Sequence Modeling: Recurrent and Recursive Nets

Topics

1. Sequential Data and RNN Overview
2. Unfolding Computational Graphs
3. Recurrent Neural Networks
4. Bidirectional RNNs
5. Encoder-Decoder Sequence-to-Sequence Architectures
6. Deep Recurrent Networks
7. Recursive Neural Networks
8. The Challenge of Long-Term Dependencies
9. Echo-State Networks
10. Leaky Units and Other Strategies for Multiple Time Scales

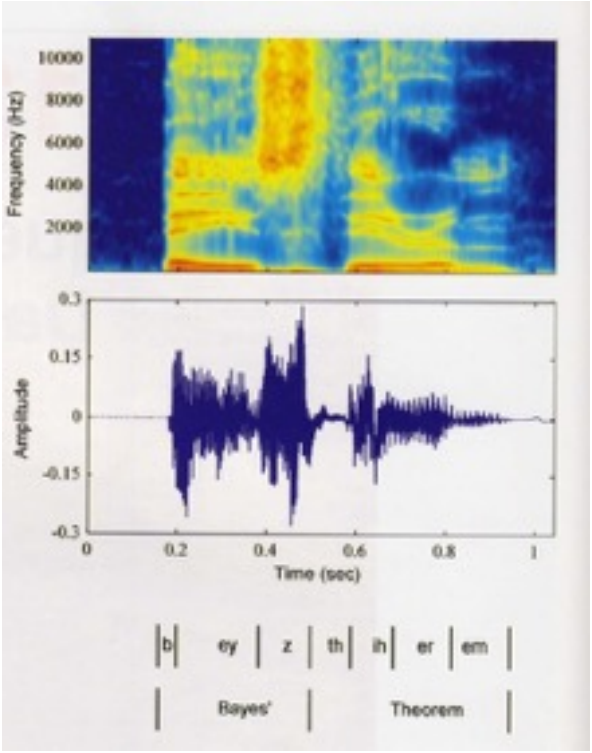
10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Sequential Data Examples

- Often arise through measurement of time series
  - Acoustic features at successive time frames in speech recognition
  - Sequence of characters in an English sentence
  - Parts of speech of successive words
  - Snowfall measurements on successive days
  - Rainfall measurements on successive days
  - Daily values of currency exchange rate
  - Nucleotide base pairs in a strand of DNA

Sound Spectrogram to Word Sequence

Bayes Theorem

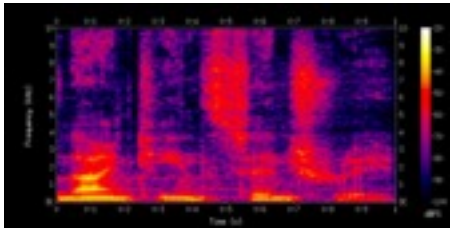


- Decompose sound waves into frequency, amplitude using Fourier transforms
- Plot of the intensity of the spectral coefficients versus time index
- Successive observations of speech spectrum highly correlated (Markov dependency)

Two common tasks with sequential data

1. Sequence-to-sequence
   1. Speech recognition using a sound spectrogram
      - decompose sound waves into frequency, amplitude using Fourier transforms

        → "Nineteenth Century"



Frequencies increase up the vertical axis, and time on the horizontal axis.
The lower frequencies are more dense because it is a male voice.
Legend on right shows that the color intensity increases with density

   1. NLP: Named Entity Recognition
      - Input: Jim bought 300 shares of Acme Corp. in 2006
      - NER: $[\text{Jim}]_{\text{Person}}$ bought 300 shares of $[\text{Acme Corp.}]_{\text{Organization}}$ in $[2006]_{\text{Time}}$
   2. Machine Translation: Echte dicke kiste → Awesome sauce

2. Sequence-to-symbol
   1. Sentiment:
      - *Best movie ever* → Positive
   2. Speaker recognition
      - Sound spectrogram → Harry

Recurrent Neural Networks process sequential data

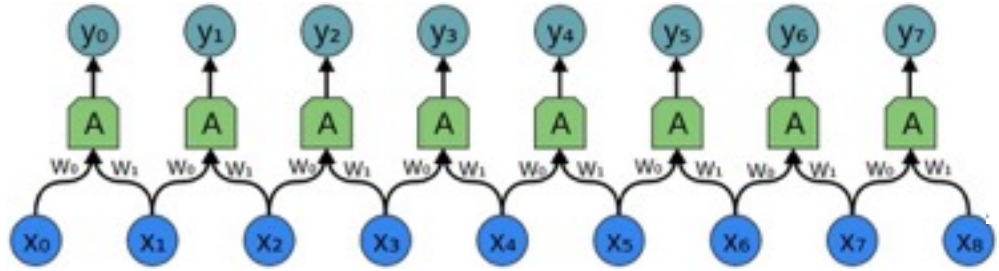
- RNNs are a family of neural nets for sequential data
- Analogy with Convolutional Neural Networks
  - Specialized architectures
    - CNN is specialized for grid of values, e.g., image
    - RNN is specialized for a sequence of values $\boldsymbol{x}^{(1)},..,\boldsymbol{x}^{(\tau)}$
  - Scaling & Variable length
    - CNNs readily scale to images with large width/height
      - CNNs can process variable size images
    - RNNs scale to longer sequences than would be practical for networks without sequence-based specialization
      - RNNs can also process variable-length sequences

RNNs share same weights across Time Steps

- To go from multi-layer networks to RNNs:
  - Need to share parameters across different parts of a model
  - Separate parameters for each value of cannot generalize to sequence lengths not seen during training
  - Share statistical strength across different sequence lengths and across different positions in time

- Sharing important when information can occur at multiple positions in the sequence
  - Given "*I went to Nepal in* 1999 " and "*In* 1999, *I went to Nepal* ", an ML method to extract year, should extract 1999 whether in position 6 or 2
  - A feed-forward network that processes sentences of fixed length would have to learn all of the rules of language separately at each position
  - An RNN shares the same weights across several time steps

1-D CNN used with a time sequence



Time Sequence

Kernel $g(t)$:

$[\ldots 0, w_1, w_0, 0 \ldots]$

Equations for outputs of this network:

$$y_0 = \sigma\,(w_0 x_0 + w_1 x_1 - b)$$
$$y_1 = \sigma\,(w_0 x_1 + w_1 x_2 - b)$$

etc. upto y_8

Note that kernel gets flipped in convolution

We can also write the equations in terms of elements of a general 8×8 weight matrix W as:

