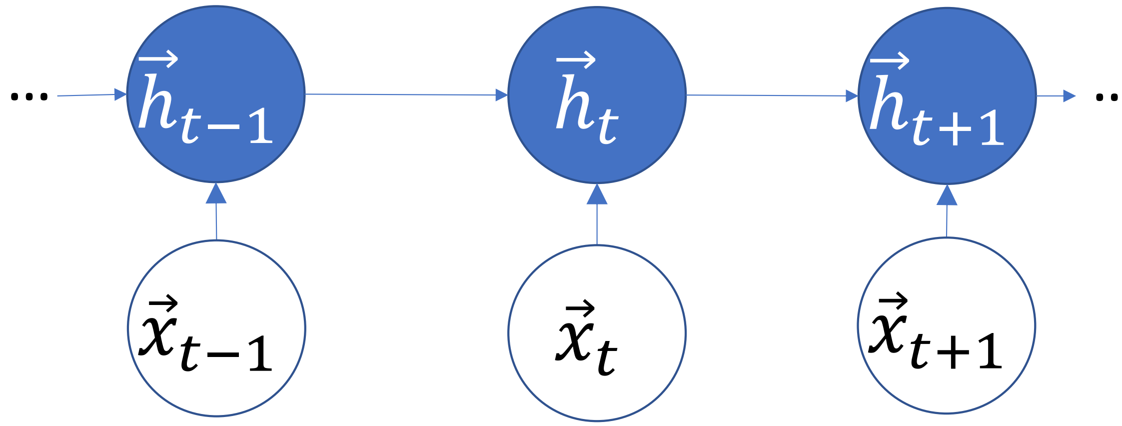


Using the HMM logic, reasonable weight matrices might be:

$$\vec{h}_t \approx \begin{bmatrix} P(Y_t = \text{Det}|X_1, \dots, X_t) \\ P(Y_t = \text{Noun}|X_1, \dots, X_t) \\ P(Y_t = \text{Verb}|X_1, \dots, X_t) \end{bmatrix}, u_{j,i} = P(Y_t = j | Y_{t-1} = i), \qquad U = \frac{1}{10}\begin{bmatrix} 1 & 1 & 8 \\ 8 & 1 & 1 \\ 1 & 8 & 1 \end{bmatrix}$$

$$\vec{x}_t = \begin{bmatrix} 1 \text{ if } X_t = \text{ball} \\ 1 \text{ if } X_t = \text{hit} \\ 1 \text{ if } X_t = \text{John} \\ 1 \text{ if } X_t = \text{the} \end{bmatrix}, w_{j,k} = P(X_t = k | Y_t = j), \qquad W = \frac{1}{100}\begin{bmatrix} 1 & 1 & 1 & 97 \\ 49 & 1 & 49 & 1 \\ 1 & 97 & 1 & 1 \end{bmatrix}$$

Recurrent neural network (RNN)



Plugging it all together, we get

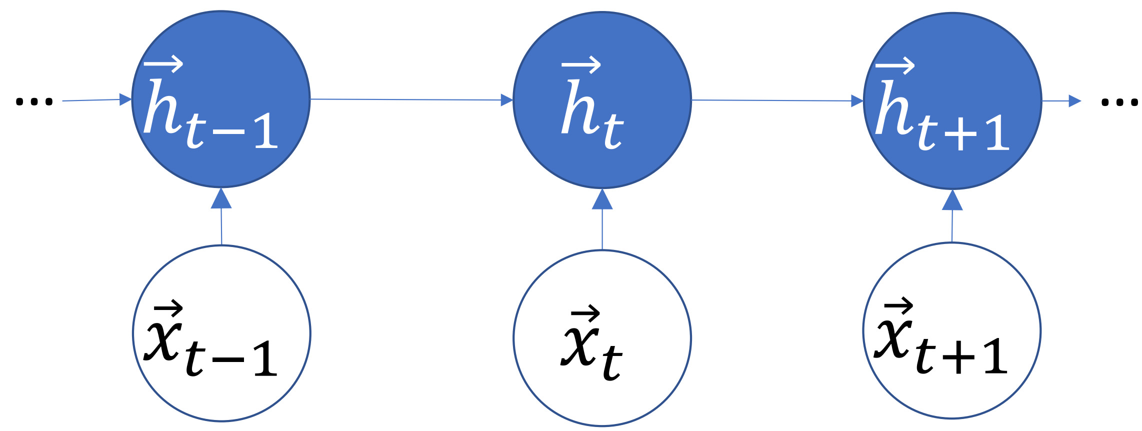
$$\vec{h}_1 = g(U\vec{h}_0 + W\vec{x}_1) = g\left(\begin{bmatrix}0\\0\\0\end{bmatrix} + \frac{1}{100}\begin{bmatrix}1 & 1 & 1 & 97\\49 & 1 & 49 & 1\\1 & 97 & 1 & 1\end{bmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}\right) = g\left(\begin{bmatrix}0.01\\0.49\\0.01\end{bmatrix}\right) = \begin{bmatrix}0.28\\0.44\\0.28\end{bmatrix}$$

