Lecture 11
Recurrent Neural Networks I

CMSC 35246: Deep Learning

Shubhendu Trivedi
&
Risi Kondor

University of Chicago

May 01, 2017

Introduction

Sequence Learning with Neural Networks

Some Sequence Tasks

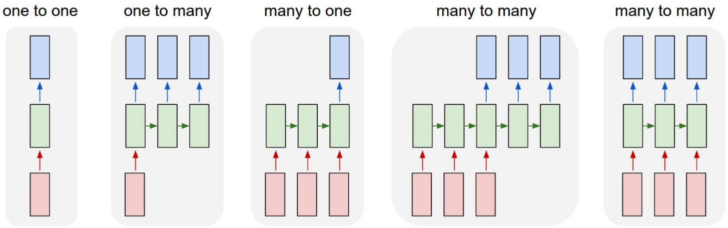


Figure credit: Andrej Karpathy

Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality
- Great e.g.: Inputs - Images, Output - Categories
- Bad e.g.: Inputs - Text in one language, Output - Text in another language
- MLPs treat every example independently. How is this problematic?
- Need to re-learn the rules of language from scratch each time
- Another example: Classify events after a fixed number of frames in a movie
- Need to resuse knowledge about the previous events to help in classifying the current.

Recurrent Networks

- Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data
- Sequential data: Each example consists of a pair of sequences. Each example can have different lengths
- Need to take advantage of an old idea in Machine Learning: Share parameters across different parts of a model
- Makes it possible to extend the model to apply it to sequences of different lengths not seen during training
- Without parameter sharing it would not be possible to share statistical strength and generalize to lengths of sequences not seen during training
- Recurrent networks share parameters: Each output is a function of the previous outputs, with the same update rule applied

Recurrence

- Consider the classical form of a dynamical system:

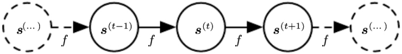
$$s^{(t)} = f(s^{(t-1)}; \theta)$$

- This is recurrent because the definition of s at time t refers back to the same definition at time $t-1$
- For some finite number of time steps τ, the graph represented by this recurrence can be unfolded by using the definition $\tau - 1$ times. For example when $\tau = 3$

$$s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta)$$

- This expression does not involve any recurrence and can be represented by a traditional directed acyclic computational graph

Recurrent Networks



- Consider another dynamical system, that is driven by an external signal $x^{(t)}$

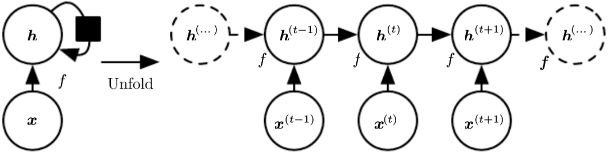
$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta)$$

- The state now contains information about the whole past sequence
- RNNs can be built in various ways: Just as any function can be considered a feedforward network, any function involving a recurrence can be considered a recurrent neural network

- We can consider the states to be the hidden units of the network, so we replace $s^{(t)}$ by $h^{(t)}$

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

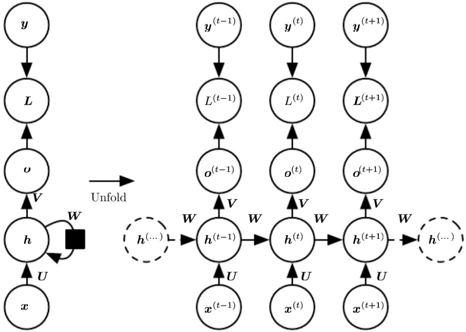
- This system can be drawn in two ways:



- We can have additional architectural features: Such as output layers that read information from h to make predictions

- When the task is to predict the future from the past, the network learns to use $h^{(t)}$ as a summary of task relevant aspects of the past sequence upto time t
- This summary is lossy because it maps an arbitrary length sequence $(x^{(1)}, x^{(t-1)}, \ldots, x^{(2)}, x^{(1)})$ to a fixed vector $h^{(t)}$
- Depending on the training criterion, the summary might selectively keep some aspects of the past sequence with more precision (e.g. statistical language modeling)
- Most demanding situation for $h^{(t)}$: Approximately recover the input sequence

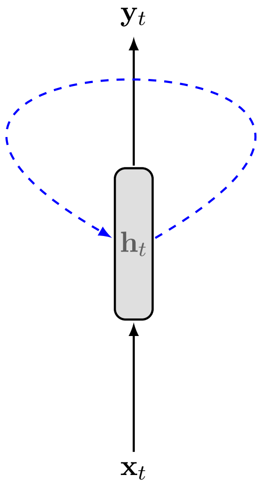
Design Patterns of Recurrent Networks



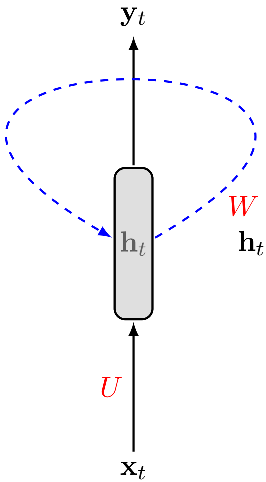
- **Plain Vanilla RNN:** Produce an output at each time stamp and have recurrent connections between hidden units
- Is infact Turing Complete (Siegelmann, 1991, 1995, 1995)