$$y_0 = \sigma(W_{0,0}x_0 + W_{0,1}x_1 + W_{0,2}x_2 \ldots)$$

$$y_1 = \sigma(W_{1,0}x_0 + W_{1,1}x_1 + W_{1,2}x_2 \ldots)$$

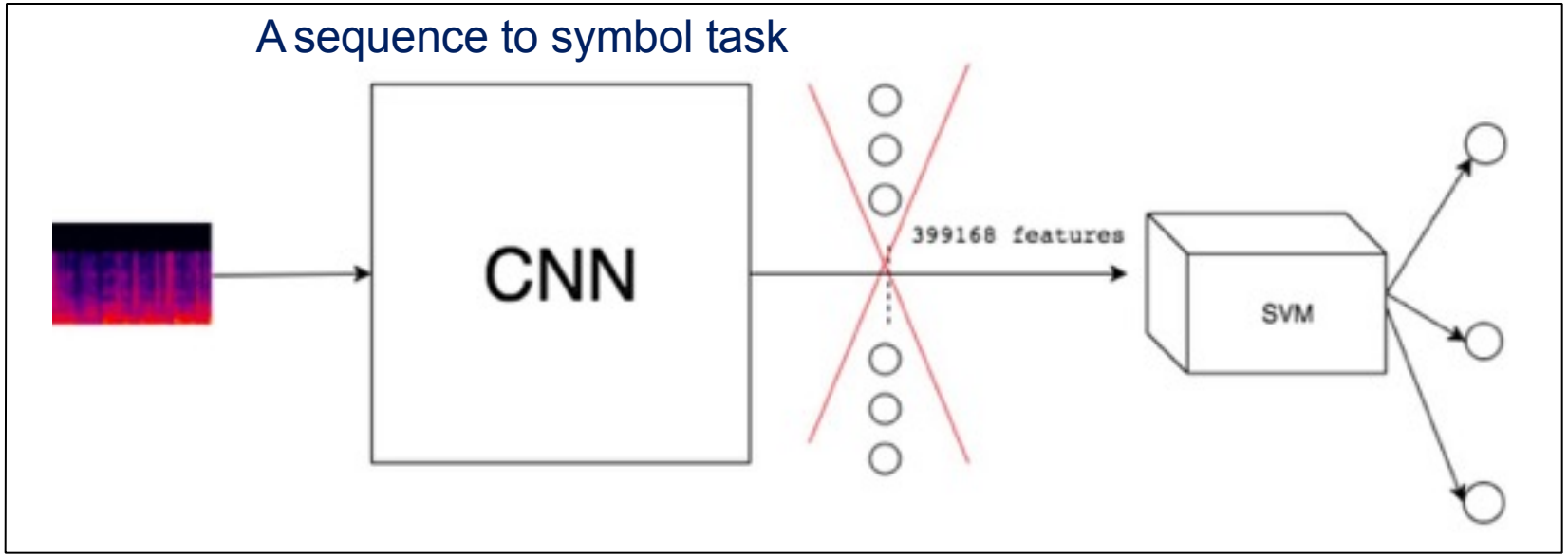
etc. upto y_8

where

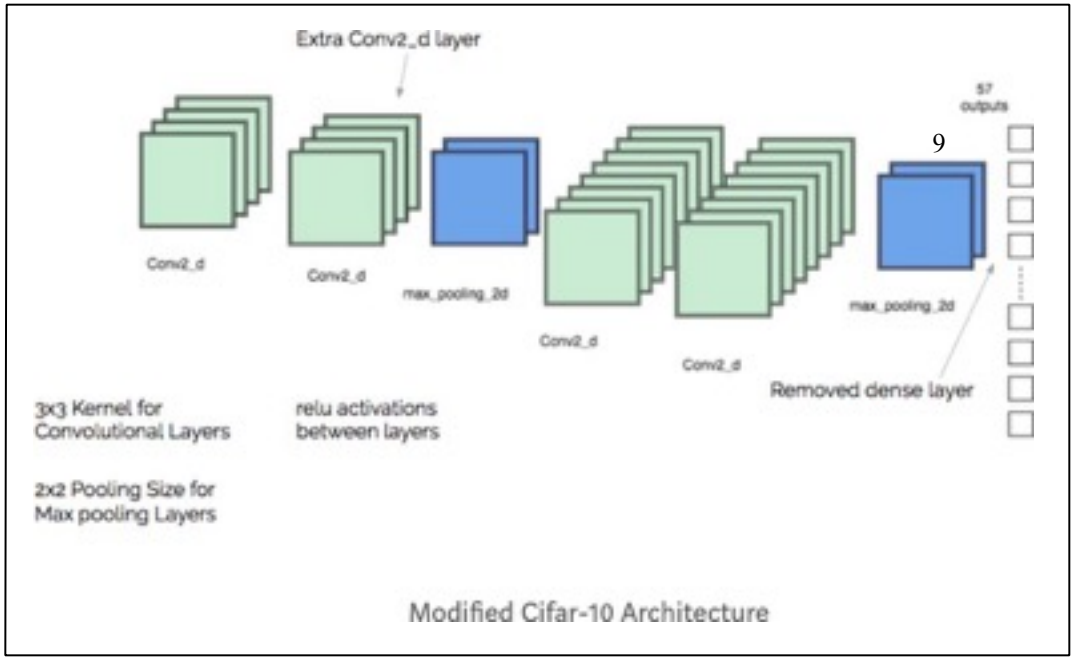
$$W = \begin{bmatrix} w_0 & w_1 & 0 & 0 & \ldots \\ 0 & w_0 & w_1 & 0 & \ldots \\ 0 & 0 & w_0 & w_1 & \ldots \\ 0 & 0 & 0 & w_0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{bmatrix}$$

http://colah.github.io/posts/2014-07-Understanding-Convolutions/

Speaker Recognition with CNN

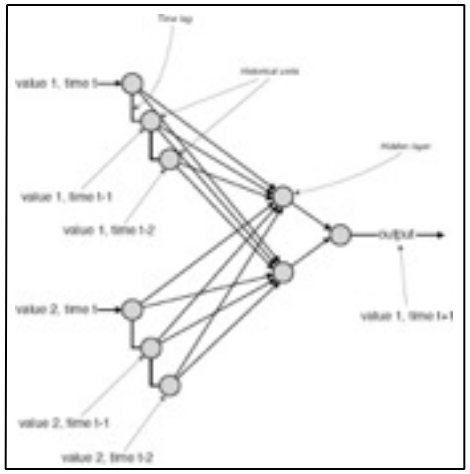


CNN
Feature Extractor
(Transfer Learning)



Time Delay Neural Networks (TDNNs)

- TDNNs use convolution for a 1-D temporal sequence
  - Convolution allows shared parameters across time, but is shallow
    - Each output is dependent upon a small no. of neighboring inputs
    - Parameter sharing manifests in the application of the same convolutional kernel at each time step

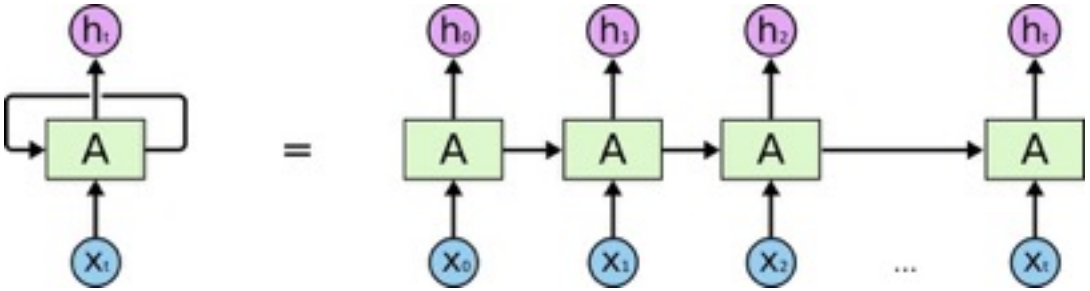


A TDNN remembers the previous few training examples and uses them as input into the network.
The network then works like a feed-forward, back propagation network.

RNN vs. TDNN

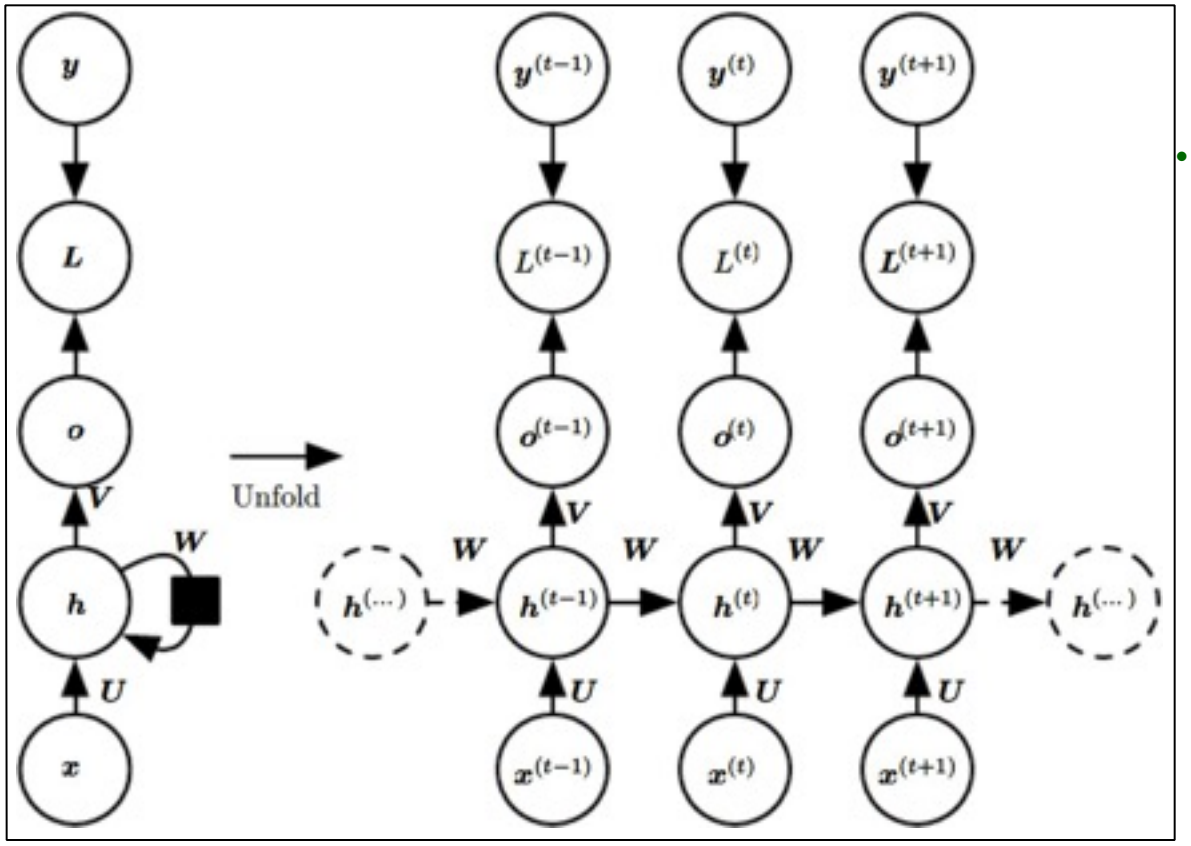
- RNNs share parameters in a different way than TDNN
  - Each member of output is a function of previous members of output
  - Each output produced using same update rule applied to previous outputs
  - This recurrent formulation results in sharing of parameters through a very deep computational graph

- An unrolled RNN



Computational Graphs for RNNs

- We extend computational graphs to include cycles
  - Cycles represent the influence of the present value of a variable on its own value at a future time step
  - In a Computational graph nodes are variables/operations
  - RNN to map input sequence of $\boldsymbol{x}$ values to output sequence of $\boldsymbol{o}$ values
    - Loss L measures how far each output $\boldsymbol{o}$ is from the training target $\boldsymbol{y}$



- Forward propagation is given as follows:

For each time step t, t=1 to t=τ

Apply the following equations

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + V\boldsymbol{h}^{(t)}$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)})$$

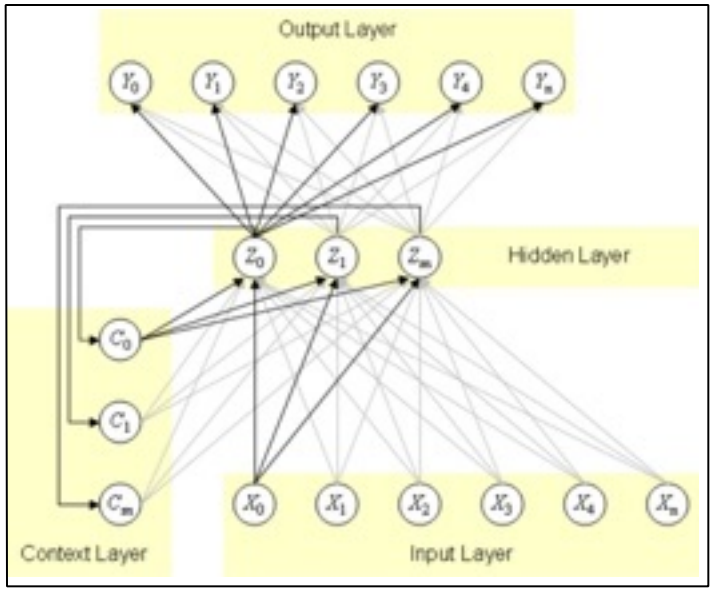
$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)}$$