$$\vec{h}_2 = g(U\vec{h}_1 + W\vec{x}_2) = g\left(\frac{1}{10}\begin{bmatrix}1 & 1 & 8\\8 & 1 & 1\\1 & 8 & 1\end{bmatrix}\begin{bmatrix}0.28\\0.44\\0.28\end{bmatrix} + \frac{1}{100}\begin{bmatrix}1 & 1 & 1 & 97\\49 & 1 & 49 & 1\\1 & 97 & 1 & 1\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\right) = \begin{bmatrix}0.20\\0.20\\0.60\end{bmatrix}$$

$$\vec{h}_3 = g(U\vec{h}_2 + W\vec{x}_3) = g\left(\frac{1}{10}\begin{bmatrix}1 & 1 & 8\\8 & 1 & 1\\1 & 8 & 1\end{bmatrix}\begin{bmatrix}0.20\\0.20\\0.60\end{bmatrix} + \frac{1}{100}\begin{bmatrix}1 & 1 & 1 & 97\\49 & 1 & 49 & 1\\1 & 97 & 1 & 1\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\right) = \begin{bmatrix}0.63\\0.18\\0.18\end{bmatrix}$$

$$\vec{h}_4 = g(U\vec{h}_3 + W\vec{x}_4) = g\left(\frac{1}{10}\begin{bmatrix}1 & 1 & 8\\8 & 1 & 1\\1 & 8 & 1\end{bmatrix}\begin{bmatrix}0.63\\0.18\\0.18\end{bmatrix} + \frac{1}{100}\begin{bmatrix}1 & 1 & 1 & 97\\49 & 1 & 49 & 1\\1 & 97 & 1 & 1\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix}\right) = \begin{bmatrix}0.24\\0.53\\0.24\end{bmatrix}$$

Recurrent neural network (RNN)



If we interpret $h_{j,t} \approx P(Y_t = j | X_1, \ldots, X_t)$, then we have that

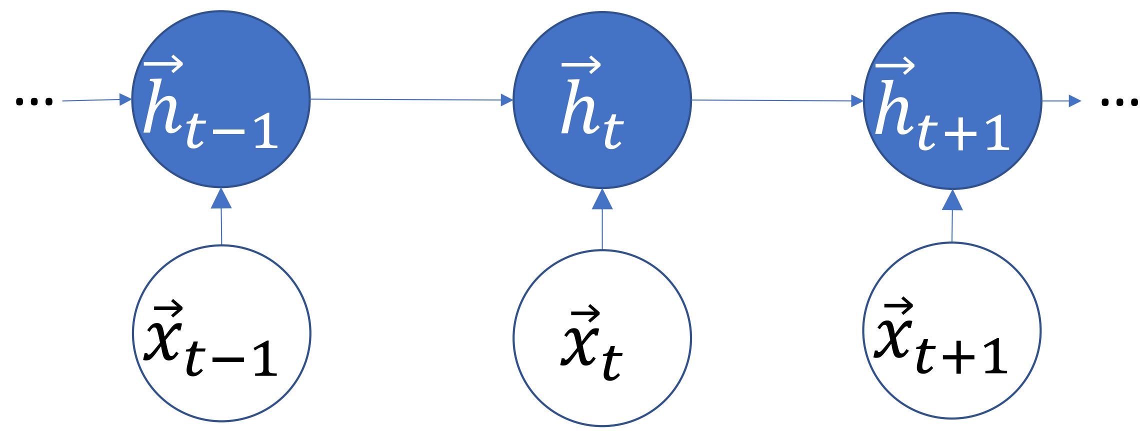
$$P(Y_1 = \text{Noun} | X_1 = \text{John}) \approx 0.44,$$
$$P(Y_2 = \text{Verb} | X_1 = \text{John}, X_2 = \text{hit}) \approx 0.60,$$
$$P(Y_3 = \text{Det} | X_1 = \text{John}, X_2 = \text{hit}, X_3 = \text{the}) \approx 0.63,$$
$$P(Y_4 = \text{Noun} | X_1 = \text{John}, X_2 = \text{hit}, X_3 = \text{the}, X_4 = \text{ball}) \approx 0.53.$$

These probabilities are not very confident --- the RNN is only calculating approximate probabilities, not exact probabilities, so it loses some confidence. But in each case, it got the right answer!