Design Patterns of Recurrent Networks

Plain Vanilla Recurrent Network

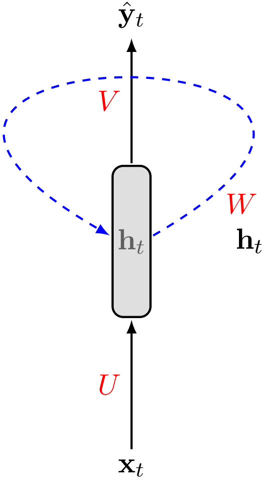


Recurrent Connections



$$\mathbf{h}_t = \psi(U\mathbf{x}_t + W\mathbf{h}_{t-1})$$

Recurrent Connections



$$\hat{\mathbf{y}}_t = \phi(V\mathbf{h}_t)$$

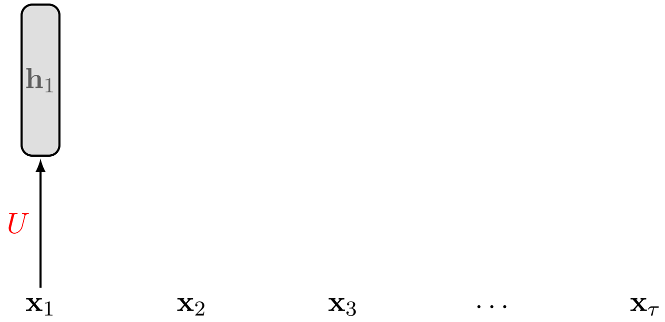
$$\mathbf{h}_t = \psi(U\mathbf{x}_t + W\mathbf{h}_{t-1})$$

ψ can be $\tanh$ and ϕ can be softmax

Unrolling the Recurrence

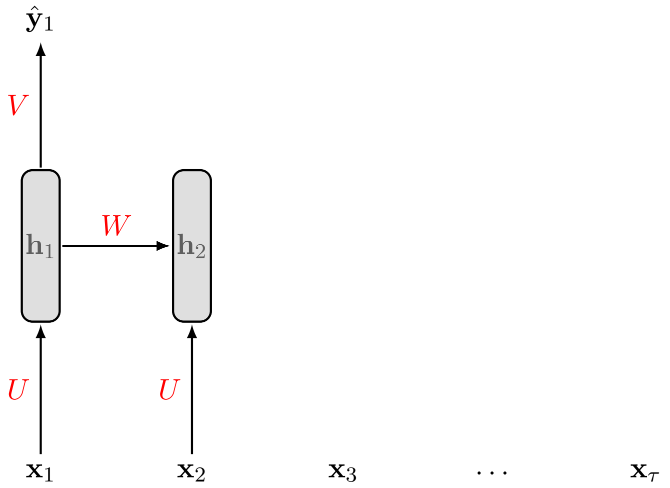
$\mathbf{x}_1 \qquad \mathbf{x}_2 \qquad \mathbf{x}_3 \qquad \cdots \qquad \mathbf{x}_\tau$