RNN operating on a sequence

- RNNs operate on a sequence that contain vector $\boldsymbol{x}^{(t)}$ with time step index t, ranging from 1 to τ
  - Sequence: $\boldsymbol{x}^{(1)},..,\boldsymbol{x}^{(\tau)}$
  - RNNs operate on minibatches of sequences of length τ
- Some remarks about sequences
  - The steps need not refer to passage of time in the real world
  - RNNs can be applied in two-dimensions across spatial data such as image
  - Even when applied to time sequences, network may have connections going backwards in time, provided entire sequence is observed before it is provided to network

RNN as a network with cycles

- An RNN is a class of neural networks where connections between units form a directed cycle
- This creates an internal state of the network which allows it to exhibit dynamic temporal behavior
- The internal memory can be used to process arbitrary sequences of inputs



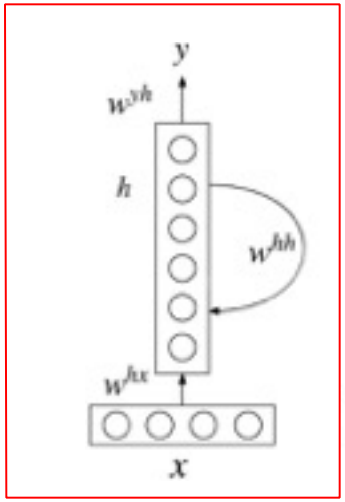
Three layer network with input ***x*** , hidden layer ***z*** and output ***y***

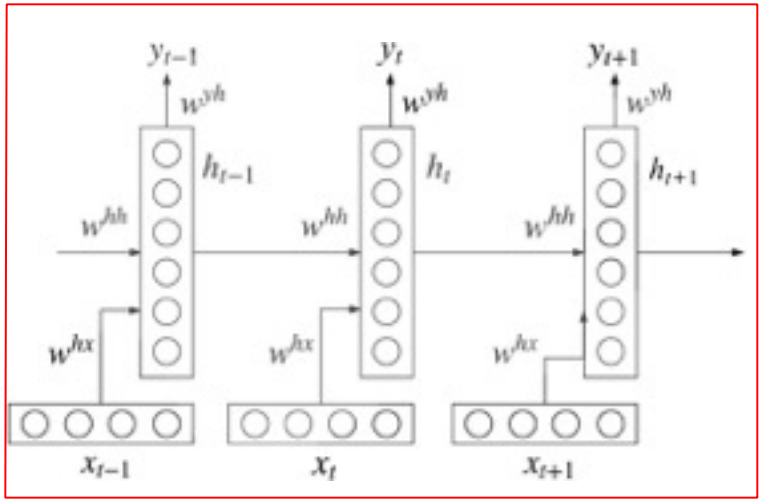
Context units ***c*** maintain a copy of the previous value of the hidden units

RNN parameters

Folded network with cycles

Unfolded sequence network with three time steps



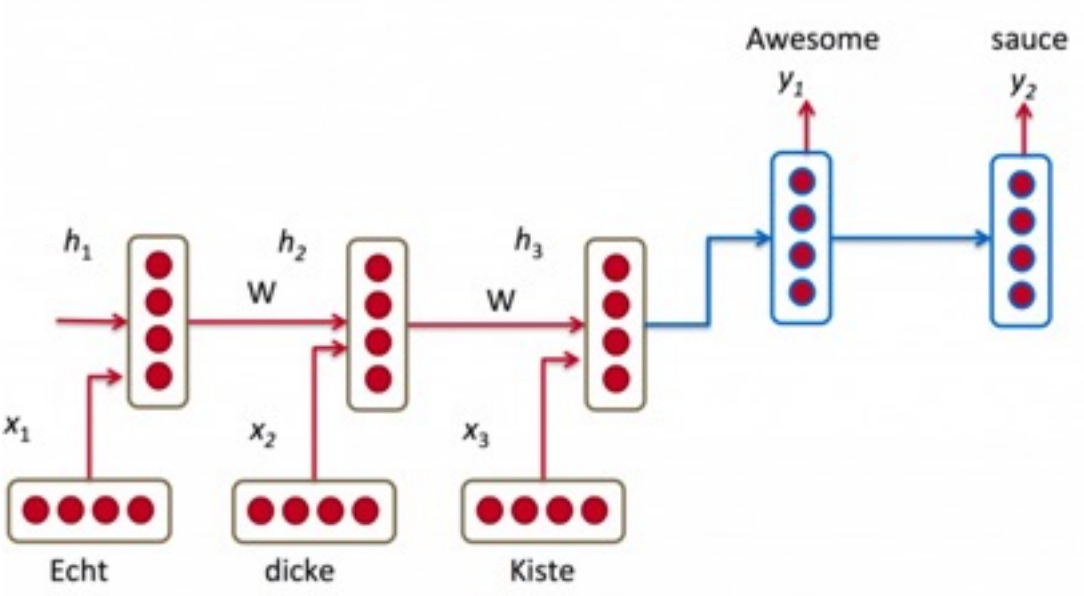


$$h_t = f(w^{hh} h_{t-1} + w^{hx} x_t)$$

$$y_t = \text{softmax}(w^{yh} h_t)$$

Unlike a feedforward neural network, which uses different parameters at each layer, RNN shares the same parameters (W^{hx}, W^{hh}, W^{yh}) across all steps

RNN for Machine Translation

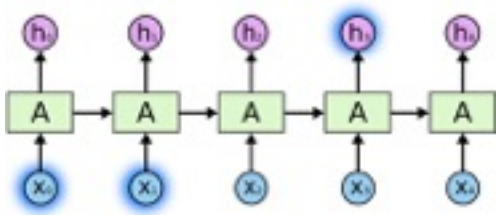


Limitation of length of time steps

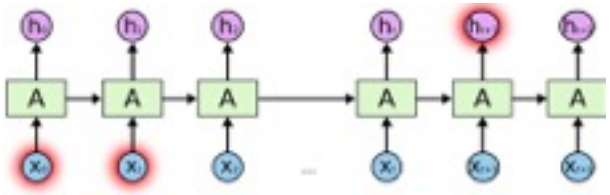
- Length of time steps is determined by length of input
  - e.g., if word sequence to be processed is a sentence of six words, RNN would be unfolded into a neural net with six time steps or layers.
    - One layer corresponds to a word
- Theoretically, RNN can make use of the information in arbitrarily long sequences
  - In practice, RNN is limited to looking back only a few steps due to the vanishing gradient or exploding gradient problem

Problem of Long-Term Dependencies

- Easy to predict last word in "the clouds are in the *sky*,"
  - When gap between relevant information and place that it's needed is small, RNNs can learn to use the past information

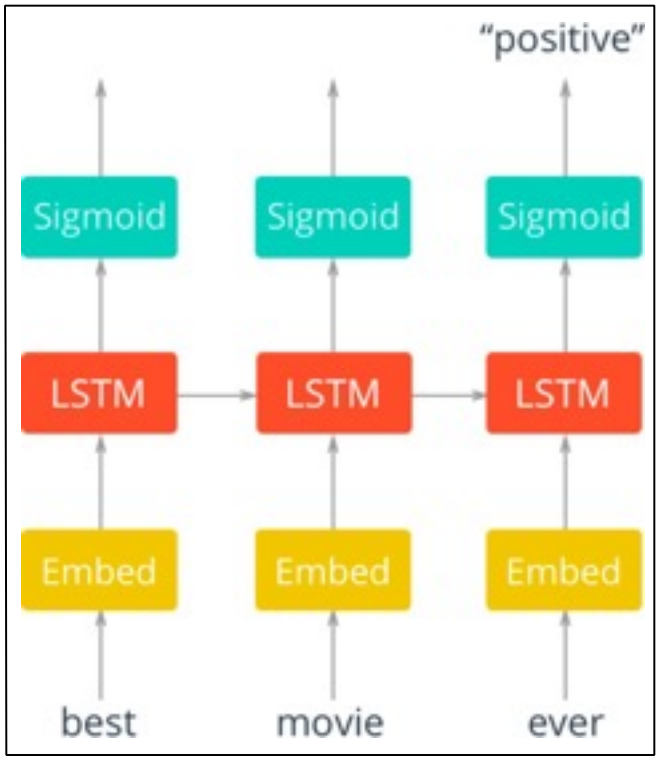


- "I grew up in France… I speak fluent *French*."
  - We need the context of France, from further back.
  - Large gap between relevant information and point where it is needed



- In principle RNNs can handle it, but fail in practice
  - LSTMs offer a solution

RNN for sentiment analysis



Embedding Layer

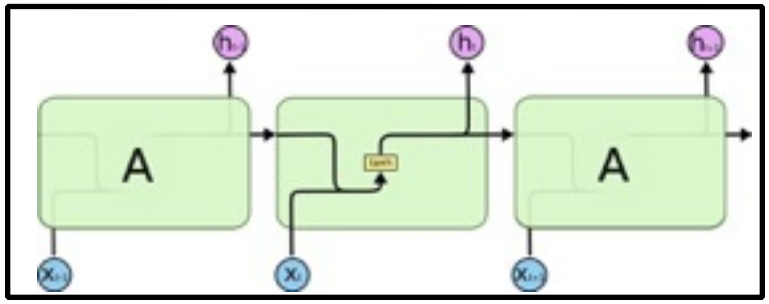
We pass in words to an embedding layer

Options

1. Actually train up an embedding with Word2vec
2. It is good enough to just have an embedding layer and let network learn the embedding table on its own.

