ECBM E6040 Neural Networks and Deep Learning

Lecture #12: Sequence Modeling: Recurrent and Recursive Nets (cont'd)

Aurel A. Lazar

Columbia University
Department of Electrical Engineering

April 19, 2016

Outline of Part I

2. Summary of the Previous Lecture
   - Topics Covered
   - Learning Objectives

Outline of Part II

3. Recurrent Neural Networks (cont'd)
   - Recurrent Networks as Directed Graphical Models
   - Modeling Sequences Conditioned in the Context with RNNs
   - Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent and Recursive Neural Networks
6. The Long Short-Term Memory and Other Gated Networks

Part I

Review of Previous Lecture

Topics Covered in Convolutional Networks (cont'd)

- Variants of the Basic Convolution Function
- The Neuroscientific Basis for Convolutional Networks

Topics Covered in Sequence Modeling: Recurrent and Recursive Nets

- Unfolding Computational Graphs
- Recurrent Neural Networks

Learning Objectives in Convolutional Networks (cont'd)

- Many variants of basic convolution function have been implemented to
  - reduce the computational cost: downsampling;
  - control the kernel width and the size of the output independently: zero-padding;
  - reduce memory requirements: parameter sharing.
- The architecture of CNNs has drawn inspiration from the vertebrate visual system but there are also strong differences:
  - the detector units of a convolutional network are designed to emulate the local properties of simple cells;
  - CNN pooling units attempt to address the small shift invariance properties of complex cells;
  - neuroscience has told us relatively little about how to train convolutional networks.

Learning Objectives in Sequence Modeling: Recurrent and Recursive Nets

- MLPs and CNNs are feedforward neural networks whose connections do not form cycles. RNNs are neural networks that allow cyclic connections.
- MLPs and CNNs can only map from input to output vectors; RNNs can in principle map from the entire history of previous inputs to each output.
- Any function computable by a Turing Machine can be computed by a finite size RNN.

Part II

Today's Lecture

We typically train the RNN to estimate the conditional distribution of the next sequence element y^t given the past inputs. This may mean that we maximize the log-likelihood

$$\log p(\mathbf{y}^t|\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^t),$$

or, if the model includes connections from the output at one time step to the next time step,

$$\log p(\mathbf{y}^t|\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^t, \mathbf{y}^1, \mathbf{y}^2, ..., \mathbf{y}^{t-1}).$$

Decomposing the joint probability over the sequence of y values as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence; the computation has a high complexity, however.

Fully Connected Graphical Models

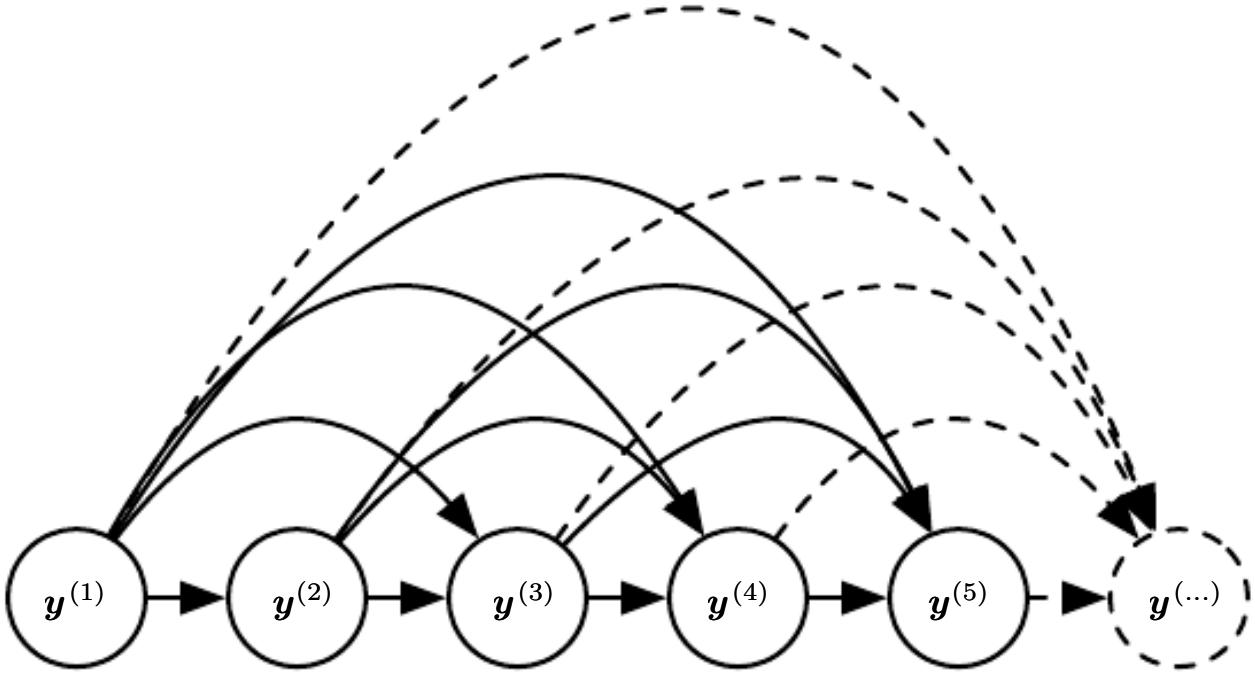
We consider here the case where the RNN models only a sequence of scalar random variables $\mathbb{Y} = (\mathbf{y}^1, \mathbf{y}^2, ..., \mathbf{y}^\tau)$, with no additional inputs $\mathbf{x}$.
We parametrize the joint distribution of these observations using the chain rule for conditional probabilities:

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

The loss function is then

$$L = \sum_t L^t \quad \text{with} \quad L^t = -\log \mathbb{P}(\mathbf{y}^t | \mathbf{y}^{t-1}, \mathbf{y}^{t-2}, ..., \mathbf{y}^1).$$

Fully Connected Graphical Models (cont'd)



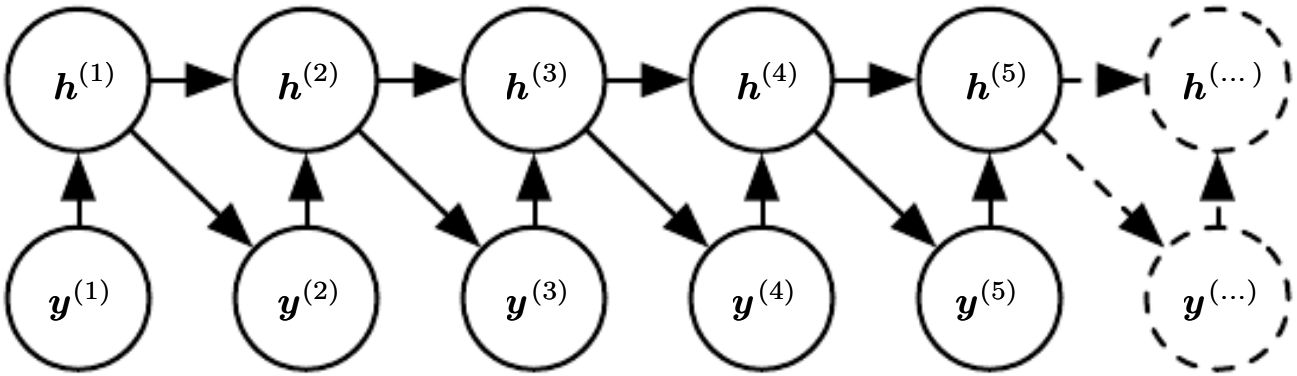
Fully connected graphical model for a sequence $\mathbf{y}^1, \mathbf{y}^2, ..., \mathbf{y}^t$: every past observation $\mathbf{y}^i$ may influence the conditional distribution of some $\mathbf{y}^t$, $t > i$, given the previous values. Parametrizing the graphical model directly according to this graph might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence.

Fully Connected Graphical Models (cont'd)

If we use the variable $\boldsymbol{h}$ to represent the state through a deterministic function:

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

we can obtain a very efficient parametrization:



Every stage in the sequence (for $\boldsymbol{h}^t$ and $\boldsymbol{y}^t$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

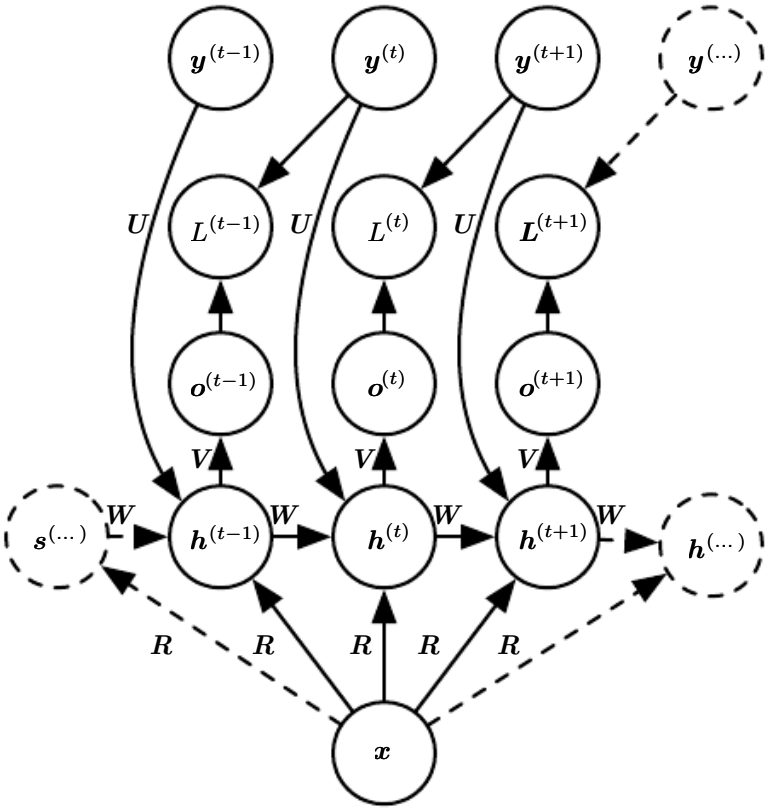
Fully Connected Graphical Models (cont'd)

To achieve statistical and computational efficiency a natural assumption to make is that the graphical model only contains edges from $\mathbf{y}^{t-k}, ..., \mathbf{y}^{t-1}$ to $\mathbf{y}^t$, rather than containing edges from the entire past history.

RNNs are useful when we believe that the distribution over $\mathbf{y}^t$ may depend on a value of $\mathbf{y}^i$ from the distant past in a way that is not captured by the effect of $\mathbf{y}^i$ on $\mathbf{y}^{t-1}$.

