Sequence Modeling: Recurrent and Recursive Nets (part 1)

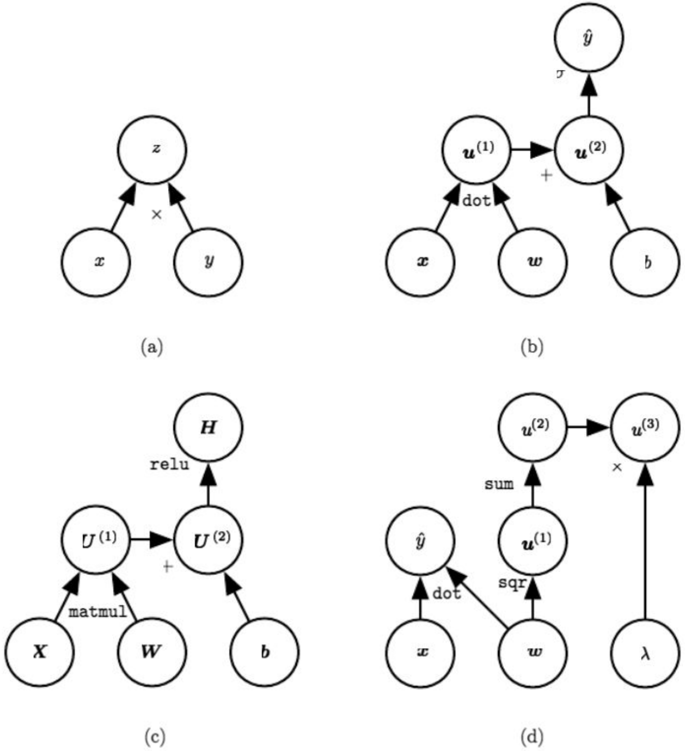
M. Sohaib Alam

12 June, 2017

Deep Learning Textbook Study Meetup Group

Computational graphs

$z = xy$



$$\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right).$$

$$\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}.$$

Processes in (a)-(c), and

$\lambda \sum_i w_i^2$.

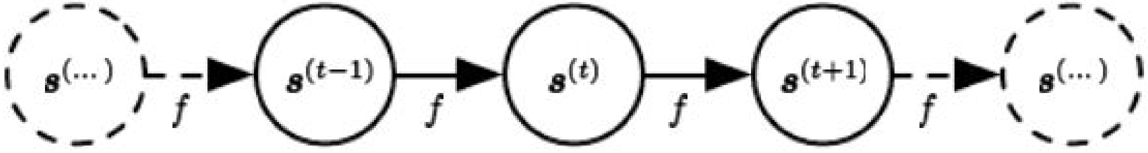
Unfolding computational graphs

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}),$$

$s^{(t)}$: state of the system at some (time) index t; recurrent

Unfolding:

$$\begin{aligned}\boldsymbol{s}^{(3)} &= f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) \\ &= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})\end{aligned}$$



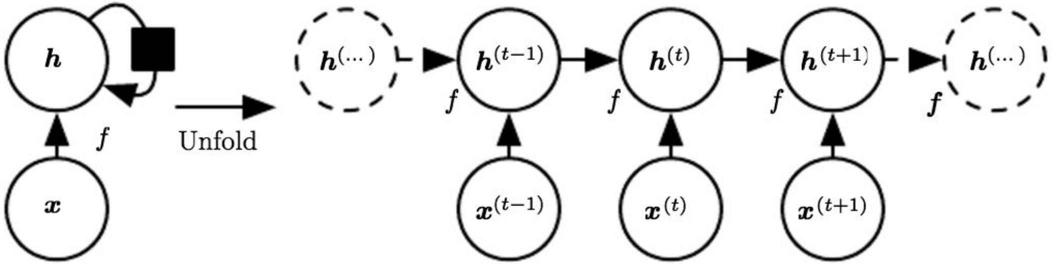
Unfolding computational graphs

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}),$$

$s^{(t)}$: state of system driven by external signal $x^{(t)}$

In particular, $s^{(t)}$ can be a hidden layer $h^{(t)}$

Circuit diagram of RNN with no output; black square indicates delay of single time step (Recurrent graph)