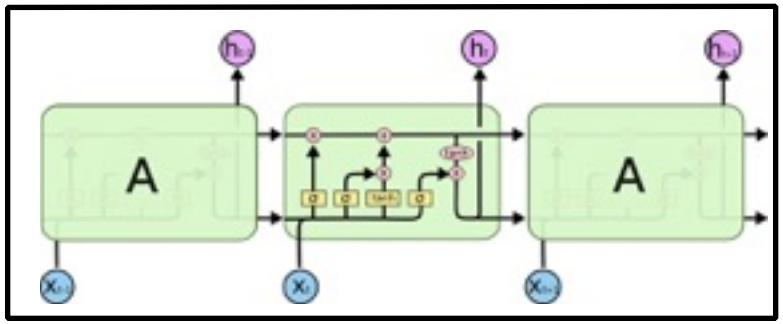
LSTM:

RNN vs LSTM



RNN

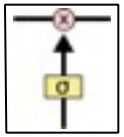
Repeating module has a simple structure Such as a tanh layer



LSTM

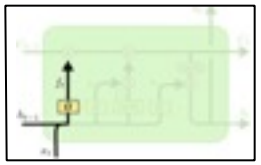
Repeating module has four interacting Layers, with notation:

Three gates of the form



Sigmoid is 0 or 1
Allows input to go through or not

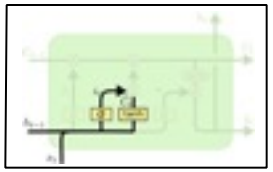
Forget gate



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

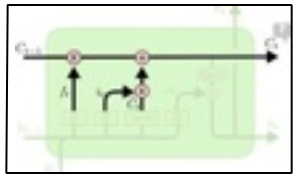
Forget gender of old subject

Input gate



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

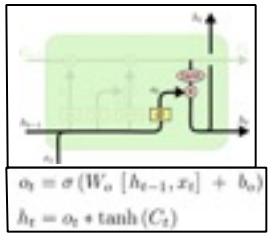
Input gender of new subject



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

Actually drop old
Add new

Output gate



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

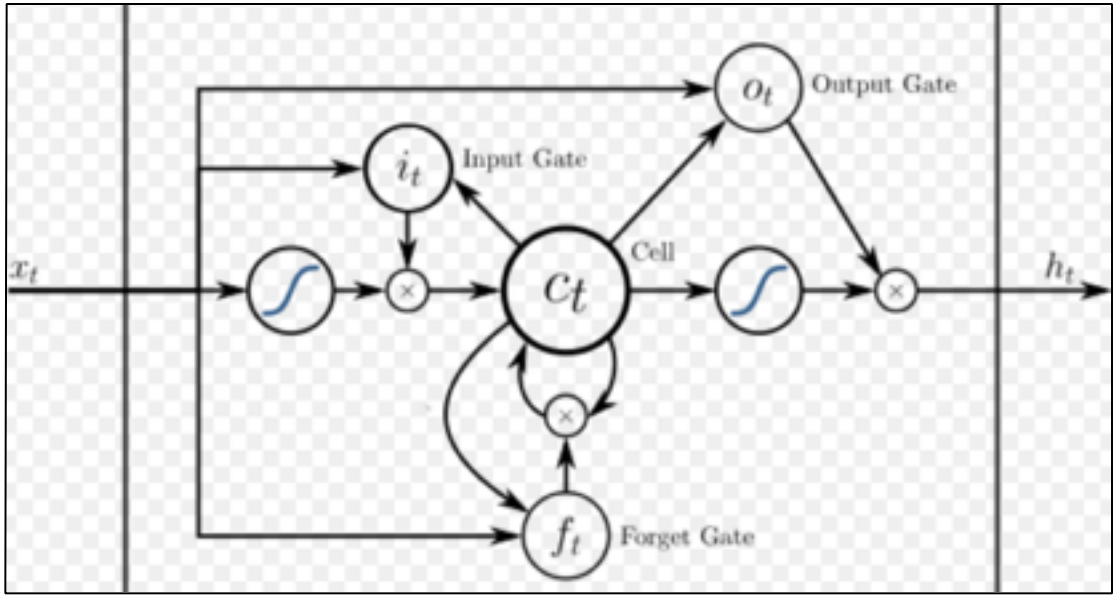
Whether subject is singular or plural

An LSTM variant

- A common LSTM unit is composed of
  - a **cell**, an **input gate**, an **output gate** and a **forget gate**

    The cell remembers values over arbitrary time intervals and

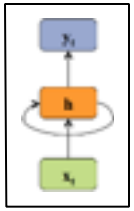
    the three *gates* regulate the flow of information into and out of the cell.
- A peephole LSTM is shown below



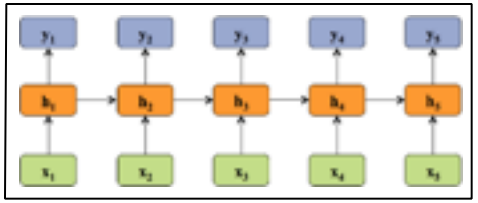
Summary of Neural Sequential Models

Recurrent Neural Network

RNN



Unrolled RNN



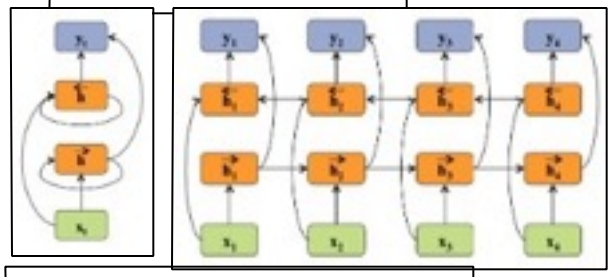
Definition

$$\begin{aligned} \text{inputs} &: x = (x_1, x_2, ..., x_T), x_i \in \mathbb{R}^I \\ \text{hidden units} &: h = (h_1, h_2, ..., h_T), h_i \in \mathbb{R}^J \\ \text{outputs} &: y = (y_1, y_2, ..., y_T), y_i \in \mathbb{R}^K \\ \text{nonlinearity} &: \mathcal{H} \end{aligned}$$

Activation Functions

$$\begin{aligned} h_t &= \mathcal{H}(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ y_t &= W_{hy}h_t + b_y \end{aligned}$$

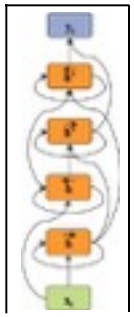
Bidirectional RNN

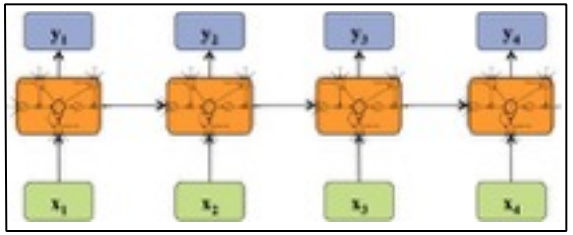


$$\begin{aligned} \text{inputs} &: x = (x_1, x_2, ..., x_T), x_i \in \mathbb{R}^I \\ \text{hidden units} &: \overrightarrow{h} \text{ and } \overleftarrow{h} \\ \text{outputs} &: y = (y_1, y_2, ..., y_T), y_i \in \mathbb{R}^K \\ \text{nonlinearity} &: \mathcal{H} \end{aligned}$$

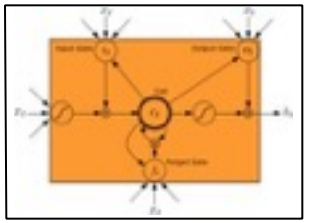
$$\begin{aligned} \overrightarrow{h}_t &= \mathcal{H}(W_{x\overrightarrow{h}}x_t + W_{\overrightarrow{h}\overrightarrow{h}}\overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}) \\ \overleftarrow{h}_t &= \mathcal{H}(W_{x\overleftarrow{h}}x_t + W_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{h}_{t-1} + b_{\overleftarrow{h}}) \\ y_t &= W_{\overleftarrow{h}y}\overleftarrow{h}_t + W_{\overrightarrow{h}y}\overrightarrow{h}_t + b_y \end{aligned}$$

Deep Bidirectional RNN





LSTM



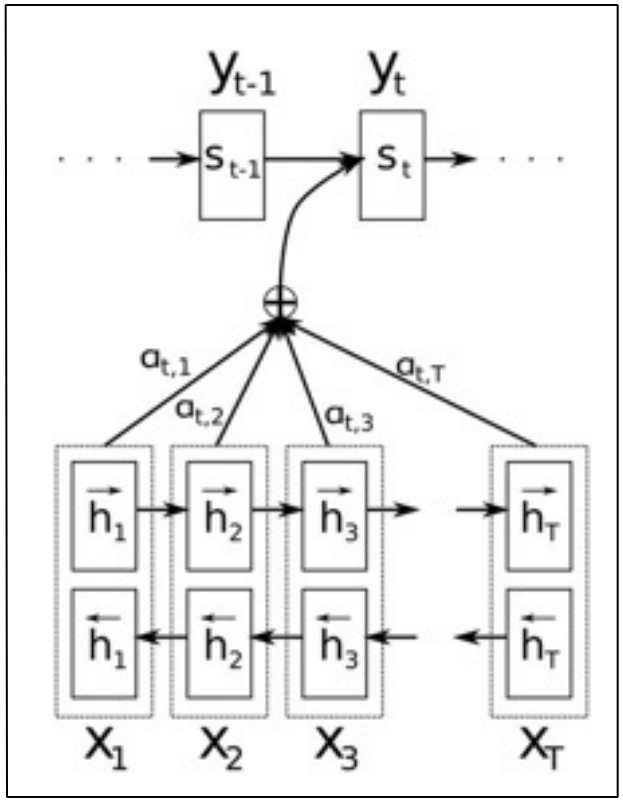
Source: http://www.cs.cmu.edu/~epxing/Class/10708-16/note/10708_scribe_lecture27.pdf