Incorporating the $\mathbf{h}^t$ nodes in the graphical model decouples the past and the future, acting as an intermediate quantity between them. A variable $\mathbf{y}^i$ in the distant past may influence a variable $\mathbf{y}^t$ via its effect on $\mathbf{h}$. The structure of this graph shows that the model can be efficiently parametrized by using the same conditional probability distributions at each time step.

RNNs with a Single Input Vector



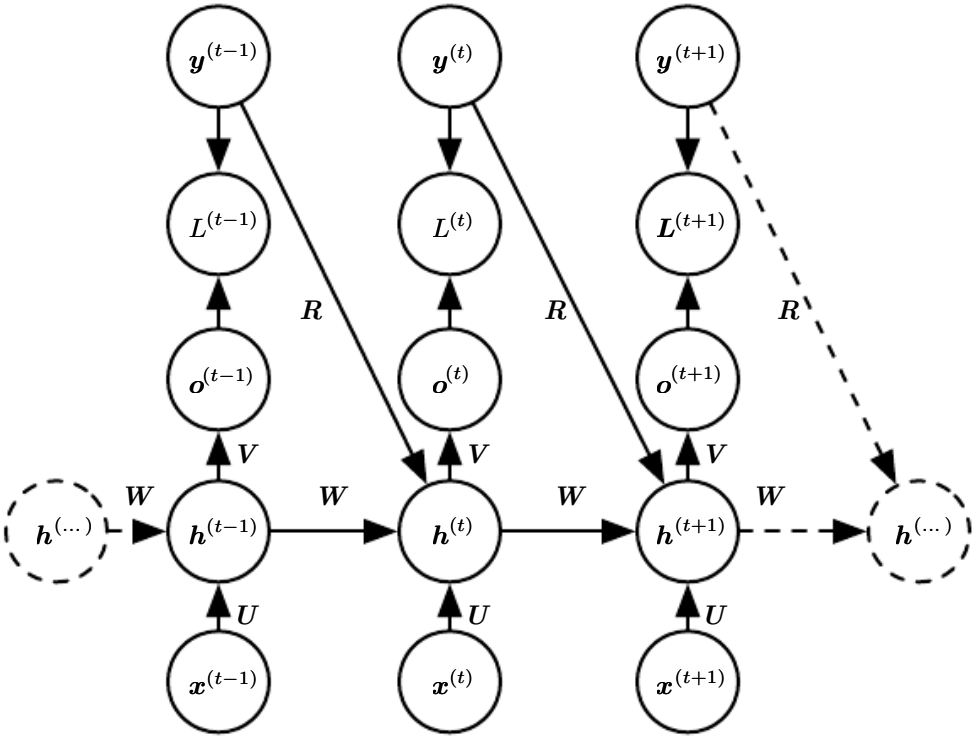
The interaction between the input **x** and each hidden unit vector $\boldsymbol{h}^t$ is parametrized by **R**.

An RNN that maps a fixed-length vector **x** into a distribution over sequences **Y**.

This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image.

Each element $\mathbf{y}^t$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

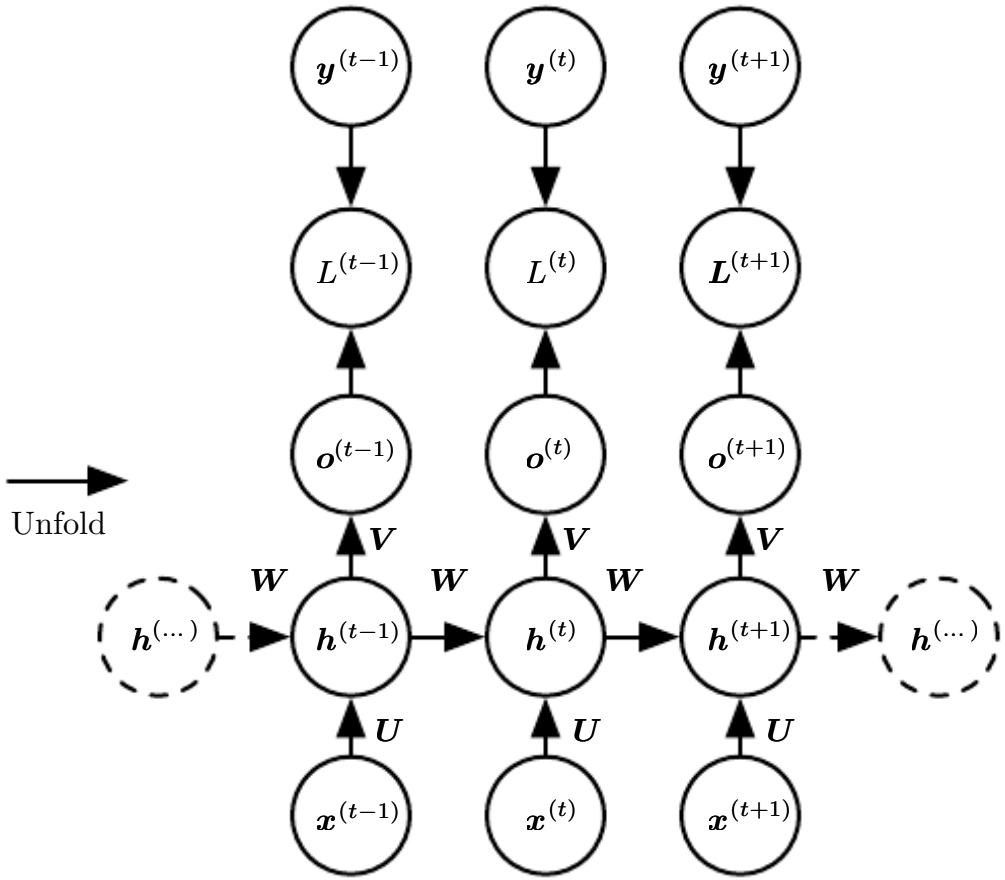
A Conditional Recurrent Neural Network



A conditional RNN mapping a variable-length sequence of **x** values into a distribution over sequences of **y** values of the same length.