Unrolling the Recurrence

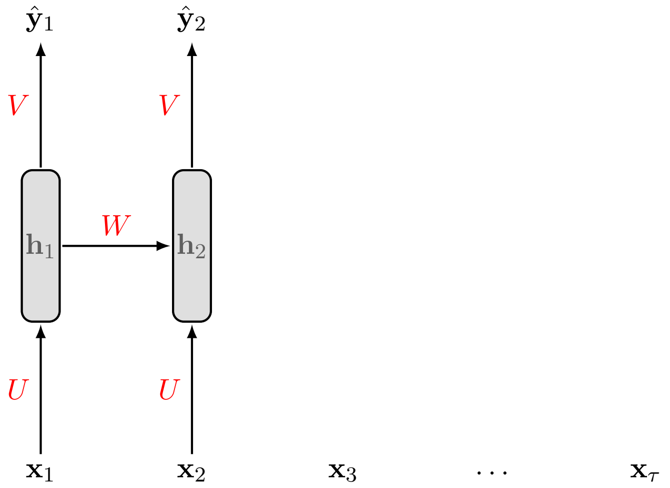


Unrolling the Recurrence

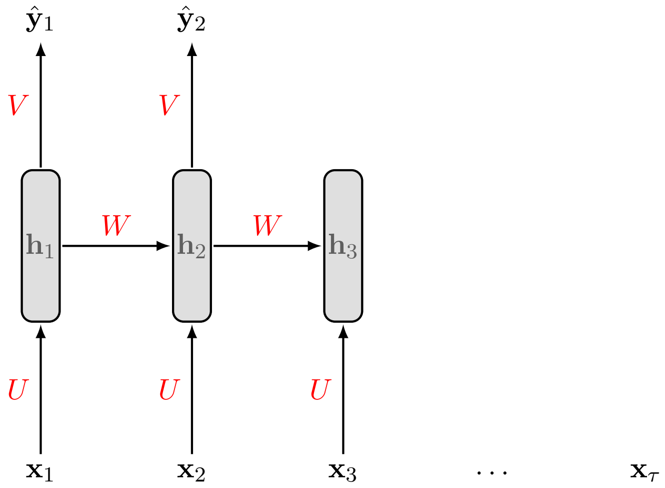
Unrolling the Recurrence



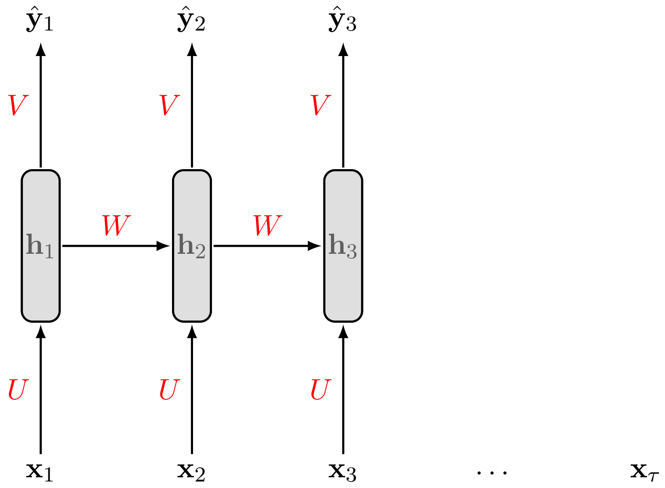
Unrolling the Recurrence



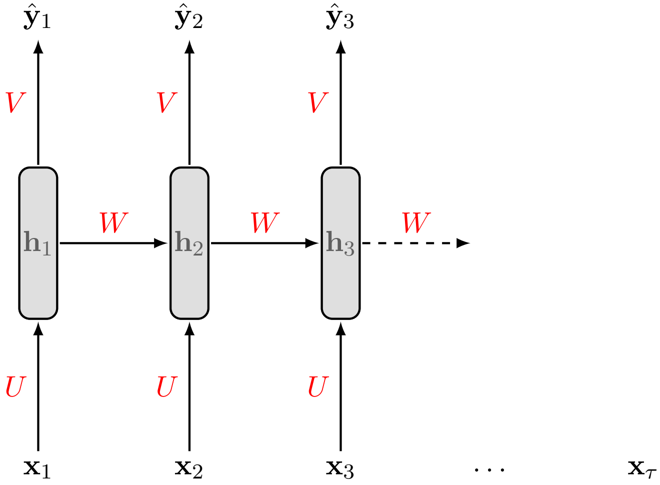
Unrolling the Recurrence



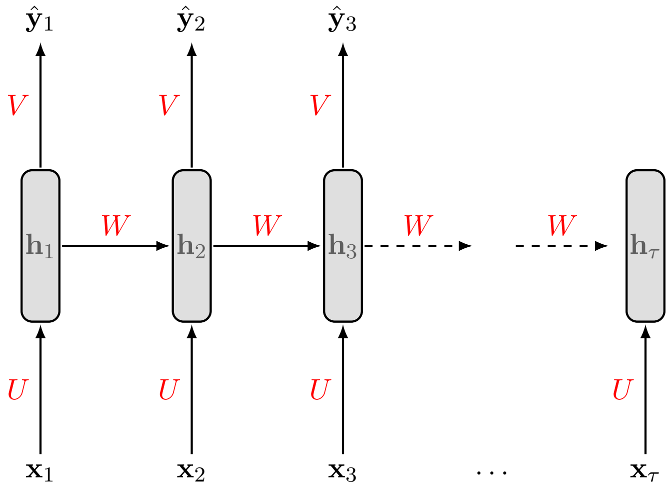
Unrolling the Recurrence



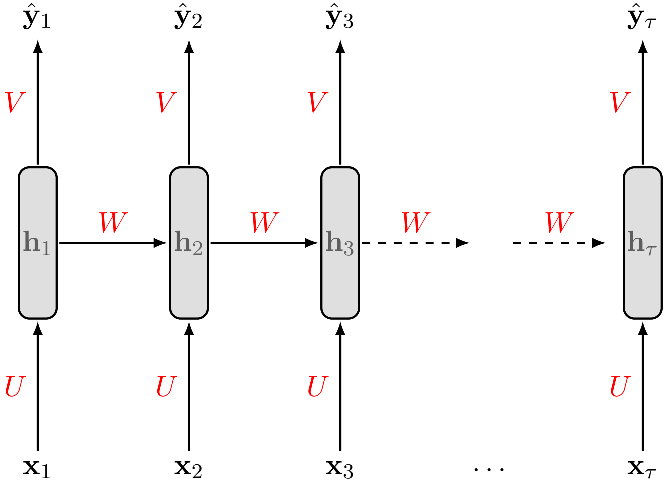
Unrolling the Recurrence



Unrolling the Recurrence



Unrolling the Recurrence



Feedforward Propagation

- This is a RNN where the input and output sequences are of the same length
- Feedforward operation proceeds from left to right
- Update Equations:

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

Feedforward Propagation

- Loss would just be the sum of losses over time steps
- If L_t is the negative log-likelihood of $\mathbf{y}_t$ given $\mathbf{x}_1, \dots, \mathbf{x}_t$, then:

$$L\Big(\{\mathbf{x}_1, \dots, \mathbf{x}_t\}, \{\mathbf{y}_1, \dots, \mathbf{y}_t\}\Big) = \sum_t L_t$$

- With:

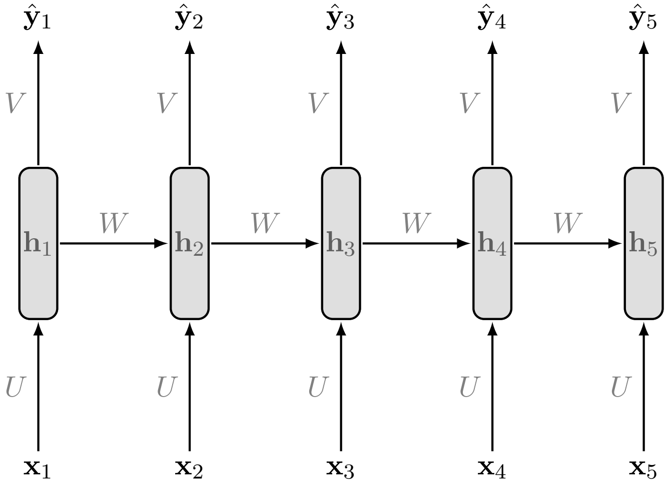
$$\sum_t L_t = -\sum_t \log p_{\text{model}}\big(\mathbf{y}_t | \{\mathbf{x}_1, \dots, \mathbf{x}_t\}\big)$$

- Observation: Forward propagation takes time $O(t)$; can't be parallelized

Backward Propagation

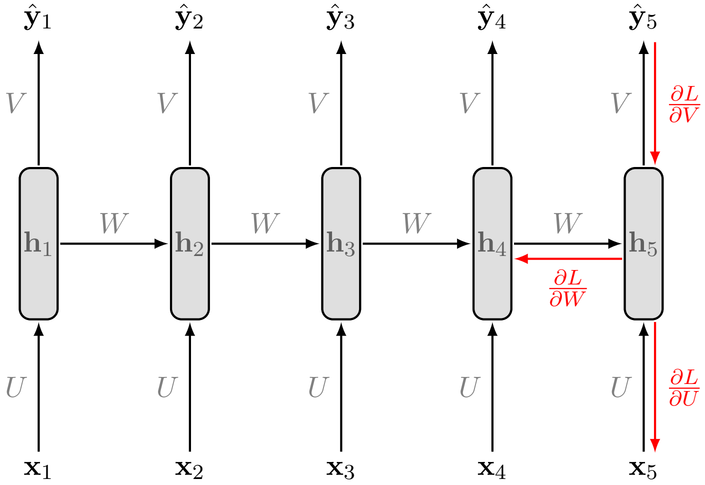
- Need to find: $\nabla_V L$, $\nabla_W L$, $\nabla_U L$
- And the gradients w.r.t biases: $\nabla_c L$ and $\nabla_b L$
- Treat the recurrent network as a usual multilayer network and apply backpropagation on the unrolled network
- We move from the right to left: This is called Backpropagation through time
- Also takes time $O(t)$

BPTT



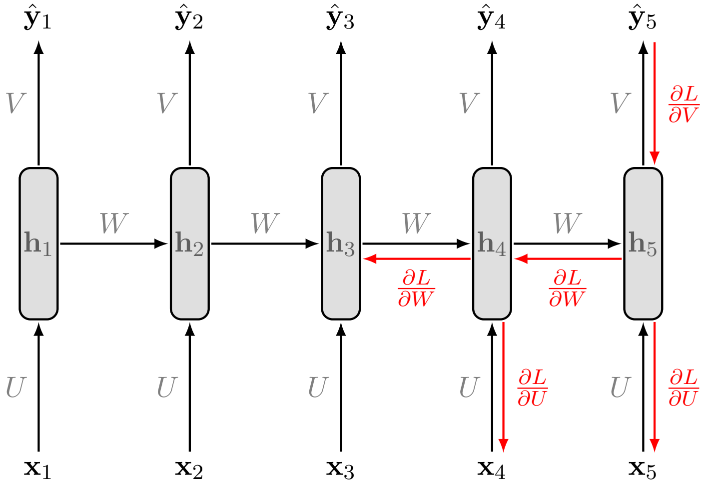
$\hat{\mathbf{y}}_1$ $\hat{\mathbf{y}}_2$ $\hat{\mathbf{y}}_3$ $\hat{\mathbf{y}}_4$ $\hat{\mathbf{y}}_5$
V V V V V
$\mathbf{h}_1$ W $\mathbf{h}_2$ W $\mathbf{h}_3$ W $\mathbf{h}_4$ W $\mathbf{h}_5$
U U U U U
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$ $\mathbf{x}_5$

BPTT



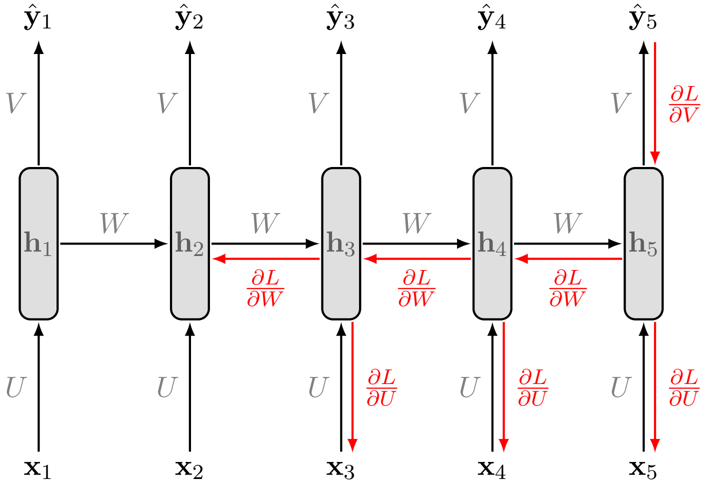
$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_4$
$\hat{\mathbf{y}}_5$
V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_4$
$\mathbf{h}_5$
W
$\frac{\partial L}{\partial W}$
U
$\frac{\partial L}{\partial U}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{x}_5$