Attention Mechanisms

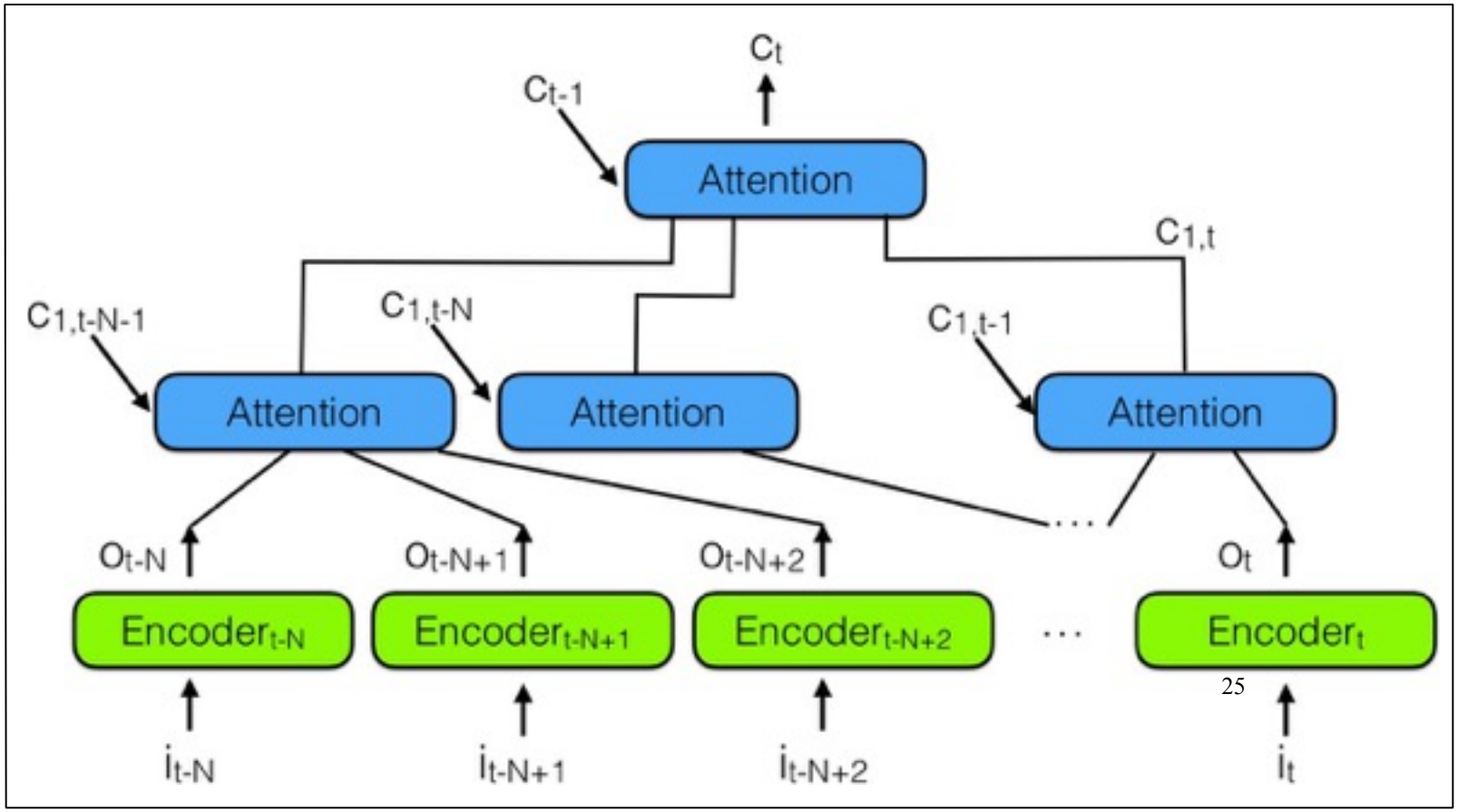
- Long range dependencies are still a problem with LSTMs
- With an attention mechanism we no longer try encode the full source sentence into a fixed-length vector
  - Rather, we allow the decoder to “attend” to different parts of the source sentence at each step of the output generation
- Attention is simply a vector, often the outputs of dense layer using softmax function.

Attention Mechanism in Translation

- y : translated words
- x : source words.
  - Use of bidirectional RNN is unimportant
- Each decoder output word now depends on
  - weighted combination of all the input states,
    - not just the last state.
  - a's are weights for each input state
    - if $a_{3,2}$ is large, decoder pays attention to second state in source sentence while producing third word of target
    - a's are normalized to sum to 1



Hierarchical neural attention encoder

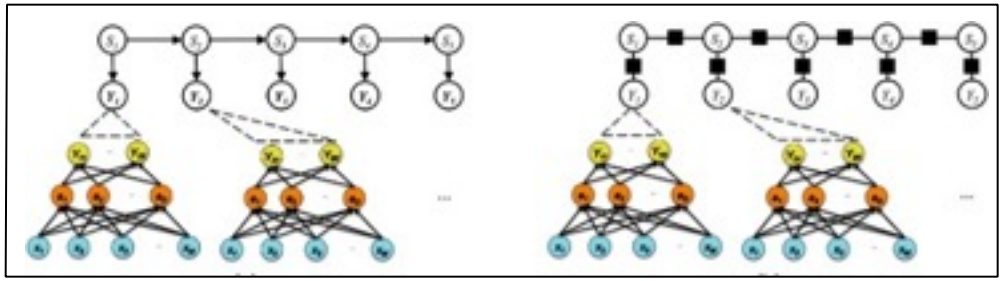


Deep Learning and Graphical Models

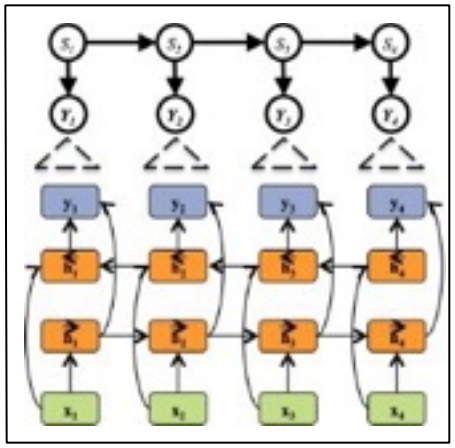
- In deep learning:
- Tasks of interest:
- Classification
  - Feature learning
  - Method of learning
    - Back propagation and gradient descent
- In graphical models:
  - Tasks of interest:
    - Transfer learning
    - Latent variable inference
  - Methods of learning
    - Parameter learning methods
    - Structure learning methods
- Hybrid graphical models combine the two types of models
  - They are trained using backpropagation

Hybrid Graphical Models and Neural Networks

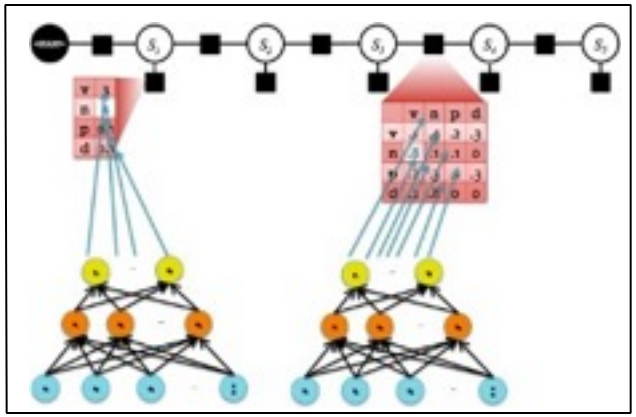
- Hybrid NN and HMM



- Hybrid RNN+HMM



Hybrid CNN+CRF



Unfolding Computational Graphs

Topics in Sequence Modeling

- Recurrent Neural Networks

1. Unfolding Computational Graphs
2. Recurrent Neural Networks
3. Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent Networks
6. Recursive Neural Networks
7. The Challenge of Long-Term Dependencies
8. Echo-State Networks
9. Leaky Units and Other Strategies for Multiple Time Scales

10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Unfolding Computational Graphs

- A Computational Graph is a way to formalize the structure of a set of computations
    - Such as mapping inputs and parameters to outputs and loss
- We can unfold a recursive or recurrent computation into a computational graph that has a repetitive structure
    - Corresponding to a chain of events
- Unfolding this graph results in sharing of parameters across a deep network structure

Example of unfolding a recurrent equation

- Classical form of a dynamical system is

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

  - where $\boldsymbol{s}^{(t)}$ is called the state of the system
  - Equation is recurrent because the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time t-1
- For a finite no. of time steps τ, the graph can be unfolded by applying the definition τ-1 times
  - E,g, for $\tau = 3$ time steps we get

$$\boldsymbol{s}^{(3)} = f\left(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}\right)$$

$$= f\left(f\left(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}\right); \boldsymbol{\theta}\right)$$

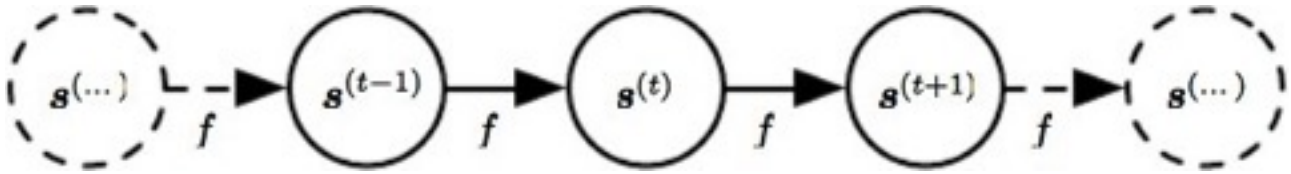
- Unfolding equation by repeatedly applying the definition in this way has yielded expression without recurrence
  - $\boldsymbol{s}^{(1)}$ is ground state and $\boldsymbol{s}^{(2)}$ computed by applying f
- Such an expression can be represented by a traditional acyclic computational graph (as shown next)