This RNN contains connections from the previous output to the current state. These connections allow this RNN to model an arbitrary distribution over sequences of **y** given sequences of **x** of the same length.

A Conditional Recurrent Neural Network (cont'd)



The RNN on the left is only able to represent distributions in which the **y** values are conditionally independent from each other given the **x** values.

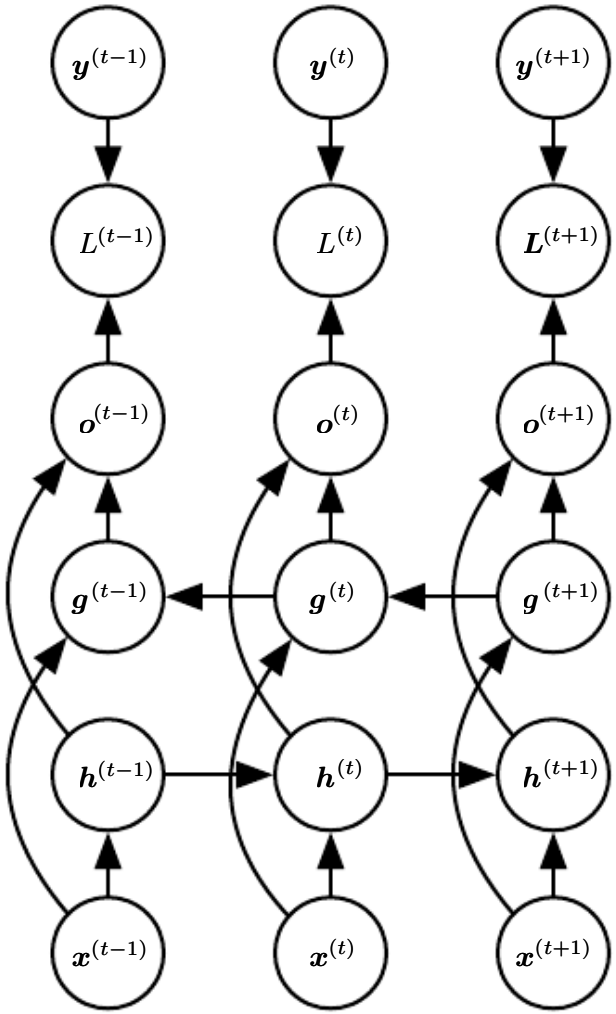
Bidirectional RNNs

In many applications we want to output a prediction of $\mathbf{y}^t$ which may depend on the whole input sequence.

For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation and potentially may even depend on the next few words because of the linguistic dependencies between nearby words: if there are two interpretations of the current word that are both acoustically plausible, we may have to look far into the future (and the past) to disambiguate them.

This is also true of handwriting recognition and many other sequence-to-sequence learning tasks.

Bidirectional RNNs (cont'd)



Computation of a typical bidirectional recurrent neural network, meant to learn to map input sequences $\mathbf{x}$ to target sequences $\mathbf{y}$, with loss L^t at each step t.

The $\boldsymbol{h}$ recurrence propagates information forward in time (towards the right) while the $\boldsymbol{g}$ recurrence propagates information backward in time (towards the left). Thus at each point t, the output units $\boldsymbol{o}^t$ can benefit from a relevant summary of the past in its $\boldsymbol{h}^t$ input and from a relevant summary of the future in its $\boldsymbol{g}^t$ input.

Bidirectional RNNs (cont'd)

In the typical bidirectional RNN, $\boldsymbol{h}^t$ stands for the state of the sub-RNN that moves forward through time and $\boldsymbol{g}^t$ stands for the state of the sub-RNN that moves backward through time. This allows the output units $\boldsymbol{o}^t$ to compute a representation that depends on both the past and the future but is most sensitive to the input values around time t, without having to specify a fixed-size window around t (as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a fixed-size look-ahead buffer).

This idea can be naturally extended to 2-dimensional input, such as images, by having four RNNs, each one going in one of the four directions: up, down, left, right. At each point (i, j) of a 2-D grid, an output $\boldsymbol{O}_{ij}$ could then compute a representation that would capture mostly local information but could also depend on long-range inputs, if the RNN is able to learn to carry that information. Compared to a convolutional network, RNNs applied to images are typically more expensive but allow for long-range lateral interactions between features in the same feature map.