BPTT

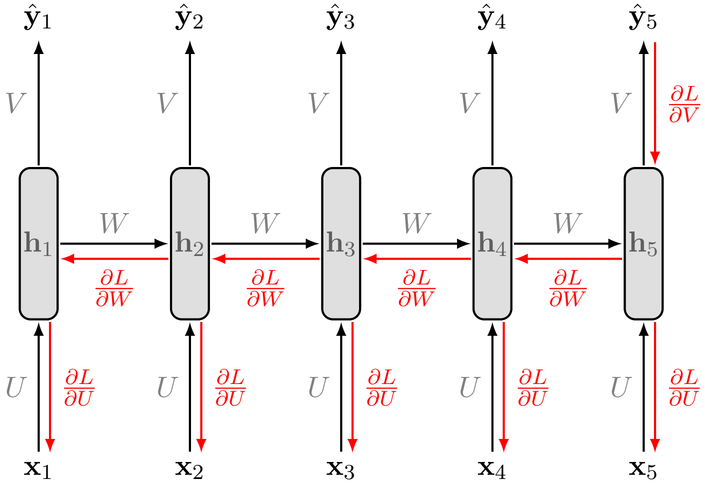


$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_4$
$\hat{\mathbf{y}}_5$
V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_4$
$\mathbf{h}_5$
W
$\frac{\partial L}{\partial W}$
$\frac{\partial L}{\partial W}$
U
$\frac{\partial L}{\partial U}$
$\frac{\partial L}{\partial U}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{x}_5$

BPTT


$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_4$
$\hat{\mathbf{y}}_5$
V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_4$
$\mathbf{h}_5$
W
$\frac{\partial L}{\partial W}$
U
$\frac{\partial L}{\partial U}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{x}_5$

BPTT


$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_4$
$\hat{\mathbf{y}}_5$
V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_4$
$\mathbf{h}_5$
W
$\frac{\partial L}{\partial W}$
U
$\frac{\partial L}{\partial U}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{x}_5$

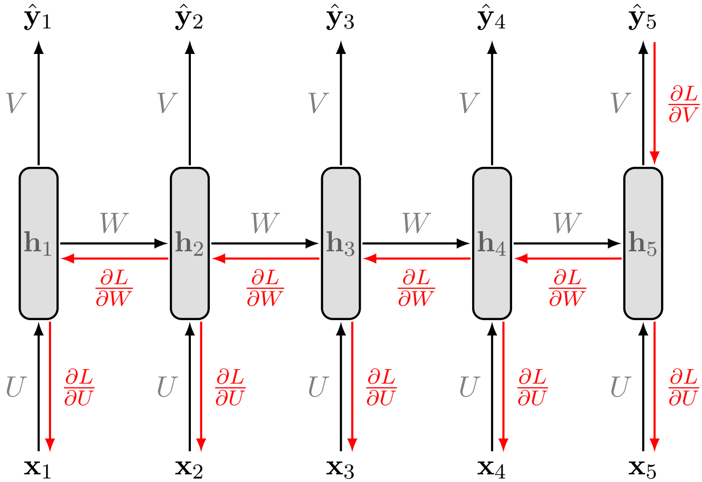
Gradient Computation

$$\nabla_V L = \sum_t (\nabla_{\mathbf{o}_t} L)\mathbf{h}_t^T$$

Where:

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

BPTT



$\hat{\mathbf{y}}_1$ $\hat{\mathbf{y}}_2$ $\hat{\mathbf{y}}_3$ $\hat{\mathbf{y}}_4$ $\hat{\mathbf{y}}_5$
V V V V V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$ $\mathbf{h}_2$ $\mathbf{h}_3$ $\mathbf{h}_4$ $\mathbf{h}_5$
W W W W
$\frac{\partial L}{\partial W}$ $\frac{\partial L}{\partial W}$ $\frac{\partial L}{\partial W}$ $\frac{\partial L}{\partial W}$
U U U U U
$\frac{\partial L}{\partial U}$ $\frac{\partial L}{\partial U}$ $\frac{\partial L}{\partial U}$ $\frac{\partial L}{\partial U}$ $\frac{\partial L}{\partial U}$
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$ $\mathbf{x}_5$

Gradient Computation

$$\nabla_W L = \sum_t diag\big(1 - (\mathbf{h}_t)^2\big)(\nabla_{\mathbf{h}_t} L)\mathbf{h}_{t-1}^T$$

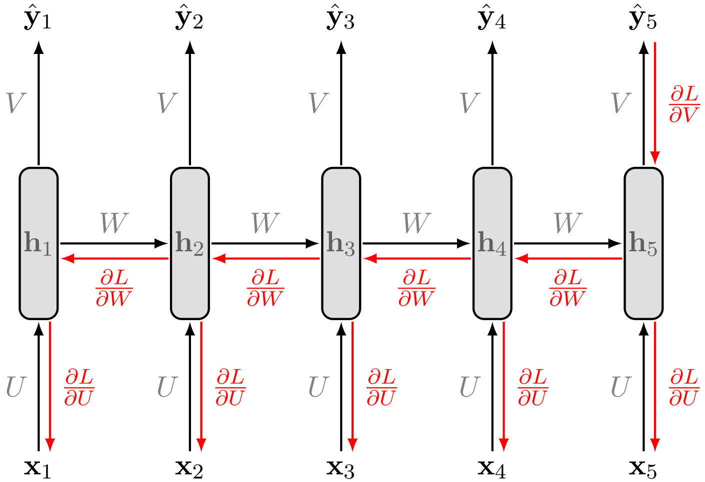
Where, for $t = \tau$ (one descendant):