Content

- Belief propagation
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Training an RNN



An RNN is trained using gradient descent, just like any other neural network!

$$u_{j,i} \leftarrow u_{j,i} - \eta \frac{\partial \mathfrak{L}}{\partial u_{j,i}}$$

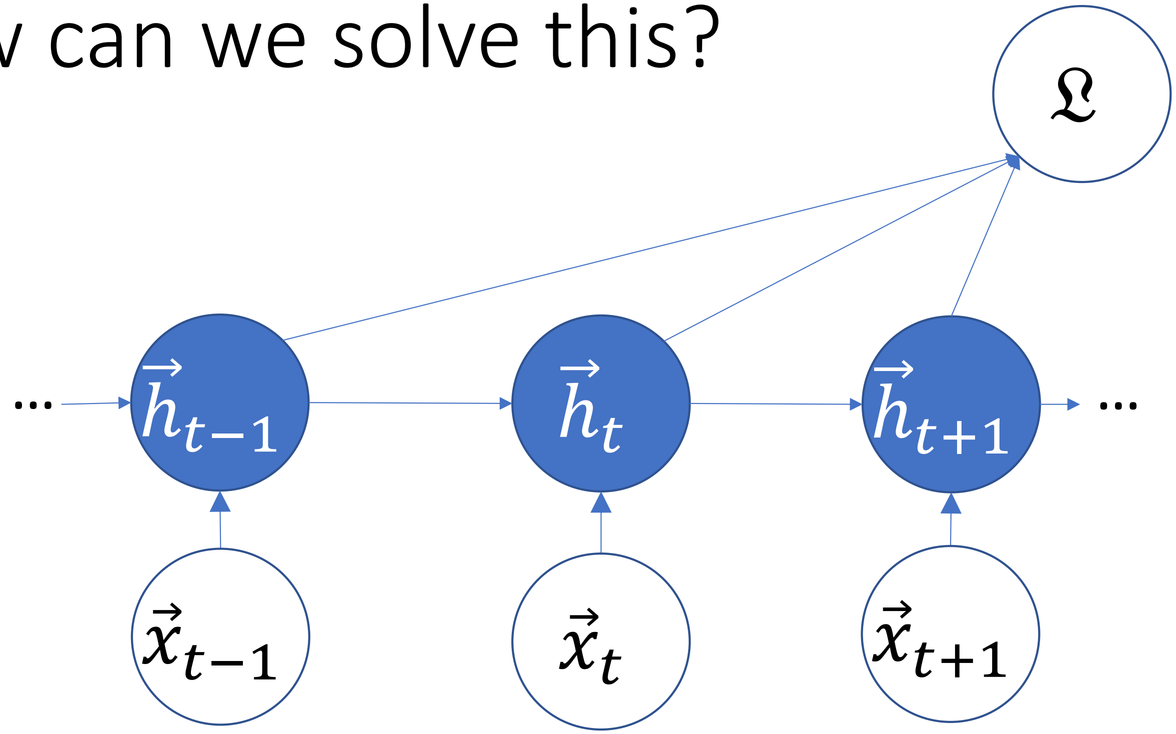
$$w_{j,k} \leftarrow w_{j,k} - \eta \frac{\partial \mathfrak{L}}{\partial w_{j,k}}$$

…where $\mathfrak{L}$ is the loss function, and η is a step size.

Training an RNN: How can we solve this?

The big difference is that now the loss function depends on U and W in many different ways:

- The loss function depends on each of the state vectors $\vec{h}_t$
- Each of the state vectors depends on U and W
- Each of the state vectors ALSO depends on the previous state vector, $\vec{h}_{t-1}$ …
- … which ALSO depends on U and W, and on $\vec{h}_{t-2}$ …
- AUGH!