Same network unfolded; each node associated with a single time step (Unfolded graph)

Unfolding computational graphs

Unfolding a recurrence which depends on the entire previous input

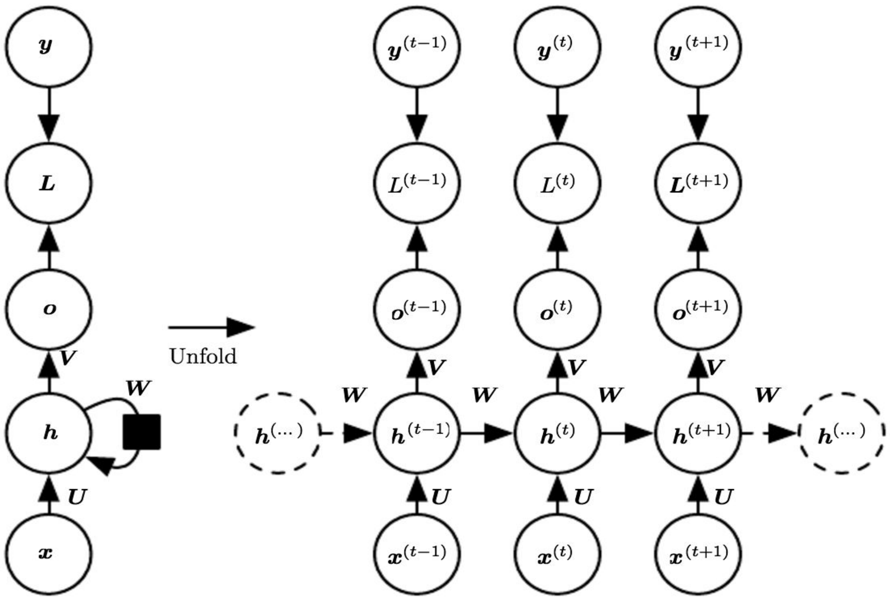
$$
\begin{aligned}
\boldsymbol{h}^{(t)} =& g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \\
=& f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})
\end{aligned}
$$

allows us to factorize a function such as $g^{(t)}$, which takes the entire past sequence as input, and factorize it into repeated applications of a single function f, and introduces several advantages:

- Input size is fixed, specified in terms of transition from one state to the next,
- Possible to use the same transition function with shared parameters at every time step,
- Generalizable to sequence lengths that did not appear in training data

Recurrent Neural Networks: Common Architectures

Several design patterns, for example: recurrence between hidden units, producing an output at each time step

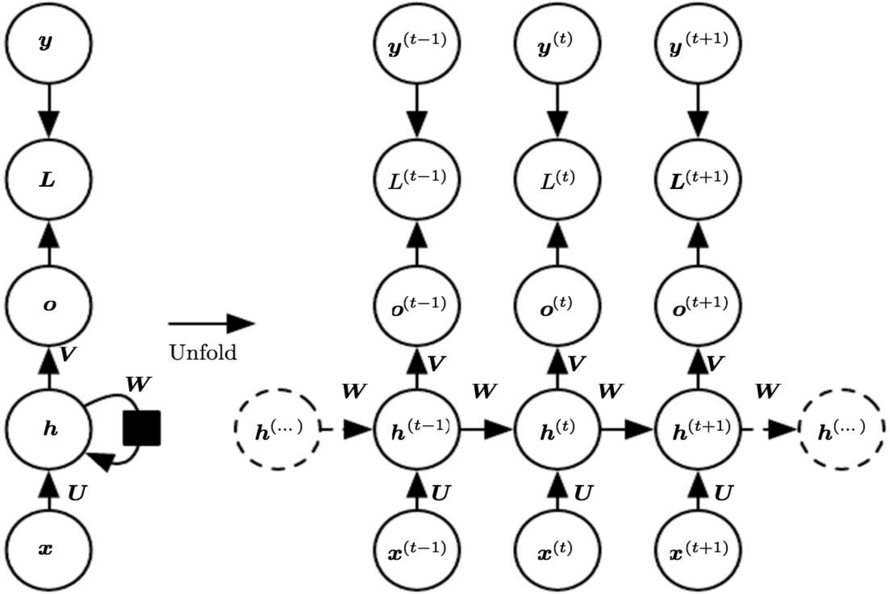


$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}$$

where
b, c: bias vectors
U: input-to-hidden weight matrix
V: hidden-to-output weight matrix
W: hidden-to-hidden weight matrix