$$(\nabla_{\mathbf{h}_\tau} L) = V^T (\nabla_{\mathbf{o}_\tau} L)$$

For some $t < \tau$ (two descendants)

$$\begin{aligned}(\nabla_{\mathbf{h}_t} L) &= \Big(\frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\Big)^T (\nabla_{\mathbf{h}_{t+1}} L) + \Big(\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\Big)^T (\nabla_{\mathbf{o}_t} L) \\ &= W^T (\nabla_{h_{t+1}} L) diag(1 - \mathbf{h_{t+1}}^2) + V(\nabla_{\mathbf{o}_t} L)\end{aligned}$$

BPTT



$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_4$
$\hat{\mathbf{y}}_5$
V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_3$
$\mathbf{h}_4$
$\mathbf{h}_5$
W
$\frac{\partial L}{\partial W}$
U
$\frac{\partial L}{\partial U}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{x}_5$

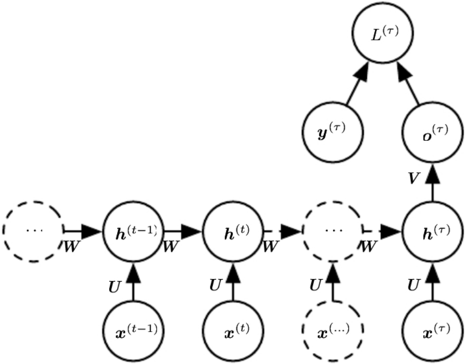
Gradient Computation

$$\nabla_U L = \sum_t diag\big(1 - (\mathbf{h}_t)^2\big)(\nabla_{\mathbf{h}_t} L)\mathbf{x}_t^T$$

Recurrent Neural Networks

- But weights are shared across different time stamps? How is this constraint enforced?
- Train the network as if there were no constraints, obtain weights at different time stamps, average them

Design Patterns of Recurrent Networks



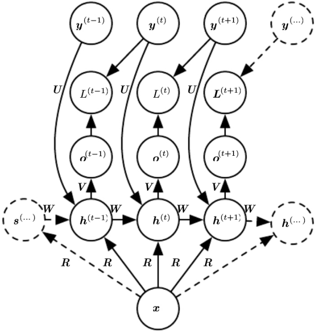
- **Summarization:** Produce a single output and have recurrent connections from output between hidden units
- Useful for summarizing a sequence (e.g. sentiment analysis)

Design Patterns: Fixed vector as input

- We have considered RNNs in the context of a sequence of vectors $x^{(t)}$ with $t = 1, \ldots, \tau$ as input
- Sometimes we are interested in only taking a single, fixed sized vector x as input, that generates the y sequence
- Some common ways to provide an extra input to an RNN are:
  - As an extra input at each time step
  - As the initial state $h^{(0)}$
  - both

Design Patterns: Fixed vector as input

- The first option (extra input at each time step) is the most common:



- Maps a fixed vector x into a distribution over sequences Y ($x^T R$ effectively is a new bias parameter for each hidden unit)

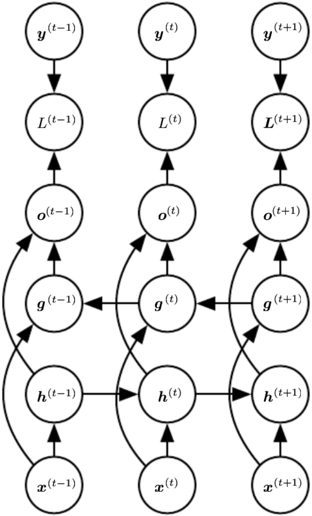
Application: Caption Generation

Caption Generation

Design Patterns: Bidirectional RNNs

- RNNs considered till now, all have a causal structure: state at time t only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$
- Sometimes we are interested in an output $y^{(t)}$ which may depend on the *whole input sequence*
- Example: Interpretation of a current sound as a phoneme may depend on the next few due to co-articulation
- Basically, in many cases we are interested in looking into the future as well as the past to disambiguate interpretations
- Bidirectional RNNs were introduced to address this need (Schuster and Paliwal, 1997), and have been used in handwriting recognition (Graves 2012, Graves and Schmidhuber 2009), speech recognition (Graves and Schmidhuber 2005) and bioinformatics (Baldi 1999)