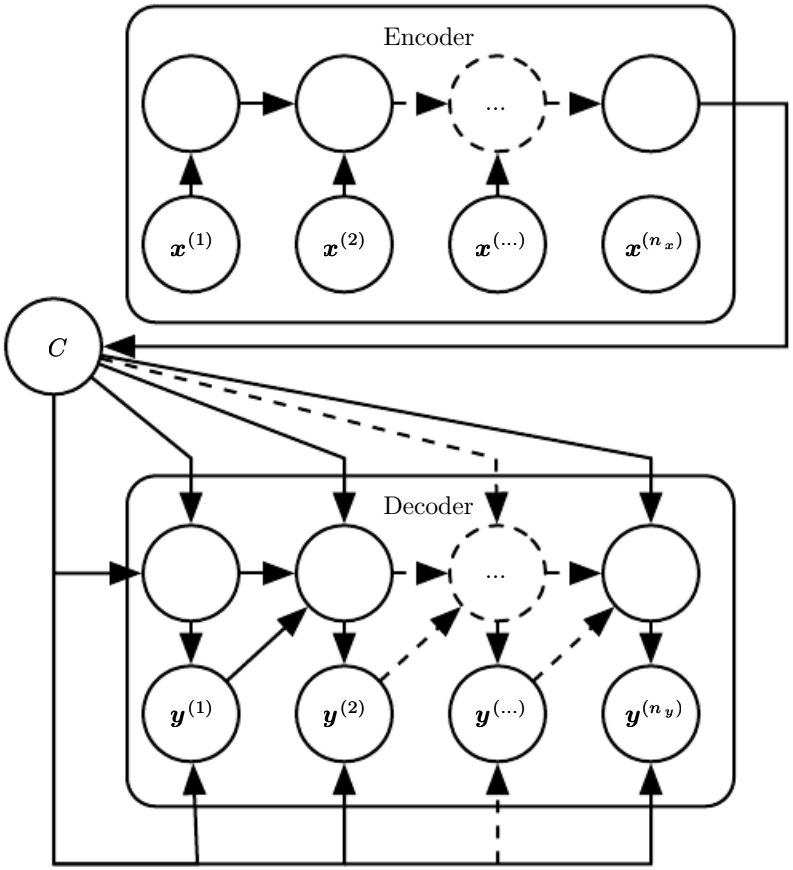
Encoder-Decoder RNN Architecture

An RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length.

This comes up in many applications, such as speech recognition, machine translation or question answering, where the input and output sequences in the training set are generally not of the same length.

The input to an RNN is often call the context. We want to produce a representation of this context, $\mathbf{C}$. The context $\mathbf{C}$ might be a vector or sequence of vectors that summarizes the input sequence $\mathbf{X} = (\mathbf{x}^1, ..., \mathbf{x}^{n_x})$.

Encoder-Decoder RNN Architecture (cont'd)



An encoder or reader or RNN processes the input sequence. The encoder emits the context $\mathbf{C}$, usually as a simple function of its final hidden state.

A decoder or writer or output RNN is conditioned on that fixed-length vector to generate the output sequence $\mathbf{Y} = (\mathbf{y}^1, ..., \mathbf{y}^{n_y})$.

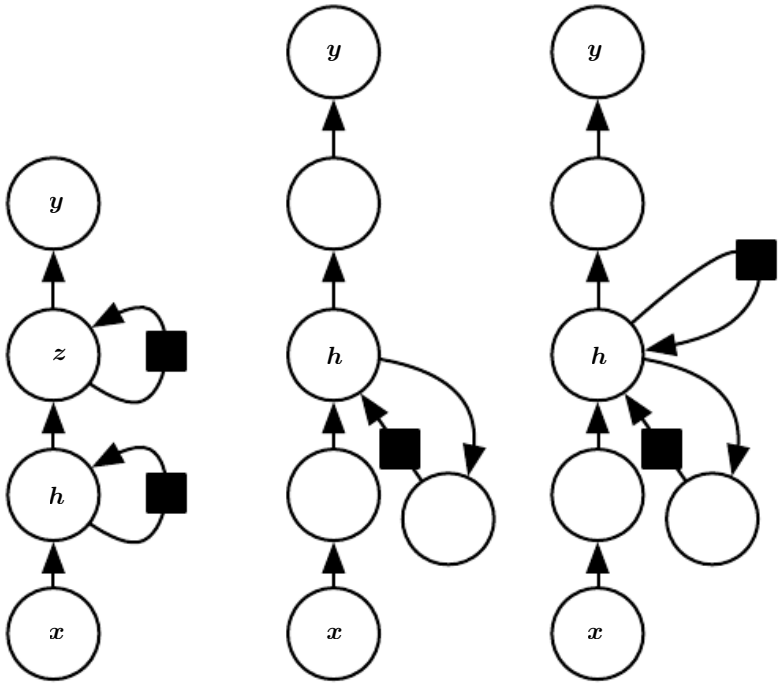
Encoder-Decoder RNN Architecture (cont'd)

The two RNNs are trained jointly to maximize the average of

$$\log \mathbb{P}(\mathbf{y}^1, ..., \mathbf{y}^{n_y} | \mathbf{x}^1, ..., \mathbf{x}^{n_x})$$

over all the pairs of $\mathbf{x}$ and $\mathbf{y}$ sequences in the training set. The last state $\boldsymbol{h}^{n_x}$ of the encoder RNN is typically used as a representation $\mathbf{C}$ of the input sequence that is provided as input to the decoder RNN.