Loss L measures how far each **o** is from the corresponding training target **y**

Recurrent Neural Networks: Common Architectures



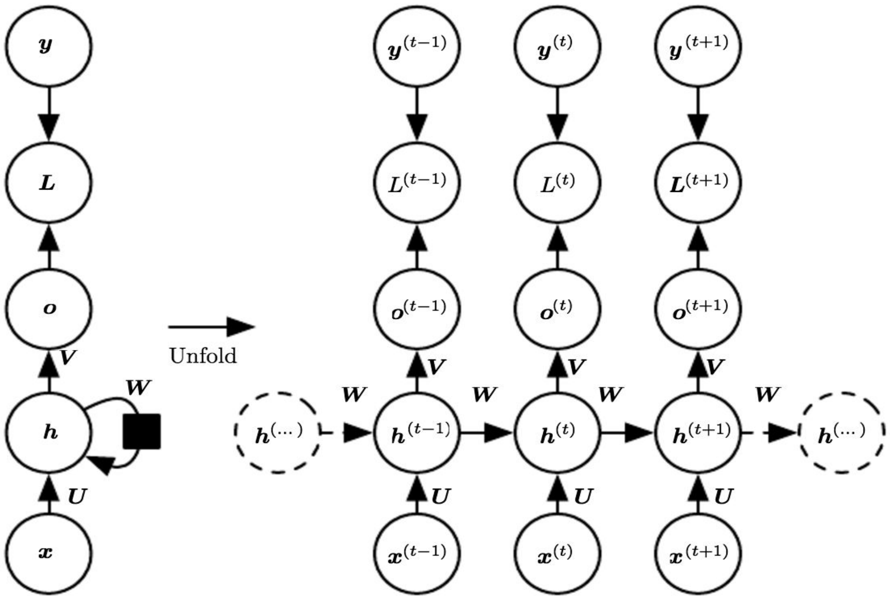
Maps an input sequence to an output sequence of the same length.

Total loss for training input sequence of values **x** with an output sequence **y** is given by the sum of log probabilities (cross-entropies):

$$L\left(\{\boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(\tau)}\},\{\boldsymbol{y}^{(1)},\dots,\boldsymbol{y}^{(\tau)}\}\right)$$
$$=\sum_t L^{(t)}$$
$$=-\sum_t \log p_{\text{model}}\left(y^{(t)} \mid \{\boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(t)}\}\right),$$

where $\log(p_{model}\,(y^{(t)}|\{x^{(1)},\dots,x^{(t)}\})$ is given by comparing $y^{(t)}$ with the actual output $yhat^{(t)}$

Recurrent Neural Networks: Common Architectures



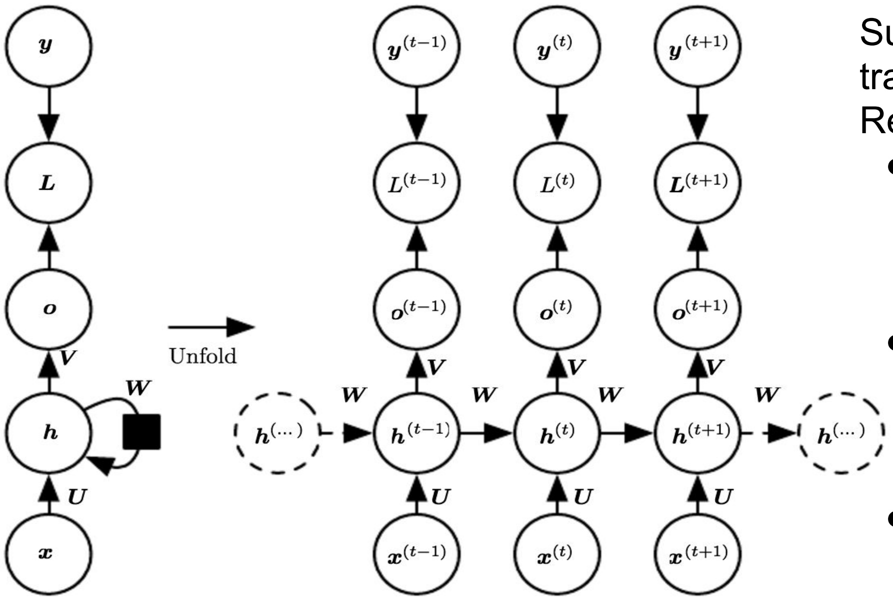
Gradient computation involves:

Performing an inherently sequential forward propagation, with runtime O(T), where T is the number of time steps.

States computed in the forward pass must be stored until they are reused during the backward pass, so memory cost is also O(T).

Back-Propagation Through Time (BPTT): Back-propagation algorithm applied to unrolled graph with O(T) cost

Recurrent Neural Networks: Common Architectures

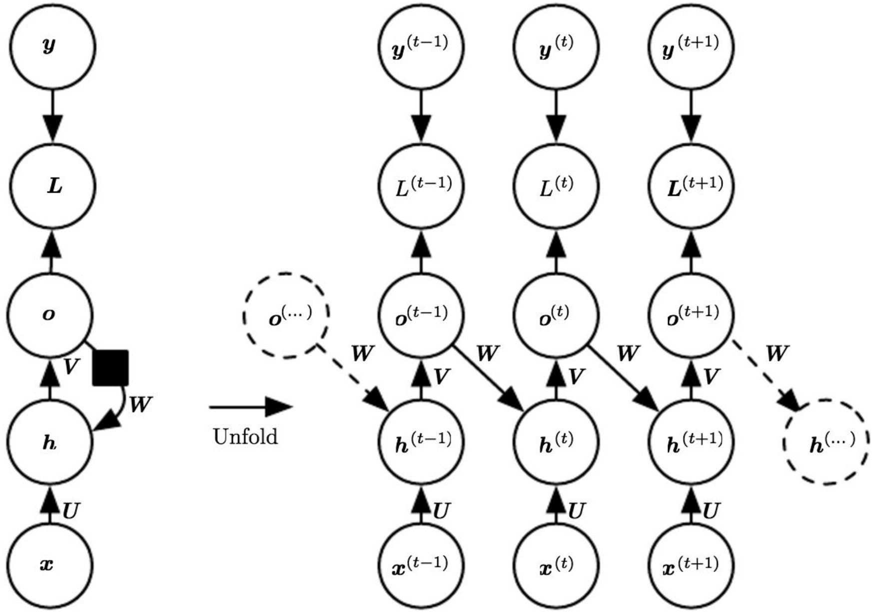


Such a network of finite size, though expensive to train, can compute anything a Turing machine can.
Refs:

- Siegelmann and Sontag, 1991: "Turing Computability with Neural Nets", http://people.cs.georgetown.edu/~cnewport/teaching/cosc844-spring17/pubs/nn-tm.pdf;
- Siegelmann and Sontag, 1995: "On the Computational Power of Neural Nets", http://www.sciencedirect.com/science/article/pii/S0022000085710136
- Hyotyniemi, 1996: "Turing Machines are Recurrent Neural Networks", http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.5161

Recurrent Neural Networks: Common Architectures

Another example: Recurrence only from output at one time step to hidden units at next time step, producing output at each time step