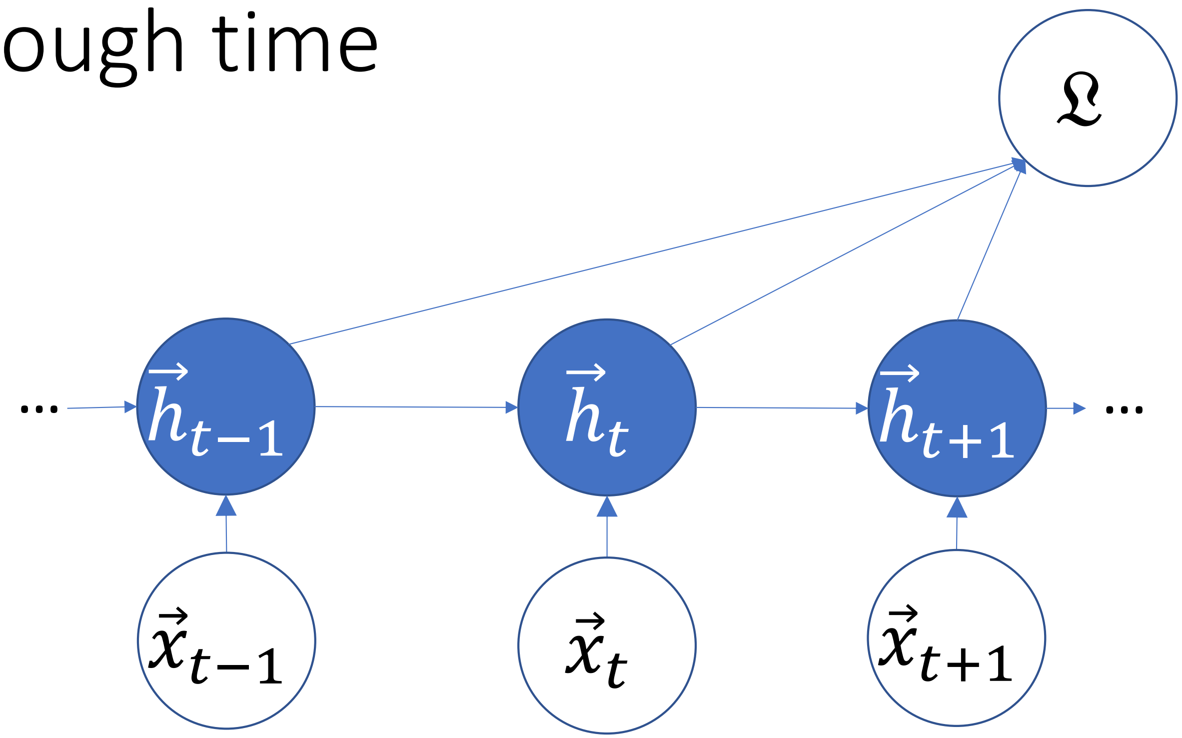


Back-propagation through time

The solution is something called back-propagation through time:

$$\frac{d\mathfrak{L}}{dh_{i,t}} = \frac{\partial\mathfrak{L}}{\partial h_{i,t}} + \frac{d\mathfrak{L}}{dh_{j,t+1}} \frac{\partial h_{j,t+1}}{\partial h_{i,t}}$$

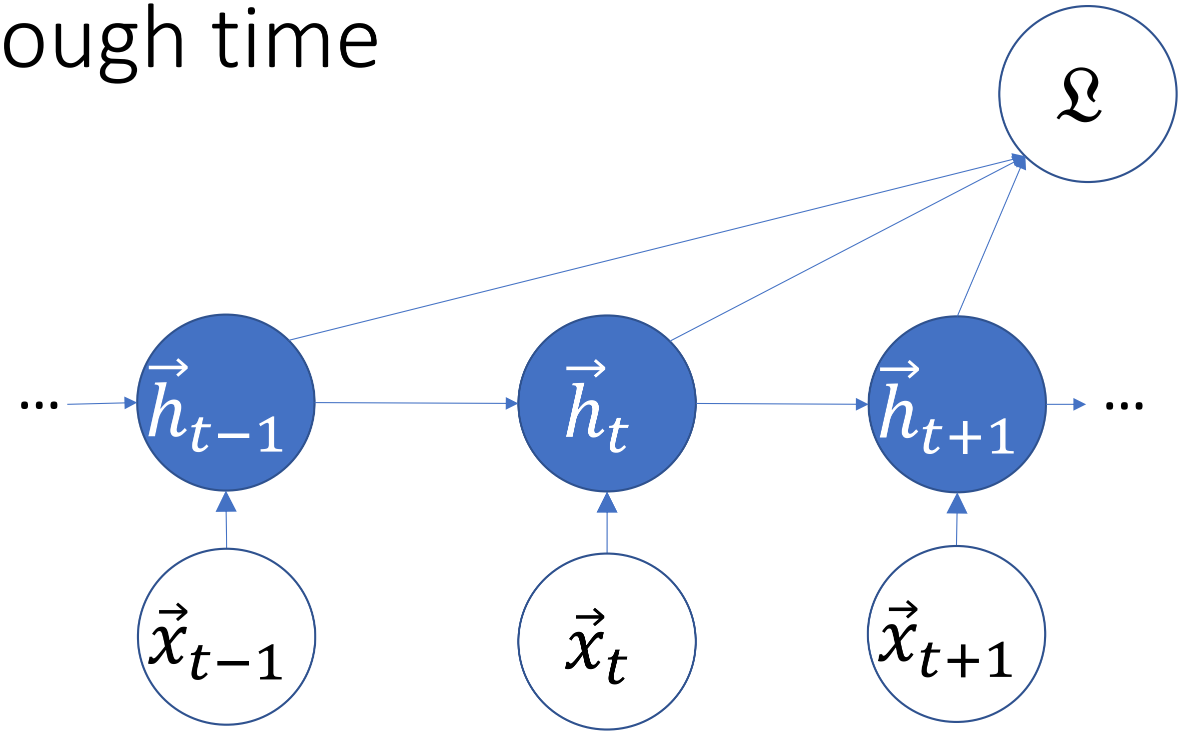
- The first term measures losses caused directly by $h_{i,t}$, for example, if $h_{i,t}$ is wrong.
- The second term measures losses caused indirectly, for example, because $h_{i,t}$ caused $h_{j,t+1}$ to be wrong.