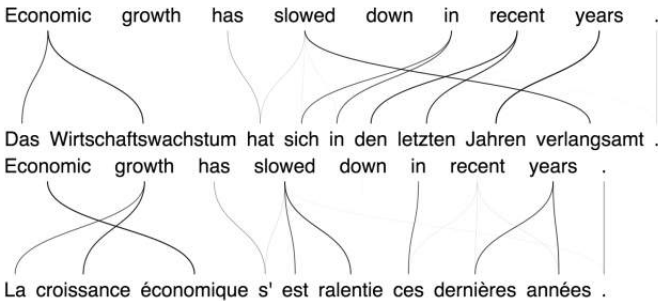
Design Patterns: Bidirectional RNNs



Design Patterns: Encoder-Decoder

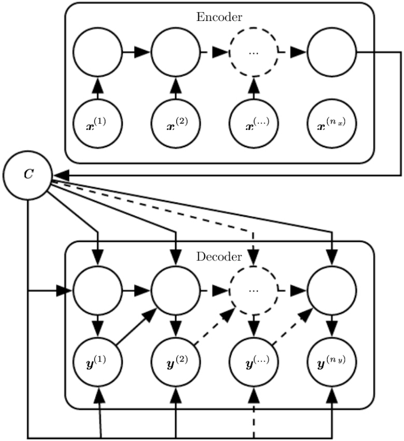
- How do we map input sequences to output sequences that are not necessarily of the same length?
- Example: Input - Kérem jöjjenek máskor és különösen máshoz. Output - 'Please come rather at another time and to another person.'
- Other example applications: Speech recognition, question answering etc.
- The input to this RNN is called the context, we want to find a representation of the context C
- C could be a vector or a sequence that summarizes $X = \{x^{(1)}, \ldots, x^{(n_x)}\}$

Design Patterns: Encoder-Decoder



- Far more complicated mappings

Design Patterns: Encoder-Decoder

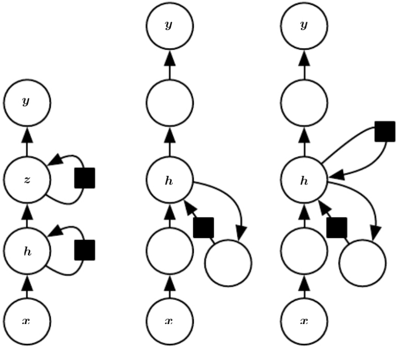


- In the context of Machine Trans. C is called a thought vector

Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
  - Input to hidden state
  - Previous hidden state to the next
  - Hidden state to the output
- Each of these transforms till now were learned affine transformations followed by a fixed nonlinearity
- Introducing depth in each of these operations is advantageous (Graves *et al.* 2013, Pascanu *et al.* 2014)
- The intuition on why depth should be more useful is quite similar to that in deep feed-forward networks
- Optimization can be made much harder, but can be mitigated by tricks such as introducing *skip connections*

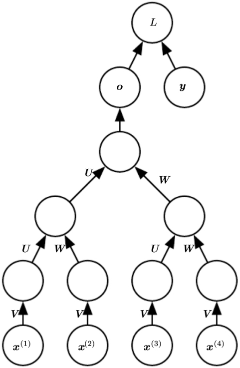
Deep Recurrent Networks



(a) (b) (c)

(b) lengthens shortest paths linking different time steps, (c) mitigates this by introducing skip layers

Recursive Neural Networks



- The computational graph is structured as a deep tree rather than as a chain in a RNN

Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi *et al* 1997), Natural Language Processing (Socher *et al* 2011) and Computer vision (Socher *et al* 2011)
- Advantage: For sequences of length τ, the number of compositions of nonlinear operations can be reduced from τ to $O(\log \tau)$
- Choice of tree structure is not very clear
  - A balanced binary tree, that does not depend on the structure of the data has been used in many applications
  - Sometimes domain knowledge can be used: Parse trees given by a parser in NLP (Socher *et al* 2011)
- The computation performed by each node need not be the usual neuron computation - it could instead be tensor operations etc (Socher *et al* 2013)

Long-Term Dependencies

Challenge of Long-Term Dependencies

- Basic problem: Gradients propagated over many stages tend to vanish (most of the time) or explode (relatively rarely)
- Difficulty with long term interactions (involving multiplication of many jacobians) arises due to exponentially smaller weights, compared to short term interactions
- The problem was first analyzed by Hochreiter and Schmidhuber 1991 and Bengio *et al* 1993

Challenge of Long-Term Dependencies

- Recurrent Networks involve the composition of the same function multiple times, once per time step
- The function composition in RNNs somewhat resembles matrix multiplication
- Consider the recurrence relationship:

$$h^{(t)} = W^T h^{(t-1)}$$

- This could be thought of as a very simple recurrent neural network without a nonlinear activation and lacking x
- This recurrence essentially describes the power method and can be written as:

$$h^{(t)} = (W^t)^T h^{(0)}$$

Challenge of Long-Term Dependencies

- If W admits a decomposition $W = Q\Lambda Q^T$ with orthogonal Q
- The recurrence becomes:

$$h^{(t)} = (W^t)^T h^{(0)} = Q^T \Lambda^t Q h^{(0)}$$

- Eigenvalues are raised to t: Quickly decay to zero or explode
- Problem particular to RNNs

Solution 1: Echo State Networks

- Idea: Set the recurrent weights such that they do a *good job* of capturing past history and learn only the output weights
- Methods: Echo State Machines, Liquid State Machines
- The general methodology is called reservoir computing
- How to choose the recurrent weights?

Echo State Networks

- **Original idea:** Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1
- In particular we pay attention to the spectral radius of J_t
- Consider gradient $\mathbf{g}$, after one step of backpropagation it would be $J\mathbf{g}$ and after n steps it would be $J^n\mathbf{g}$
- Now consider a perturbed version of $\mathbf{g}$ i.e. $\mathbf{g} + \delta\mathbf{v}$, after n steps we will have $J^n(\mathbf{g} + \delta\mathbf{v})$
- Infact, the separation is exactly $\delta|\lambda|^n$
- When $|\lambda > 1|$, $\delta|\lambda|^n$ grows exponentially large and vice-versa