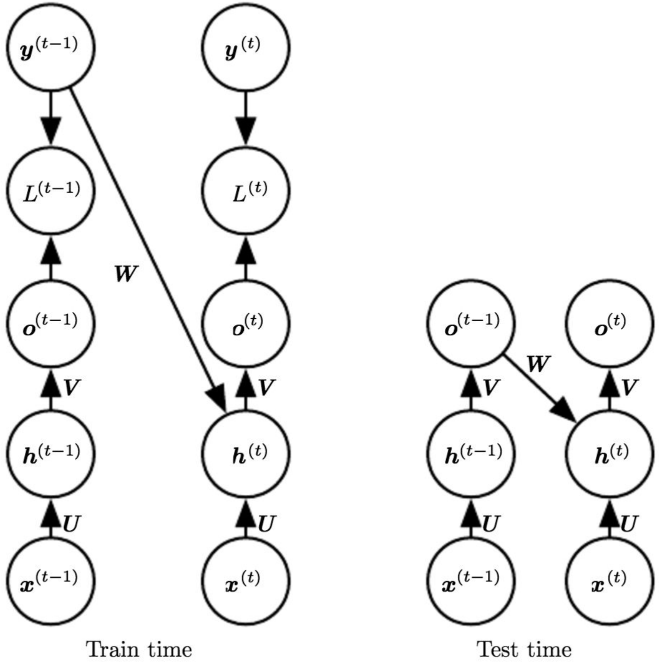


Less powerful than previous example because it lacks hidden-to-hidden recurrent connections, but easier to train for the same reason.

Not able to capture as many functions, e.g. cannot simulate a universal Turing machine.

Recurrent Neural Networks: Common Architectures

Teacher Forcing: Method to train RNNs with recurrent connections from output back into hidden states



Train time: Feed the correct output $\mathbf{y}^{(t)}$ as input to $\mathbf{h}^{(t+1)}$; training now parallelizable

Test time: Feed model's output $\mathbf{o}^{(t)}$ (as an approximation to the true output) as input to $\mathbf{h}^{(t+1)}$

Emerges from decomposing conditional probabilities, e.g. a sequence with two time steps:

$$\log p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$$
$$= \log p\left(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) + \log p\left(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$$

The first term on the right above inspires the architecture for training time. The model is then trained to maximize conditional (log) probability given **x** sequence so far, and any previous **y** values from training data.

Gradient Calculation in RNNs

- Set of nodes in computational graph given by **U, V, W, b** and **c**
- For each node **N**, calculate the gradient $\nabla_{\mathbf{N}} L$, based on computed gradient at all descendant nodes
- Recursion begins at nodes immediately preceding final loss:

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

- Assumptions:
  - Outputs $\mathbf{o}^{(t)}$ are used as the argument to softmax function to obtain the vector **yhat** of probabilities over output
  - Loss is the negative log-likelihood of true target $y^{(t)}$ given the input so far

Gradient Calculation in RNNs

- Gradient w.r.t. outputs at time step t, for all i, t:

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i, y^{(t)}}$$

- Continuing to work our way backwards, at final time step T, $\mathbf{h}^{(T)}$ only has $\mathbf{o}^{(T)}$ as descendant, so gradient is:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L.$$

Gradient Calculation in RNNs

- Continue iterating backwards in time from t = T-1 to t = 1 (note that **h**$^{(t)}$ has both **o**$^{(t)}$ and **h**$^{(t+1)}$ as descendants):

$$\begin{aligned}\nabla_{\boldsymbol{h}^{(t)}} L &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} (\nabla_{\boldsymbol{o}^{(t)}} L) \\ &= \boldsymbol{W}^{\top} (\nabla_{\boldsymbol{h}^{(t+1)}} L)\, \mathrm{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^{\top} (\nabla_{\boldsymbol{o}^{(t)}} L)\end{aligned}$$

where $\mathrm{diag}\left(1 - (\boldsymbol{h}^{(t+1)})^2\right)$ is the Jacobian of the hyperbolic tangent associated with hidden layer at time t+1

Gradient Calculation in RNNs

Having calculated gradients on the internal nodes, we can then calculate gradients on the parameter nodes:

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \right)^\top \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}} \right)^\top \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top}$$

Gradient Calculation in RNNs

Note the use of dummy variables $\mathbf{W}^{(t)}$ defined to be copies of $\mathbf{W}$ but with each $\mathbf{W}^{(t)}$ used only at time step t, used to calculate contribution of weights to gradient at time step t

$$
\begin{aligned}
\nabla_{\boldsymbol{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)} \right)^2 \right) \left(\nabla_{\boldsymbol{h}^{(t)}} L \right) \boldsymbol{h}^{(t-1)^\top} \\
\nabla_{\boldsymbol{U}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)} \right)^2 \right) \left(\nabla_{\boldsymbol{h}^{(t)}} L \right) \boldsymbol{x}^{(t)^\top}
\end{aligned}
$$