Back-propagation through time

Once we've back-propagated through time, then we add up all the different ways in which the weight matrix affects the output:

$$\frac{d\mathfrak{L}}{du_{j,i}} = \sum_{t=1}^{T} \frac{d\mathfrak{L}}{dh_{i,t}} \frac{\partial h_{i,t}}{\partial u_{j,i}}$$

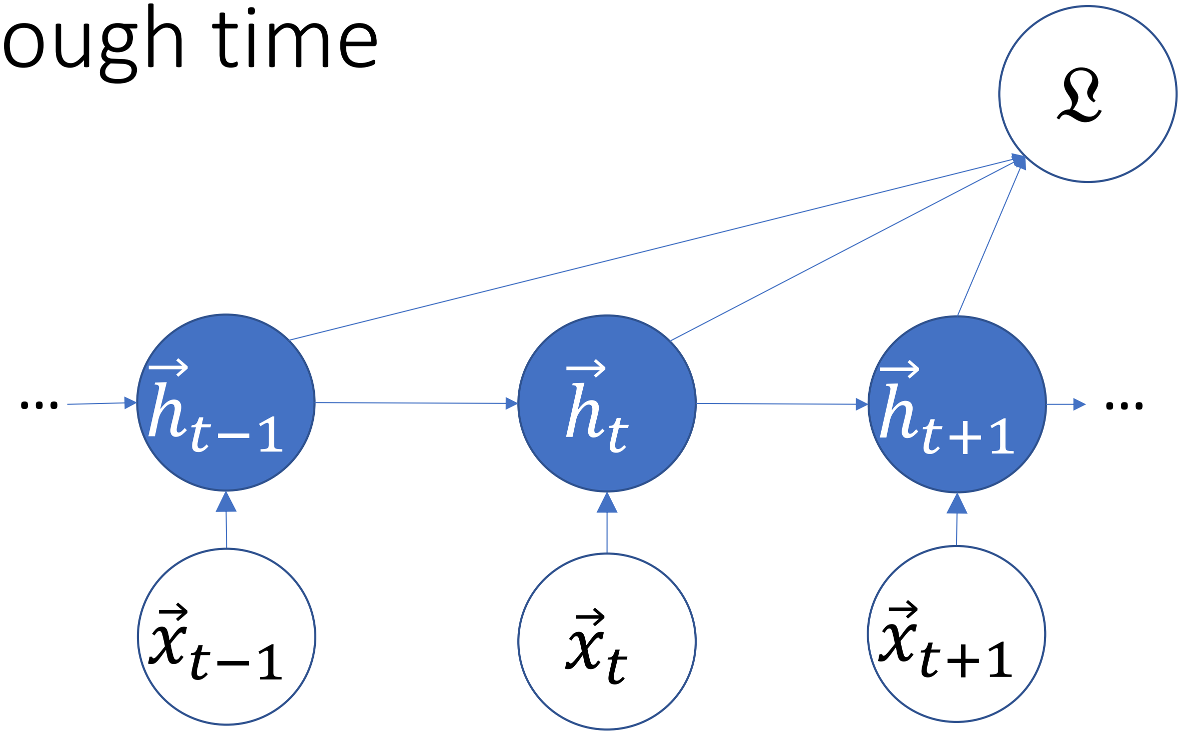


Back-propagation through time

Notice that this is just like training a very deep network!

- Back-propagation through time: back-propagate from time step $t+1$ to time step t
- Back-propagation in a very deep network: back-propagate from layer $l+1$ to layer l

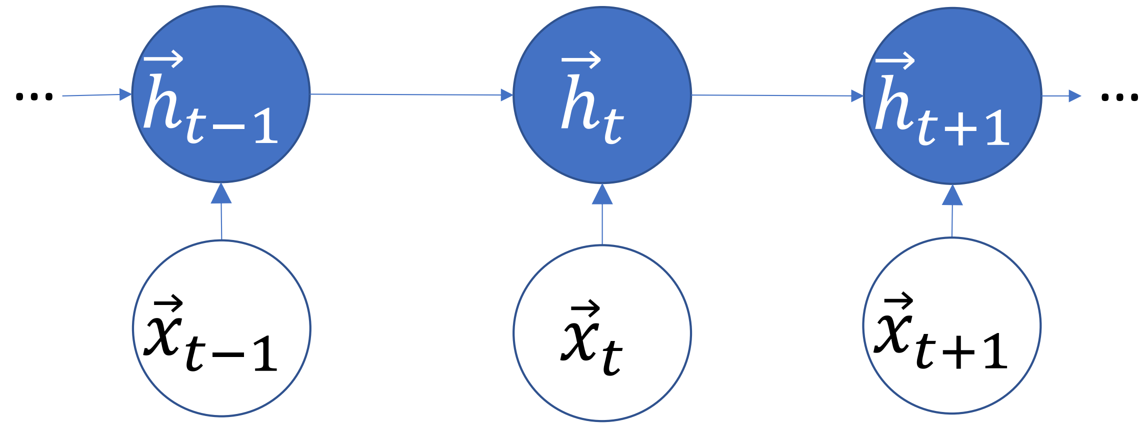
Toolkits like PyTorch use the same code in both cases.