Deep Recurrent Networks



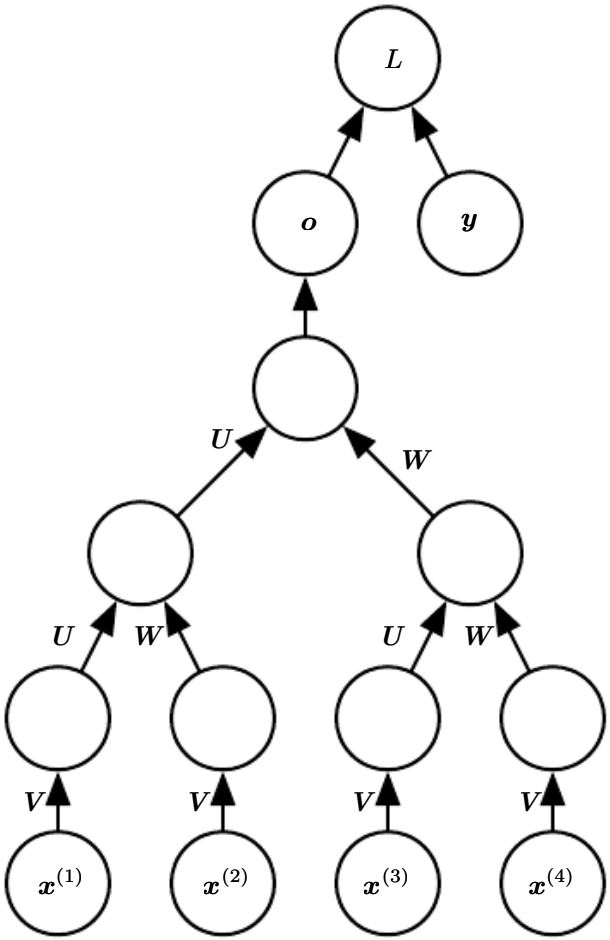
A recurrent neural network can be made deep in many ways:
(a) the hidden recurrent state can be broken down into groups organized hierarchically;
(b) deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps;
(c) the path-lengthening effect can be mitigated by introducing skip connections.

Recursive Neural Networks

Recursive Neural Networks exhibit a computational graph that is structured as a deep tree, rather than the chain-like graph of RNNs.

Recursive networks have been successfully applied to processing data structures as input to neural nets, in natural language processing as well as in computer vision.

Recursive Neural Networks (cont'd)



A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree.

A variable-size sequence $(\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^t)$ can be mapped to a fixed-size representation (the output $\mathbf{o}$), with a fixed set of parameters (the weight matrices $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$).

The graph illustrates a supervised learning case in which some target $\mathbf{y}$ associated with the whole sequence is provided.

The Challenge of Long-Term Dependencies

The mathematical challenge of learning long-term dependencies in recurrent networks is that gradients propagated over many stages tend to either vanish (most of the time) or explode (rarely, but with much damage to the optimization).

Even if we assume that the parameters are such that the recurrent network is stable (can store memories, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the multiplication of many Jacobians) compared to short-term ones.

The Challenge of Long-Term Dependencies (cont'd)

The function composition employed by recurrent neural networks somewhat resembles matrix multiplication. We can think of the recurrence relation

$$\boldsymbol{h}^t = \mathbf{W}^T \boldsymbol{h}^{t-1}$$

as a very simple recurrent neural network lacking a nonlinear activation function, and lacking inputs $\mathbf{x}$. It can be written

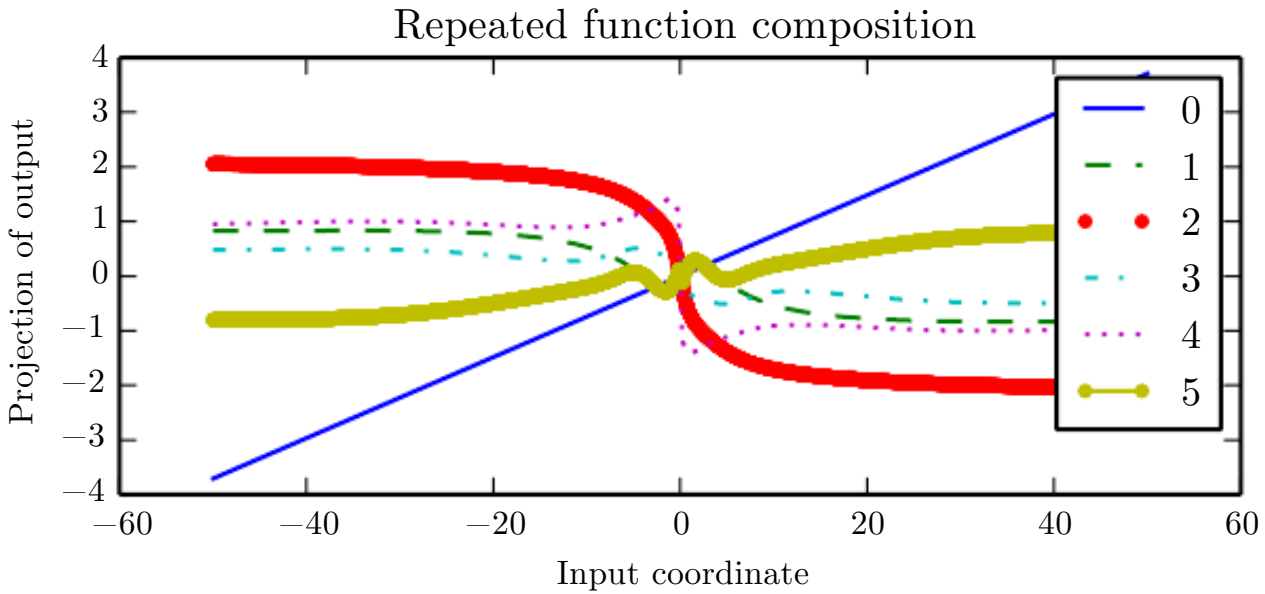
$$\boldsymbol{h}^t = (\mathbf{W}^t)^T \boldsymbol{h}^0,$$

and if $\mathbf{W}$ admits an eigendecomposition $\mathbf{W} = \mathbf{Q}\boldsymbol{\Lambda}^t\mathbf{Q}^T$,

$$\boldsymbol{h}^t = \mathbf{Q}^T\boldsymbol{\Lambda}\mathbf{Q}\ \boldsymbol{h}^0.$$