Recurrent Networks as Directed Graphical Models

So far, losses $L^{(t)}$ were cross-entropies between training targets $\mathbf{y}^{(t)}$ and outputs $\mathbf{o}^{(t)}$, i.e. we train the RNN to maximize the log-likelihood

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}),$$

or if model contains connections between output at some time step to output at next time step

$$\log p(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}, \boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)})$$

Recurrent Networks as Directed Graphical Models

Simple example: Sequence of scalar random variables Y = {$y^{(1)}$, …, $y^{(T)}$} with no additional inputs x. Then, the joint distribution of these observations is

$$P(\mathbb{Y}) = P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\mathbf{y}^{(t)} \mid \mathbf{y}^{(t-1)}, \mathbf{y}^{(t-2)}, \ldots, \mathbf{y}^{(1)})$$

and the loss function is given by

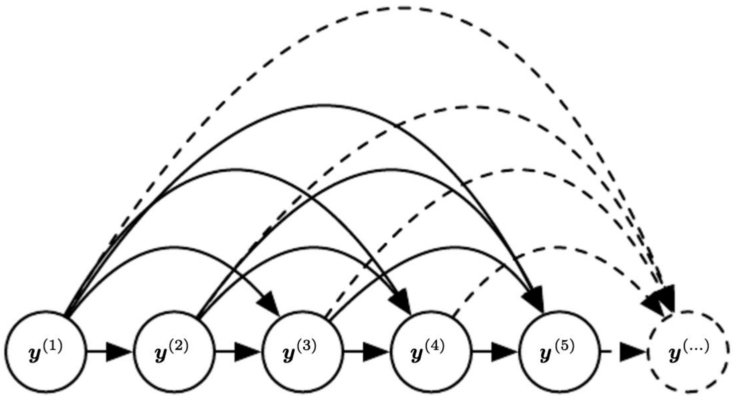
$$L = \sum_t L^{(t)}$$

where

$$L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \ldots, y^{(1)})$$

Recurrent Networks as Directed Graphical Models

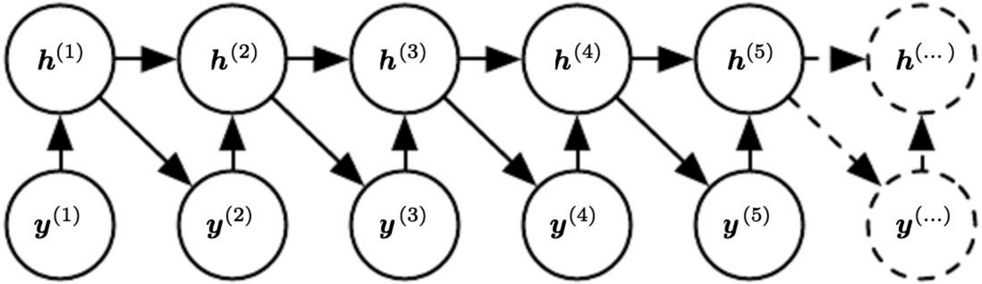
Graphically, this RNN can be represented as a fully connected graphical model by marginalizing out the hidden units $\mathbf{h}^{(t)}$, as follows



which is very inefficient; if each y takes on k values, this results in $O(k^T)$ parameters, where T is the length of the sequence.

Recurrent Networks as Directed Graphical Models

Alternatively, we can introduce the $\mathbf{h}^{(t)}$ nodes as mediators of the effect of any past variable $y^{(t)}$ on any future variable $y^{(t+k)}$. Graphically,



Note that every stage in the sequence shares the same structure. Further, if the time-series is stationary, then we can invoke parameter sharing to reduce the number of parameters in the RNN to O(1) as a function of sequence length.