Unfolded dynamical system

- The classical dynamical system described by

  $$\boldsymbol{s}^{(t)} = f\,(\boldsymbol{s}^{(t-1)};\ \boldsymbol{\theta}\ ) \text{ and } \boldsymbol{s}^{(3)} = f\,(f\,(\boldsymbol{s}^{(1)};\ \boldsymbol{\theta});\ \boldsymbol{\theta})$$

  is illustrated as an unfolded computational graph



- Each node represents state at some time t
- Function f maps state at time t to the state at t+1
- The same parameters (the same value of $\boldsymbol{\theta}$ used to parameterize f) are used for all time steps

Dynamical system driven by external signal

- As another example, consider a dynamical system driven by external (input) signal $\boldsymbol{x}^{(t)}$

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

  - State now contains information about the whole past input sequence
  - Note that the previous dynamic system was simply $\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}\right)$

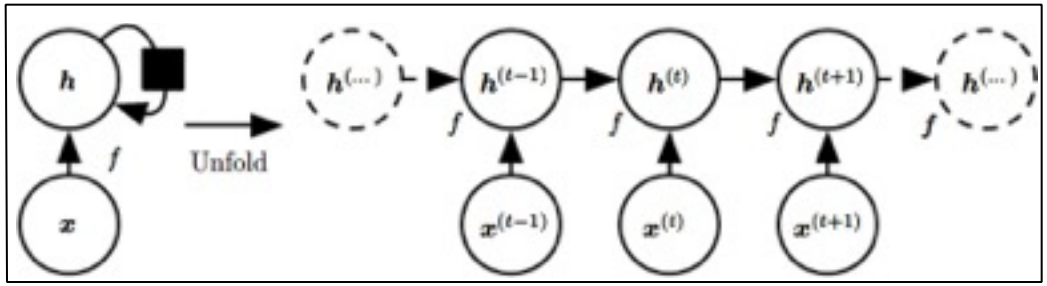
- Recurrent neural networks can be built in many ways
  - Much as almost any function is a feedforward neural network, any function involving recurrence can be considered to be a recurrent neural network

Defining values of hidden units in RNNs

- Many recurrent neural nets use same equation (as dynamical system with external input) to define values of hidden units
  - To indicate that the state is hidden rewrite using variable $\boldsymbol{h}$ for state:

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

  - Illustrated below:



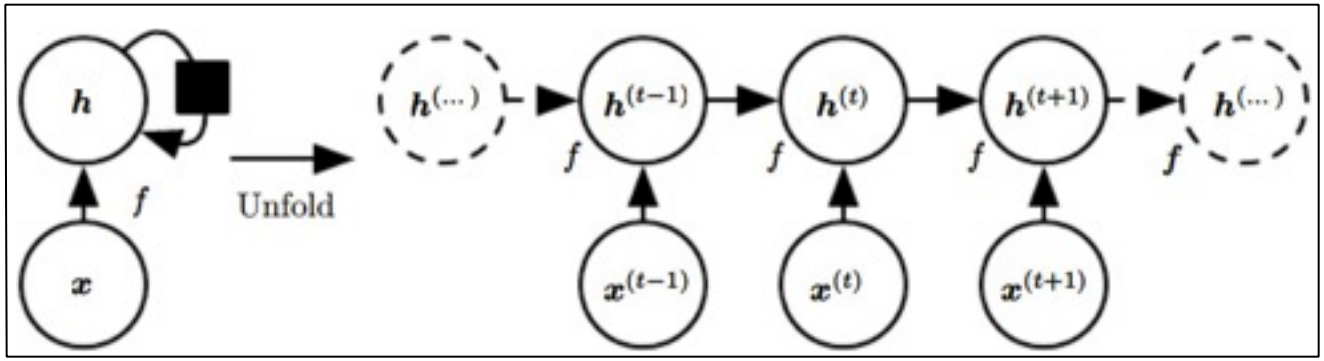
- Typical RNNs have extra architectural features such as output layers that read information out of state $\boldsymbol{h}$ to make predictions

A recurrent network with no outputs

- This network just processes information from input $\boldsymbol{x}$ by incorporating it into state $\boldsymbol{h}$ that is passed forward through time

Circuit diagram:
Black square indicates
Delay of one time step

Unfolded computational graph:
each node is now
associated with one time instance



Typical RNNs will add extra architectural features such as output layers to read information out of the state $\boldsymbol{h}$ to make predictions

Predicting the Future from the Past

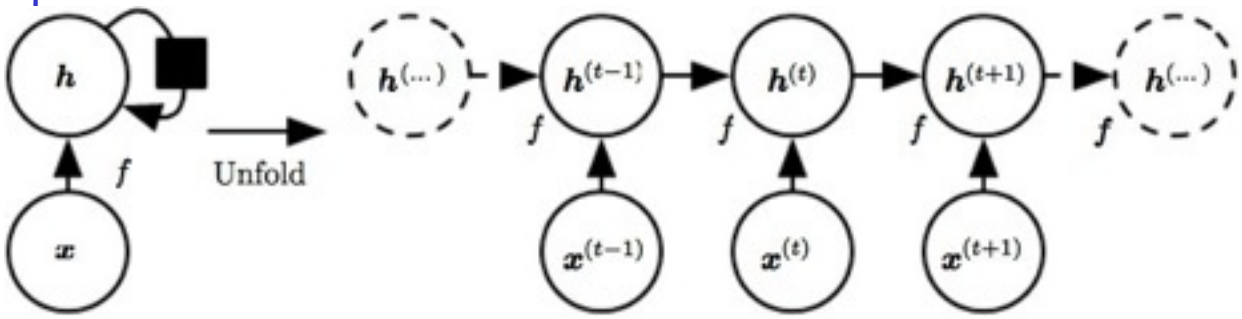
- When RNN is required to perform a task of predicting the future from the past, network typically learns to use $\boldsymbol{h}^{(t)}$ as a lossy summary of the task-relevant aspects of the past sequence of inputs upto t
- The summary is in general lossy since it maps a sequence of arbitrary length $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)},..,\boldsymbol{x}^{(2)},\boldsymbol{x}^{(1)})$ to a fixed length vector $\boldsymbol{h}^{(t)}$

Information Contained in Summary

- Depending on criterion, summary keeps some aspects of past sequence more precisely than other aspects
- Examples:
  - RNN used in statistical language modeling, typically to predict next word from past words
    - it may not be necessary to store all information upto time t but only enough information to predict rest of sentence
  - Most demanding situation: we ask $\boldsymbol{h}^{(t)}$ to be rich enough to allow one to approximately recover the input sequence as in autoencoders

Unfolding: from circuit diagram to computational graph

- Equation $\boldsymbol{h}^{(t)}=f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$ can be written in two different ways: circuit diagram or an unfolded computational graph



- Unfolding is the operation that maps a circuit to a computational graph with repeated pieces
- The unfolded graph has a size dependent on the sequence length

Process of Unfolding

- We can represent unfolded recurrence after t steps with a function $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$$

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

- Function $g^{(t)}$ takes in whole past sequence $\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$ as input and produces the current state but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f
- The unfolding process introduces two major advantages as discussed next.

Unfolding process allows learning a single model

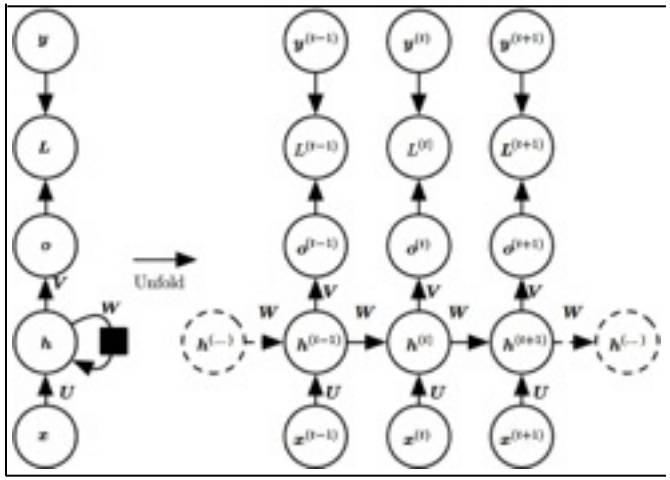
- The unfolding process introduces two major advantages:
  1. Regardless of sequence length, learned model has same input size
     - because it is specified in terms of transition from one state to another state rather than specified in terms of a variable length history of states
  2. Possible to use same function f with same parameters at every step
- These two factors make it possible to learn a single model f
  - that operates on all time steps and all sequence lengths
  - rather than needing separate model $g^{(t)}$ for all possible time steps
- Learning a single shared model allows:
  - Generalization to sequence lengths that did not appear in the training
  - Allows model to be estimated with far fewer training examples than would be required without parameter sharing

Both recurrent graph and unrolled graph are useful

- Recurrent graph is succinct
- Unrolled graph provides explicit description of what computations to perform
  - Helps illustrate the idea of information flow forward in time
    - Computing outputs and losses
  - And backwards in time
    - Computing gradients
  - By explicitly showing path along which information flows