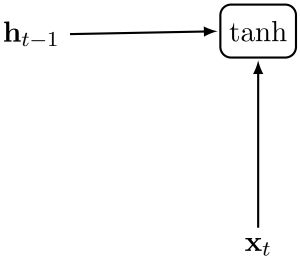
Echo State Networks

- For a vector $\mathbf{h}$, when a linear map W always shrinks $\mathbf{h}$, the mapping is said to be contractive
- The strategy of echo state networks is to make use of this intuition
- The Jacobian is chosen such that the spectral radius corresponds to stable dynamics

Other Ideas

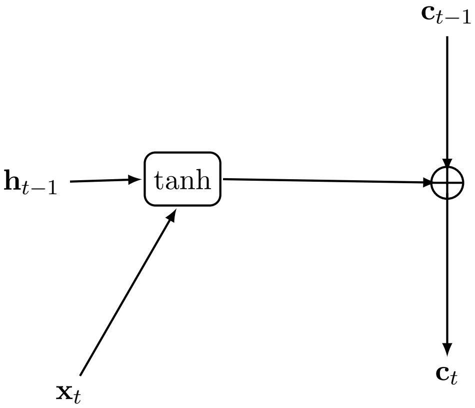
- Skip Connections
- Leaky Units

Long Short Term Memory



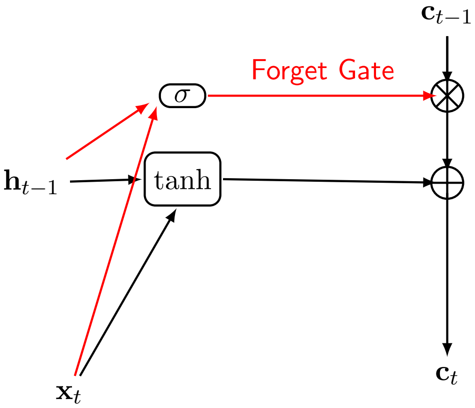
$$\mathbf{h}_t = \tanh(W\mathbf{h}_{t-1} + U\mathbf{x}_t)$$

Long Short Term Memory



$$\tilde{\mathbf{c}}_t = \tanh(W\mathbf{h}_{t-1} + U\mathbf{x}_t)$$
$$\mathbf{c}_t = \mathbf{c}_{t-1} + \tilde{c}_t$$

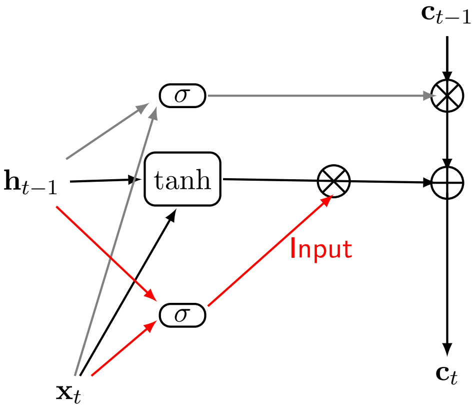
Long Short Term Memory



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

$$\tilde{\mathbf{c}}_t = \tanh(W \mathbf{h}_{t-1} + U \mathbf{x}_t)$$
$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + \tilde{c}_t$$

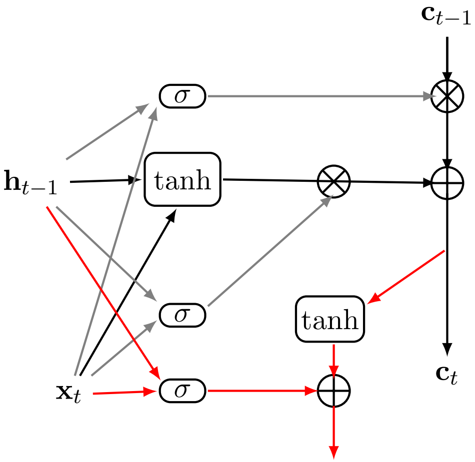
Long Short Term Memory



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

$$\tilde{\mathbf{c}}_t = \tanh(W \mathbf{h}_{t-1} + U \mathbf{x}_t)$$
$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{c}_t$$

Long Short Term Memory



$$f_t = \sigma(W_f \mathbf{h}_{t-1} + U_f \mathbf{x}_t)$$
$$i_t = \sigma(W_i \mathbf{h}_{t-1} + U_i \mathbf{x}_t)$$
$$o_t = \sigma(W_o \mathbf{h}_{t-1} + U_o \mathbf{x}_t)$$

$$\tilde{\mathbf{c}}_t = \tanh(W \mathbf{h}_{t-1} + U \mathbf{x}_t)$$
$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{c}_t$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Gated Recurrent Unit

- Let $\tilde{\mathbf{h}}_t = \tanh(W\mathbf{h}_{t-1} + U\mathbf{x}_t)$ and $\mathbf{h}_t = \tilde{\mathbf{h}}_t$
- Reset gate: $r_t = \sigma(W_r\mathbf{h}_{t-1} + U_r\mathbf{x}_t)$
- New $\tilde{\mathbf{h}}_t = \tanh(W(r_t \odot \mathbf{h}_{t-1}) + U\mathbf{x}_t)$
- Find: $z_t = \sigma(W_z\mathbf{h}_{t-1} + U_z\mathbf{x}_t)$
- Update $\mathbf{h}_t = z_t \odot \tilde{\mathbf{h}}_t$
- Finally: $\mathbf{h}_t = (1 - z_t) \odot \mathbf{h}_{t-1} + z_t \odot \tilde{\mathbf{h}}_t$
- Comes from attempting to factor LSTM and reduce gates