Recurrent Networks as Directed Graphical Models

Typically, we sample from the conditional probability at every time step. However, the RNN should know when a sequence ends (equivalently, determine the length of the sequence), or it can result in (for example) sentences that end before they are complete. This is achievable in a few ways:

- Add a special symbol at the end of each training sequence (Schmidhuber, 2012)
- Introduce an extra Bernoulli output with probability p to continue generation, and probability 1-p to halt further generation at each time step
- Predict sequence length T as an extra output, and use this as a recurrent input in the next time step (Goodfellow et al. 2014)

Sequences conditioned on context

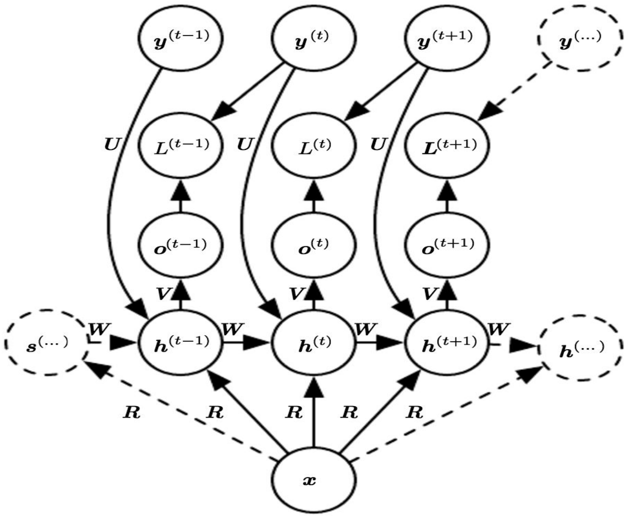
Recall that a model representing a variable P(**y; theta**) can be interpreted as a model representing a conditional distribution P(**y**|**w**) with **w**=**theta** (some constant).

We can extend this to represent a distribution P(**y**|**x**) by using the same P(**y**|**w**) but with **w**=**theta**(**x**) (a function of **x**).

Previously, we discussed RNNs that take a sequence of vectors $\mathbf{x}^{(t)}$ as input. Another option is to take only a single vector **x** of fixed-size as input.

Sequences conditioned on context

A common approach to do this is to introduce **x** as an extra input at each time step



where we introduce a new weight matrix **R** that introduces new effective bias parameters $\mathbf{x}^T\mathbf{R}$ for each of the hidden units. This RNN is appropriate for tasks such as image captioning, i.e. producing sequence of words describing image.

Sequences conditioned on context

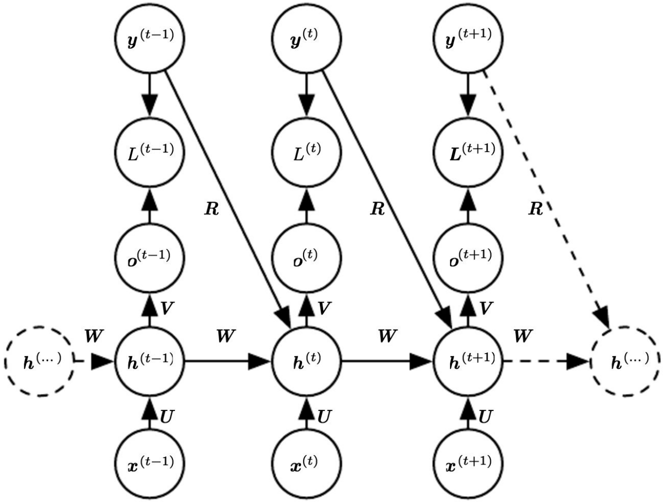
Instead of receiving only a single vector **x** as input, the RNN may receive a sequence of vectors $\mathbf{x}^{(t)}$ as input. Previously, we described such an RNN with a conditional distribution $P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)})$ that is assumed to factorize as

$$\prod_t P(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}).$$

We can remove this assumption of conditional independence by adding connections from output at time t to hidden unit at time t+1, as in the following slide.

Sequences conditioned on context

This allows us to represent arbitrary probability distributions over the **y** sequence.



Note however the constraint here that the length of both sequences **x** and **y** must be the same.