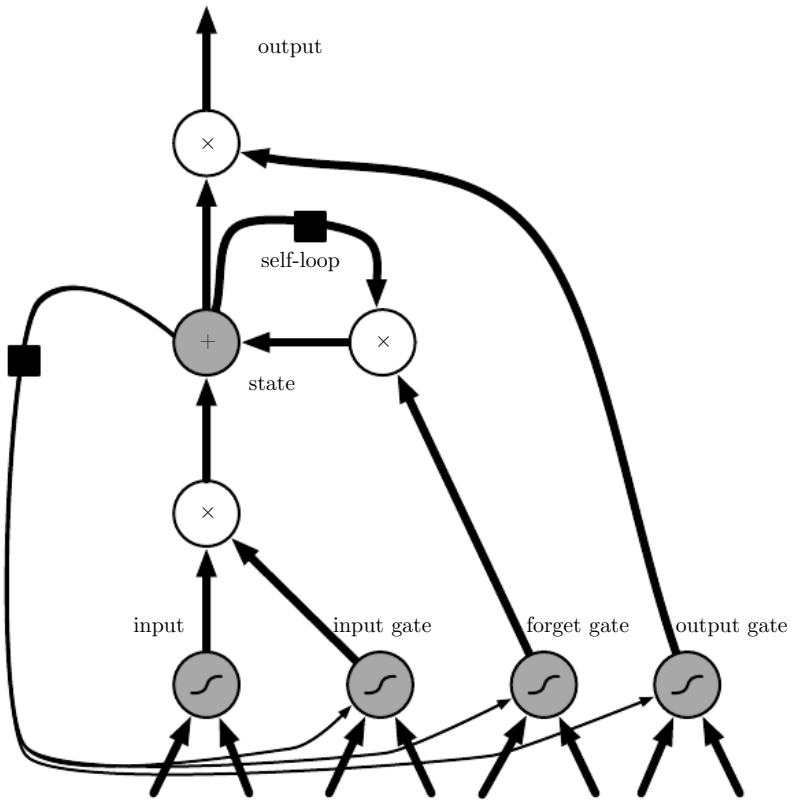
Therefore, eigenvalues with magnitude less than one will decay to zero and eigenvalues with magnitude greater than one will explode. Any component of $\boldsymbol{h}^0$ that is not aligned with the largest eigenvector will eventually be discarded.

The Challenge of Long-Term Dependencies (cont'd)



A linear projection of a 100-dimensional hidden state down to a single dimension, plotted on the y-axis. The x-axis is the coordinate of the initial state along a random direction in the 100-dimensional space. We can thus view this plot as a linear cross-section of a high-dimensional function. The plots show the function after each time step, or equivalently, after each number of times the transition function has been composed. The result is an extremely nonlinear behavior.

The Long Short-Term Memory Networks



Block diagram of the LSTM recurrent network cell. The self-loop is a path where the gradient can flow for long durations.

Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units.

The Long Short-Term Memory Networks (cont'd)

LSTM recurrent networks have LSTM cells that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information.

The state unit s_i^t that has a linear self-loop whose weight (or the associated time constant) is controlled by a forget gate unit f_i^t that sets this weight to a value between 0 and 1 via a sigmoid unit:

$$f_i^t = \sigma(b_i^f + \sum_j U_{ij}^f x_j^t + \sum_j W_{ij}^f h_j^{t-1}),$$

where $\mathbf{x}^t$ is the current input vector and $\mathbf{h}^t$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and $\mathbf{b}^f$, $\mathbf{U}^f$, $\mathbf{W}^f$ are respectively biases, input weights and recurrent weights for the forget gates.

The Long Short-Term Memory Networks (cont'd)

The LSTM cell internal state is thus updated with a conditional self-loop weight f_i^t as follows:

$$s_i^t = f_i^t s_i^{t-1} + g_i^t \sigma(b_i + \sum_j U_{ij} x_j^t + \sum_j W_{ij} h_j^{t-1})$$

where **b**, **U**, **W** respectively denote the biases, input weights and recurrent weights into the LSTM cell.
The external input gate unit g_i^t is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^t = \sigma(b_i^g + \sum_j U_{ij}^g x_j^t + \sum_j W_{ij}^g h_j^{t-1}).$$

The Long Short-Term Memory Networks (cont'd)