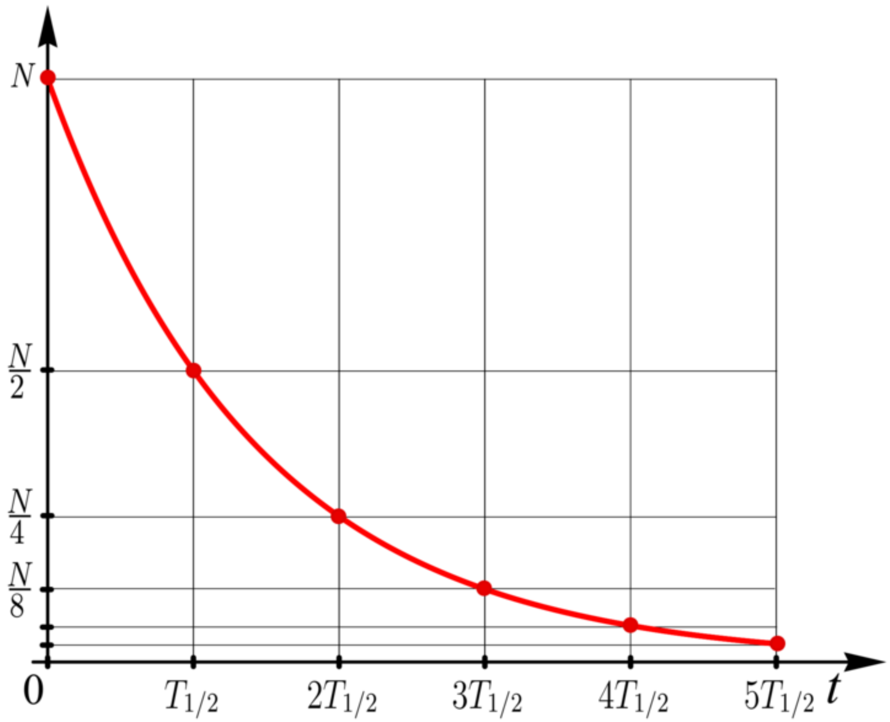


Content

- Log belief propagation
- Recurrent neural networks
- Training a recurrent neural network
- Long short-term memory (LSTM)

Exponential forgetting





Exponential-decay.png. CC-SA-4.0, Svjo, 2017

Regular RNNs have a problem: they forget what they know!

For example, suppose that the feedback matrix is $U = \left(\frac{1}{2}\right)$, so that $\vec{h}_t = \left(\frac{1}{2}\right)\vec{h}_{t-1}$.

Then the state vector decays as $\left(\frac{1}{2}\right)^t$!

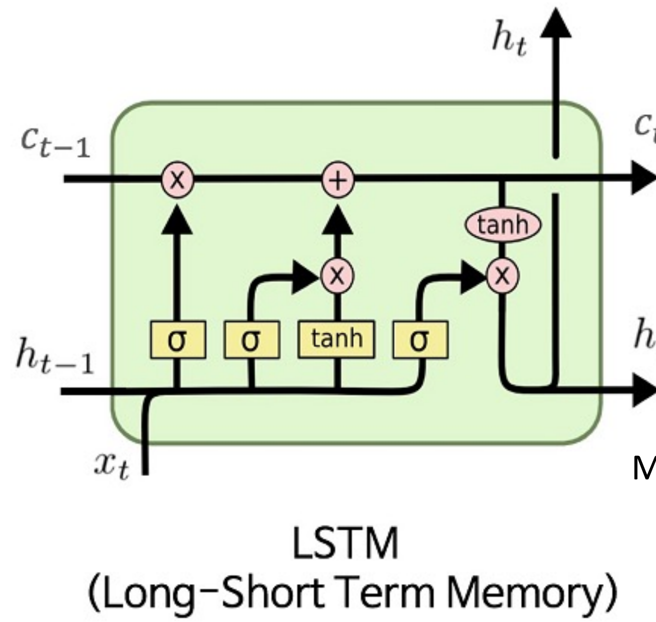
Long-Short Term Memory (LSTM)

A Long-Short Term Memory network (LSTM) solves the exponential forgetting problem using something called a gate.

Remember that a normal RNN computes

$$\vec{h}_t = g\left(U\vec{h}_{t-1} + W\vec{x}_t\right)$$

…so if $\mathrm{U} = \left(\frac{1}{2}\right)$, and if $g(\cdot)$ is linear, then $\vec{h}_t = \left(\frac{1}{2}\right)^t$.

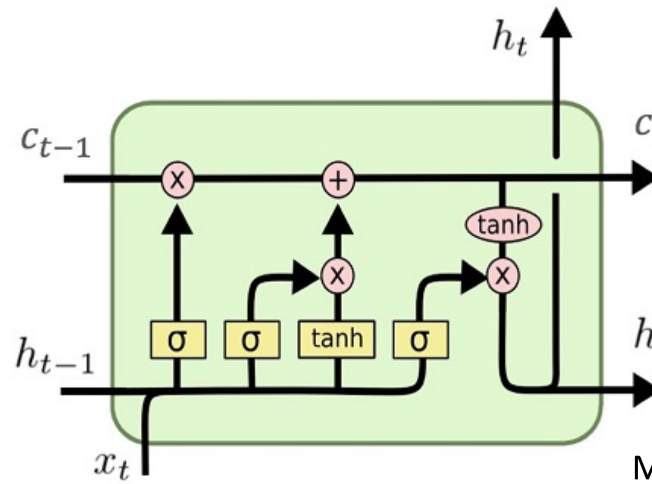


Long-Short Term Memory (LSTM)

An LSTM computes

$$\vec{c}_t = f_t \vec{c}_{t-1} + i_t \vec{x}_t$$

This is just like a regular RNN, except that now, f_t and i_t are not constant. They are adjusted, depending on what the LSTM sees in the input.