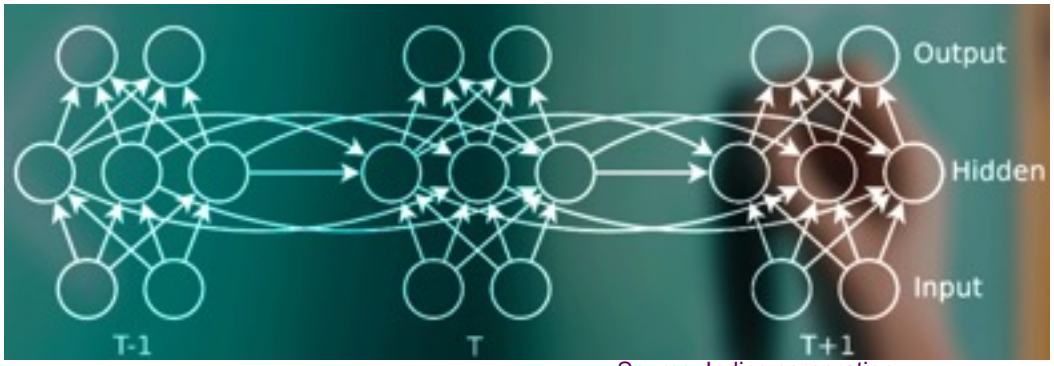
Unfolding Computational Graphs

- A Computational Graph is a way to formalize the structure of a set of computations
  - Such as mapping inputs and parameters to outputs and loss
- We can unfold a recursive or recurrent computation into a computational graph that has a repetitive structure
  - Corresponding to a chain of events
- Unfolding this graph results in sharing of parameters across a deep network structure



$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + V\boldsymbol{h}^{(t)}$$
$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)})$$
$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)}$$

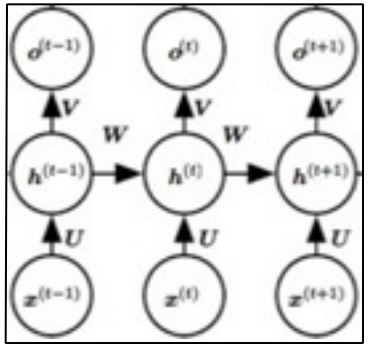
A more complex unfolded computational graph



Source: Indico corporation

Two units in input layer, instead of 1
Three units in hidden layer, instead of 1
Two units in output layer, instead of 1

Previous one:



Recurrent Neural Networks

Topics in Sequence Modeling

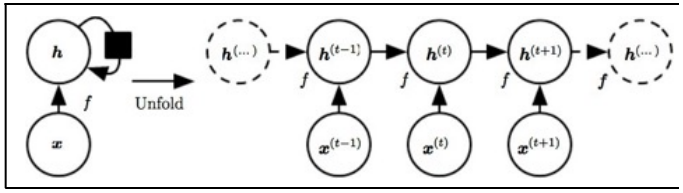
- Overview

1. Unfolding Computational Graphs
2. Recurrent Neural Networks
3. Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent Networks
6. Recursive Neural Networks
7. The Challenge of Long-Term Dependencies
8. Echo-State Networks
9. Leaky Units and Other Strategies for Multiple Time Scales
10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Graph unrolling/parameter sharing in RNN design

- Process of graph unrolling:

$$\boldsymbol{h}^{(t)} = g^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}\right)$$
$$= f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right)$$



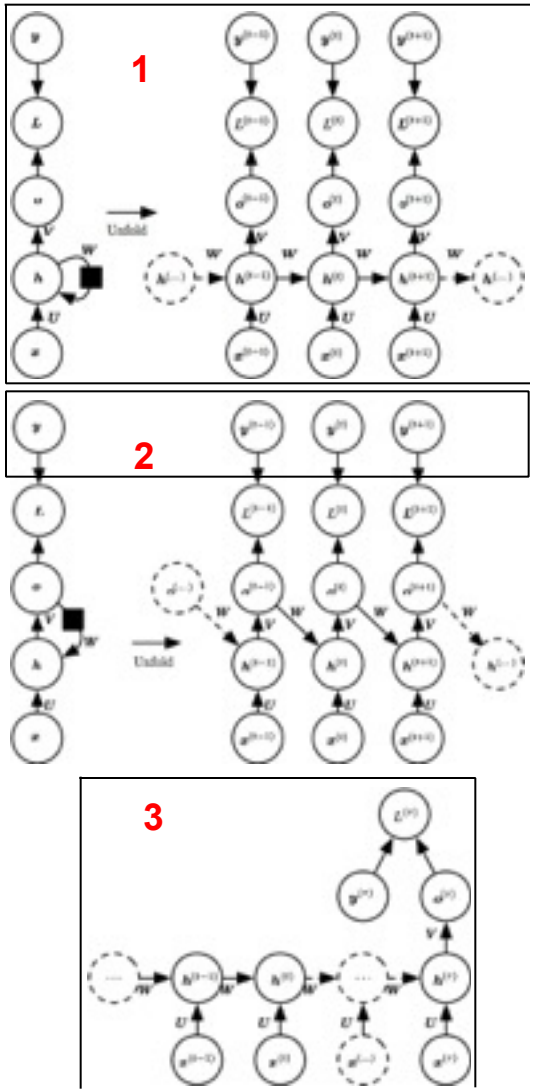
  - Function $g^{(t)}$ takes in whole past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, .., \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ as input and produces the current state but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f
  - It does not need a separate model $g^{(t)}$ for all possible time steps
- Process of unrolling and parameter sharing allows us to design a wide variety of recurrent neural networks
- Examples of important design patterns shown next

Three design patterns of RNNs

1. Produce output at each time step and have recurrent connections between hidden units

2. Produce output at each time step and have recurrent connections only from output at one time step to hidden units at next time step

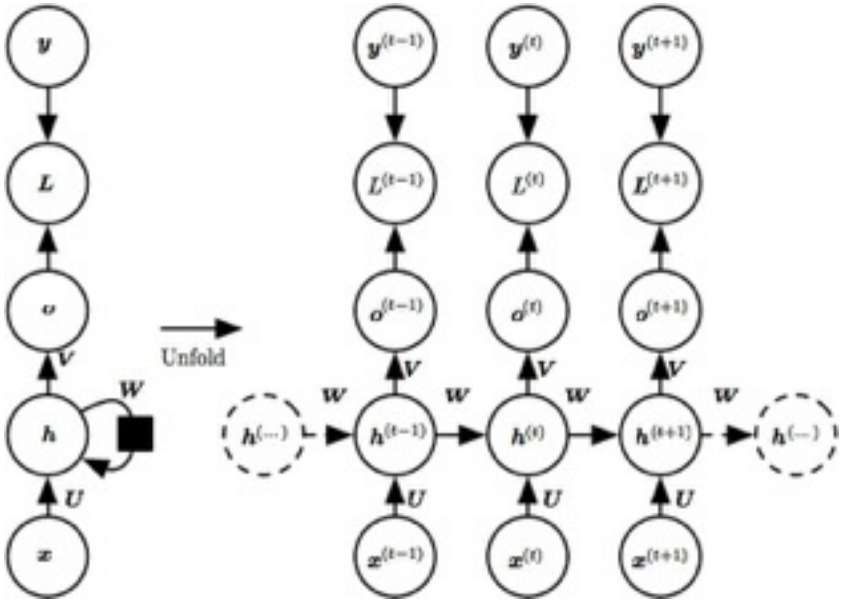
3. Recurrent connections between hidden units to read entire input sequence and produce a single output
   - Can summarize sequence to produce a fixed size representation for further processing



Design 1: RNN with recurrence between hidden units

- Maps input sequence $\boldsymbol{x}$ to output $\boldsymbol{o}$ values
  - Loss L measures how far each $\boldsymbol{o}$ is from the corresponding target $\boldsymbol{y}$
    - With softmax outputs we assume $\boldsymbol{o}$ is the unnormalized log probabilities
    - Loss L internally computes $\boldsymbol{y}=\text{softmax}(\boldsymbol{o})$ and compares to target $\boldsymbol{y}$
- Update equation

$$\boldsymbol{a}^{(t)}=\boldsymbol{b}+W\boldsymbol{h}^{(t-1)}+U\boldsymbol{x}^{(t)}$$



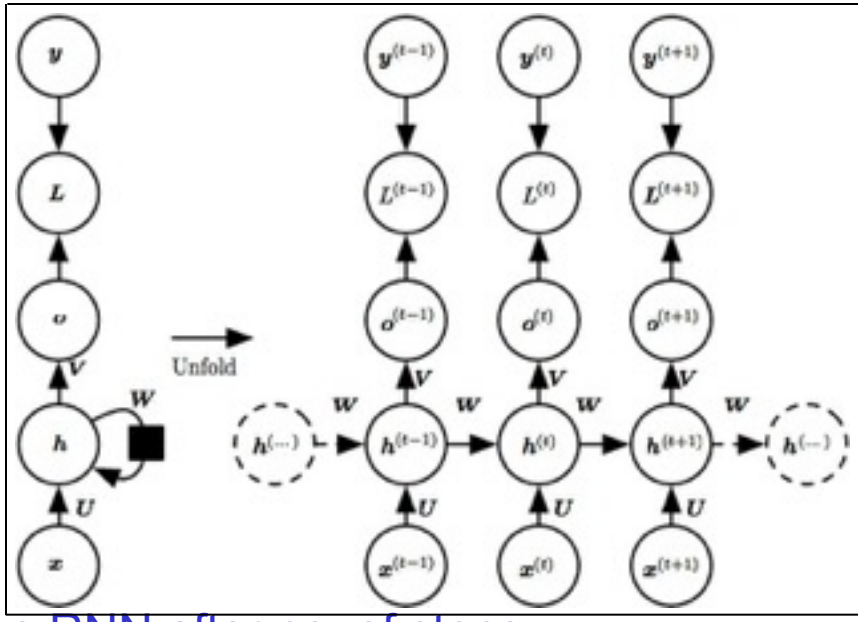
RNN with hidden unit recurrence is a Universal TM