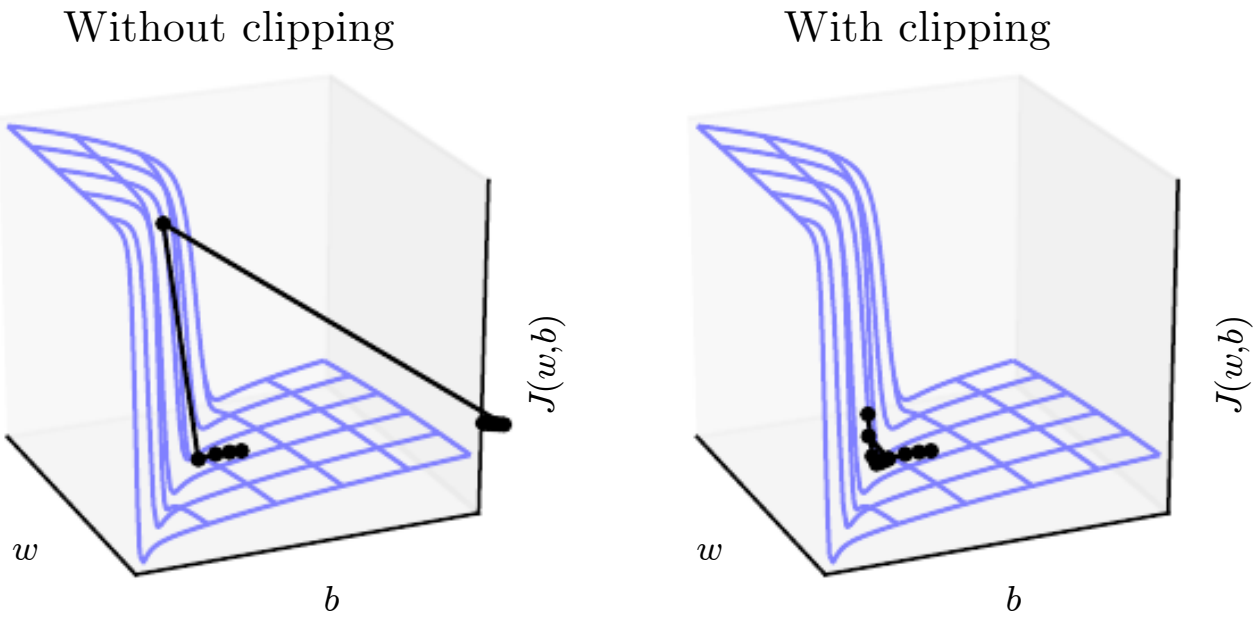
The output h_i^t of the LSTM cell can also be shut off, via the output gate q_i^t, which also uses a sigmoid unit for gating:

$$h_i^t = \tanh(s_i^t) q_i^t$$
$$q_i^t = \sigma(b_i^o + \sum_j U_{ij}^o x_j^t + \sum_j W_{ij}^o h_j^{t-1})$$

which has parameters $\mathbf{b}^o$, $\mathbf{U}^o$, $\mathbf{W}^o$ for its biases, input weights and recurrent weights, respectively.

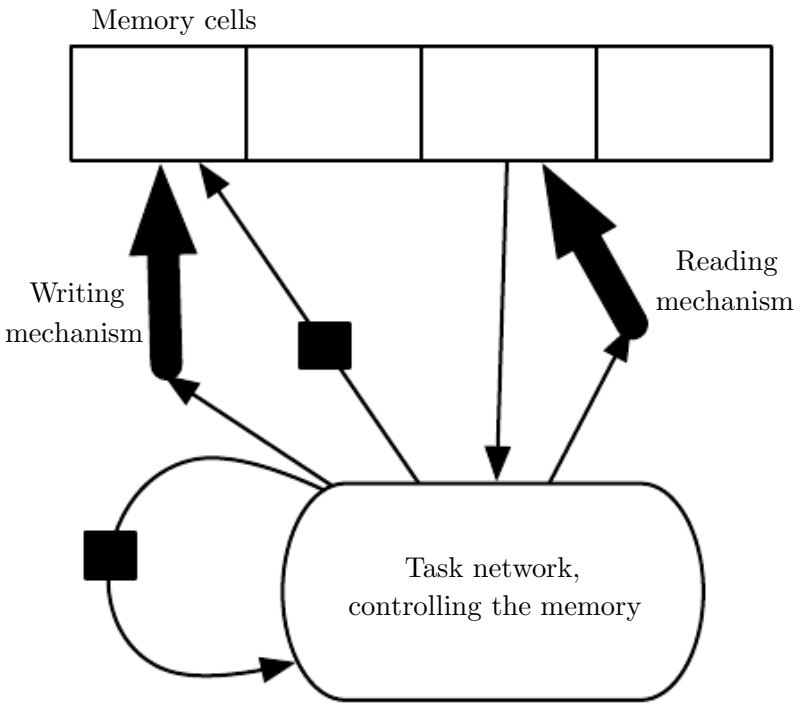
LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures, first on artificial data sets designed for testing the ability to learn long-term dependencies, then on challenging sequence processing tasks where state-of-the-art performance was obtained.

Gradient Clipping in Recurrent Networks



(Left) Gradient descent clipping overshoots the bottom of the small ravine, then receives a very large gradient from the cliff face. The large gradient propels the parameters outside the axes of the plot.
(Right) Gradient descent clipping has a more moderate reaction to the cliff. While it does ascend the cliff face, the step size is restricted so that it cannot be propelled away from steep region near the solution.

Explicit Memory



An example of a network with an explicit memory, capturing some of the key design elements of the Neural Turing Machine.
Here, we distinguish the representation part of the model (the task network, here a recurrent net at the bottom) from the memory part of the model (the set of cells), which can store facts.
The task network learns to control the memory, deciding where to read from and where to write to within the memory (through the reading and writing mechanisms, indicated by bold arrows pointing at the reading and writing addresses).