LSTM
(Long-Short Term Memory)

Long-Short Term Memory (LSTM)

An LSTM computes

$$\vec{c}_t = f_t \vec{c}_{t-1} + i_t \vec{x}_t$$

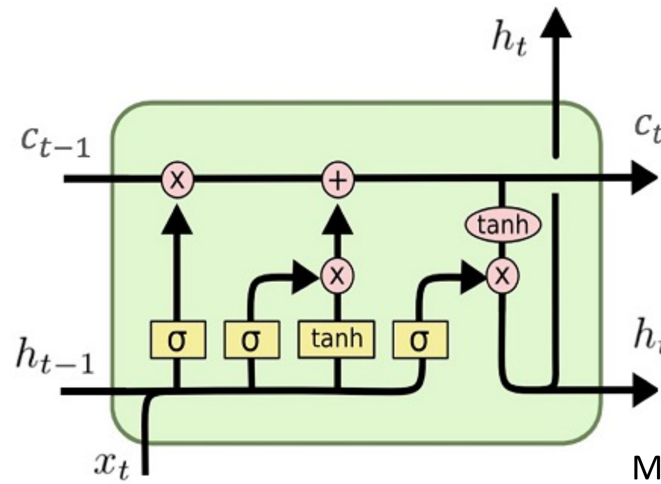
f_t and i_t are called the "forget gate" and the "input gate," respectively. They are computed as

$$f_t = \sigma\left(U_f \vec{h}_{t-1} + W_f \vec{x}_t\right)$$

$$i_t = \sigma\left(U_i \vec{h}_{t-1} + W_i \vec{x}_t\right)$$

…where $\sigma(\cdot)$ is the logistic sigmoid function. Remember that $0 < \sigma(\cdot) < 1$. So:

- If the LSTM wants to remember what it knows, then it will choose $f_t \approx 1$.
- If the LSTM wants to forget what it knows, then it will choose $f_t \approx 0$.



LSTM
(Long−Short Term Memory)

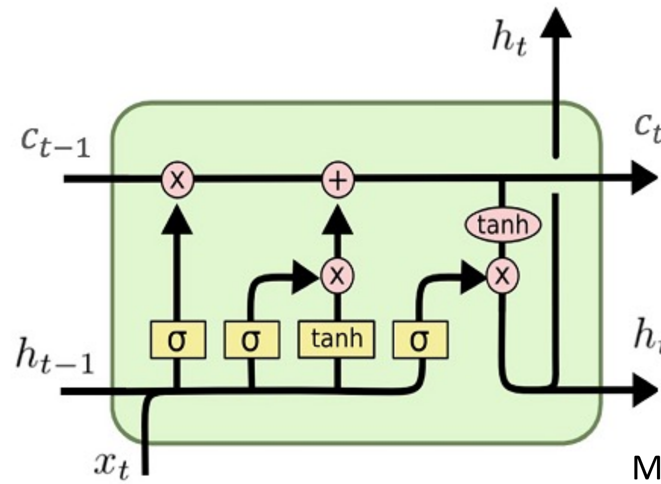
Long-Short Term Memory (LSTM)

$$f_t = \sigma\left(U_f \vec{h}_{t-1} + W_f \vec{x}_t\right)$$
$$i_t = \sigma\left(U_i \vec{h}_{t-1} + W_i \vec{x}_t\right)$$

In order to decide whether to remember what it knows, the LSTM compares $U_f \vec{h}_{t-1}$ to $W_f \vec{x}_t$.

Before it does that, it decides whether it needs to make such a comparison: $\vec{h}_{t-1}$ is equal to the previous time step's memory cell, multiplied by an "output gate" o_{t-1}:

$$\vec{h}_t = o_t \vec{c}_t$$
$$o_t = \sigma\left(U_o \vec{h}_{t-1} + W_o \vec{x}_t\right)$$



LSTM
(Long−Short Term Memory)

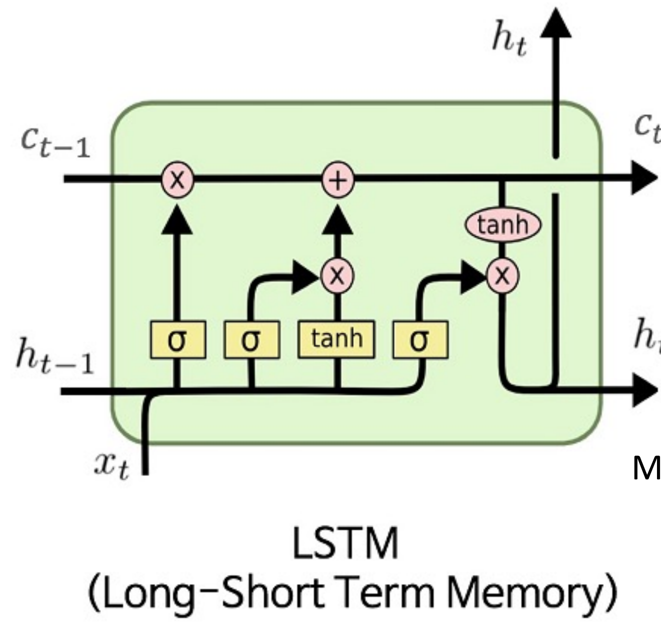
Long-Short Term Memory (LSTM)

An LSTM replaces the one equation of a normal RNN:

$$\vec{h}_t = g\left(U\vec{h}_{t-1} + W\vec{x}_t\right)$$

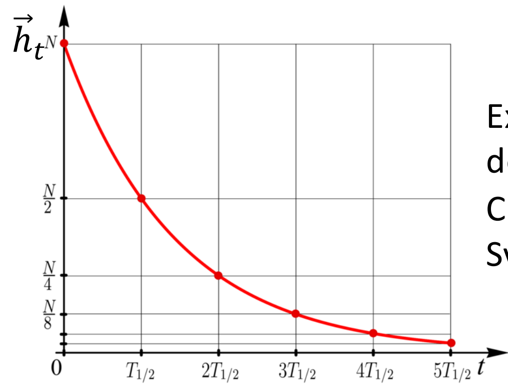
…with these five equations:

- **Forget Gate:** $f_t = \sigma\left(U_f\vec{h}_{t-1} + W_f\vec{x}_t\right)$
- **Input Gate:** $i_t = \sigma\left(U_i\vec{h}_{t-1} + W_i\vec{x}_t\right)$
- **Output Gate:** $o_t = \sigma\left(U_o\vec{h}_{t-1} + W_o\vec{x}_t\right)$
- **Cell:** $\vec{c}_t = f_t\vec{c}_{t-1} + i_t\vec{x}_t$
- **Output:** $\vec{h}_t = o_t\vec{c}_t$



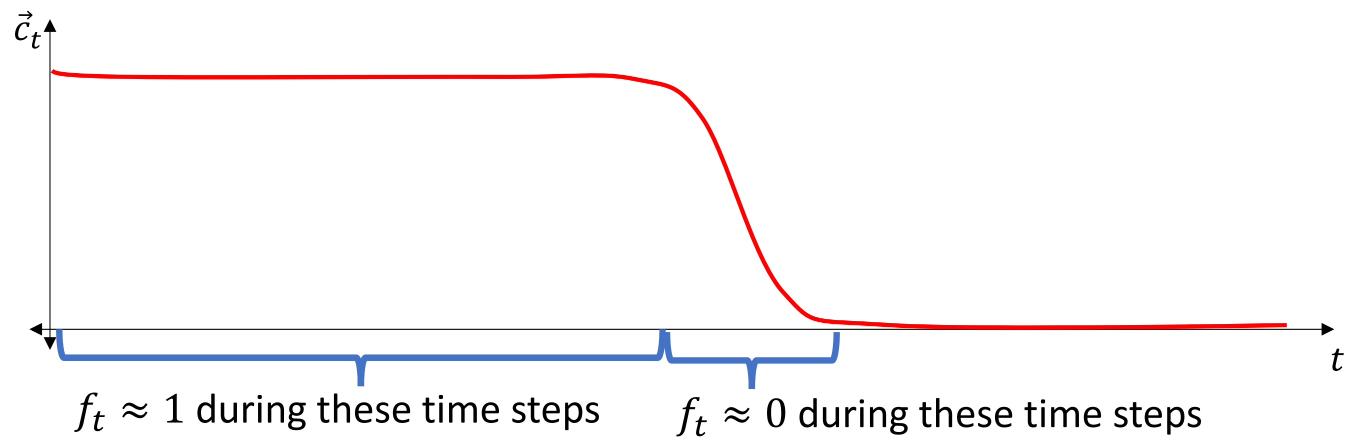
LSTM: Remember when you want to remember, forget when you want to forget

Remember that an RNN tends to forget exponentially, like this:



Exponential-decay.png. CC-SA-4.0, Svjo, 2017

An LSTM forgets more like this:



Content

- Belief propagation

$$P(\dots,Y_{t-1},X_{t-1},Y_t,X_t) = P(\dots,Y_{t-1},X_{t-1})P(Y_t|Y_{t-1})P(X_t|Y_t)$$

- Recurrent neural networks

$$\vec{h}_t = g\left(U\vec{h}_{t-1}, W\vec{x}_t\right)$$

- Training a recurrent neural network

$$\frac{d\mathfrak{L}}{dh_{i,t}} = \frac{\partial\mathfrak{L}}{\partial h_{i,t}} + \frac{d\mathfrak{L}}{dh_{j,t+1}}\frac{\partial h_{j,t+1}}{\partial h_{i,t}}$$

- Long short-term memory (LSTM)

$$\vec{c}_t = f_t\vec{c}_{t-1} + i_t\vec{x}_t$$