- RNN with hidden unit connections together with

$$\boldsymbol{a}^{(t)}=\boldsymbol{b}+W\boldsymbol{h}^{(t-1)}+U\boldsymbol{x}^{(t)}$$

is Universal, i.e.,

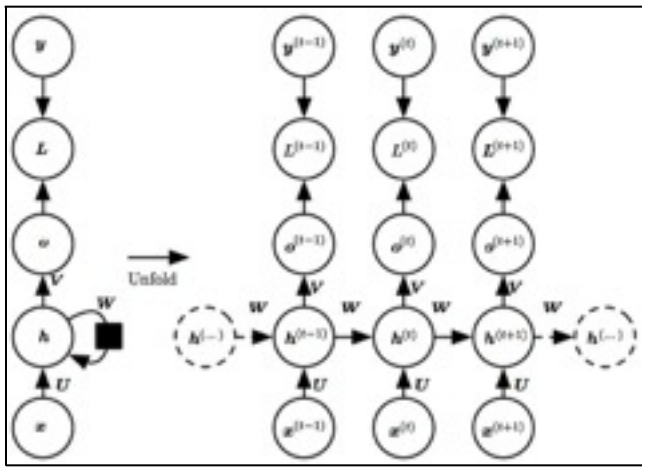
  - Any function computable by a TM is computable by such an RNN of finite size



- Output can be read from the RNN after no. of steps asymptotically linear in no. of steps used by the TM and asymptotically linear in length of input

Forward prop for RNN with hidden unit recurrence

- This design does not specify
  1. activation functions for hidden units,
  2. form of output and
  3. loss function
- Assume for this graph:
  1. Hyperbolic tangent activation function
  2. Output is discrete



  - E.g., RNN predicts words/characters
  - Natural way to represent discrete vars:
    - Output $\boldsymbol{o}$ gives unnormalized log probabilities of each possible value $\hat{\boldsymbol{y}}$
    - Can apply softmax as a postprocessing step to obtain vector of normalized probabilities over the output
- Forward prop begins with specification of initial state $\boldsymbol{h}^{(0)}$

Forward Prop Equations for RNN with hidden recurrence

- Begins with initial specification of $\boldsymbol{h}^{(0)}$
- Then for each time step from $t=1$ to $t=\tau$ we apply the following update equations

$$\boldsymbol{o}^{(t)}=\boldsymbol{c}+V\boldsymbol{h}^{(t)}$$
$$\boldsymbol{h}^{(t)}=\tanh(\boldsymbol{a}^{(t)})$$
$$\boldsymbol{a}^{(t)}=\boldsymbol{b}+W\boldsymbol{h}^{(t-1)}+U\boldsymbol{x}^{(t)}$$

$$\hat{\boldsymbol{y}}^{(t)}=\text{softmax}(\boldsymbol{o}^{(t)})$$

- Where the parameters are:
  - bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$
  - weight matrices U (input-to-hidden), V (hidden-to-output) and W (hidden-to-hidden) connections

Loss function for a given sequence

- This is an example of an RNN that maps an input sequence to an output sequence of the same length
- Total loss for a given sequence for a given sequence of $\boldsymbol{x}$ values with a sequence of $\boldsymbol{y}$ values is the sum of the losses over the time steps
- If $L^{(t)}$ is the negative log-likelihood of $\boldsymbol{y}^{(t)}$ given $\boldsymbol{x}^{(1)},..\boldsymbol{x}^{(t)}$ then

$$L\left(\left\{\boldsymbol{x}^{(1)},..\boldsymbol{x}^{(t)}\right\}\bigcup\left\{\boldsymbol{y}^{(1)},..\boldsymbol{y}^{(t)}\right\}\right)=\sum_t L^{(t)}$$
$$=-\sum_t \log p_{\text{model}}\left(\boldsymbol{y}^{(t)}\,|\left\{\boldsymbol{x}^{(1)},..\boldsymbol{x}^{(t)}\right\}\right)$$

- where p_{model} is given by reading the entry for $\boldsymbol{y}^{(t)}$ from the model's output vector $\boldsymbol{y}^{(t)}$

Computing Gradient of Loss Function

- Computing gradient of this loss function wrt parameters is expensive
  - It involves performing a forward propagation pass moving left to right through the unrolled graph followed by a backward propagation moving right to left through the graph
- Run time is $O(\tau)$ and cannot be reduce by parallelization
  - Because forward propagation graph is inherently sequential
    - Each time step computable only after previous step
  - States computed during forward pass must be stored until reused in the backward pass
    - So memory cost is also $O(\tau)$
- Backpropagation applied to the unrolled graph with $O(\tau)$ cost is called *Backward Propagation through time* (BPTT)

Backward Propagation through Time (BPTT)

- RNN with hidden unit recurrence is very powerful but also expensive to train
- Is there an alternative?

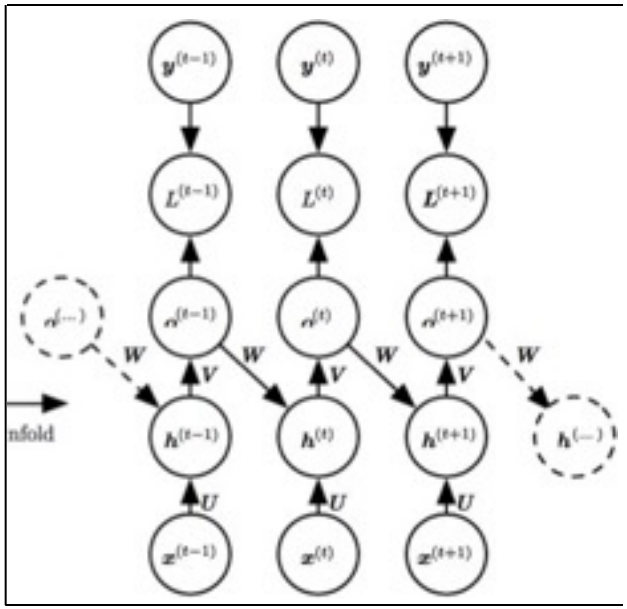
Topics in Recurrent Neural Networks

- Overview
  - Design patterns for RNNs
  - RNN with recurrence between hidden units
  - Forward propagation equations
  - Loss function for a sequence
  - RNN with recurrence from output units to hidden units

1. Teacher forcing and Networks with Output Recurrences
2. Computing the gradient in a RNN
3. Recurrent Networks as Graphical Models
4. Modeling Sequences Conditioned on Context with RNNs

Models with recurrence from output to hidden

- Less powerful than with hidden-to-hidden recurrent connections
  - It cannot simulate a universal TM
  - It requires that the output capture all information of past to predict future
- Advantage
  - In comparing loss function to output all time steps are decoupled
    - Each step can be trained in isolation
  - Training can be parallelized
    - Gradient for each step t computed in isolation
    - No need to compute output for the previous step first, because training set provides ideal value of output



- Can be trained with Teacher Forcing
  - Described next

Teacher Forcing

- Used for training models with recurrent connections from their outputs
- Teacher forcing during training means
  - Instead of summing activations from incoming units (possibly erroneous)
  - Each unit sums correct teacher activations as input for the next iteration

Teacher Forcing Procedure

- Teacher forcing is a procedure for training RNNs with output-to-hidden recurrence
- It emerges from the maximum likelihood criterion
- During training time model receives ground truth output $\boldsymbol{y}^{(t)}$ as input at time $t+1$
  - We can see this by examining a sequence with two time steps (next)

Maximum Likelihood in Teacher Forcing

- Consider a sequence with two time steps
- The conditional maximum likelihood criterion is

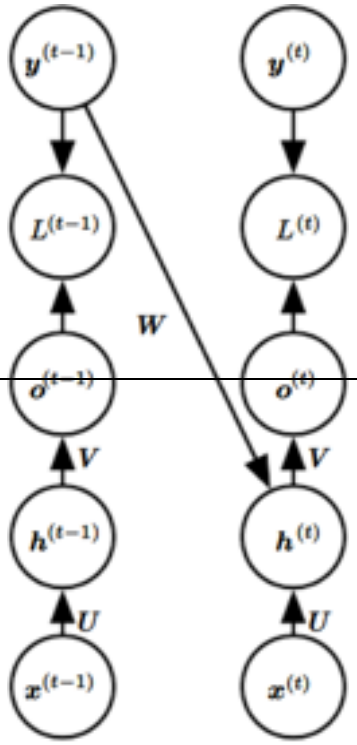
$$\log\left(p(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) = \log p(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}) + \log p(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)})$$

  - At time $t=2$, model is trained to maximize conditional probability of $\boldsymbol{y}^{(2)}$ given both the $\boldsymbol{x}$ sequence so far and the previous $\boldsymbol{y}$ value from the training set.
    - We are using $\boldsymbol{y}^{(1)}$ as teacher forcing, rather than only $\boldsymbol{x}^{(i)}$.
  - Maximum likelihood specifies that during training, rather than feeding the model's own output back to itself, target values should specify what the correct output should be

Illustration of Teacher Forcing

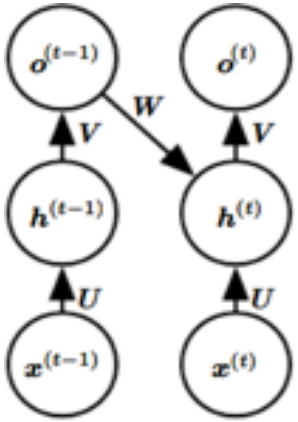
- Teacher Forcing is a training technique applicable to RNNs that have connections from output to hidden states at next time step

Train time:
We feed the correct output $\boldsymbol{y}^{(t)}$ (from teacher) drawn from the training set as input to $\boldsymbol{h}^{(t+1)}$



Train time

Test time:
True output is not known.
We approximate the correct output $\boldsymbol{y}^{(t)}$ with the model's output $\boldsymbol{o}^{(t)}$ and feed the output back to the model



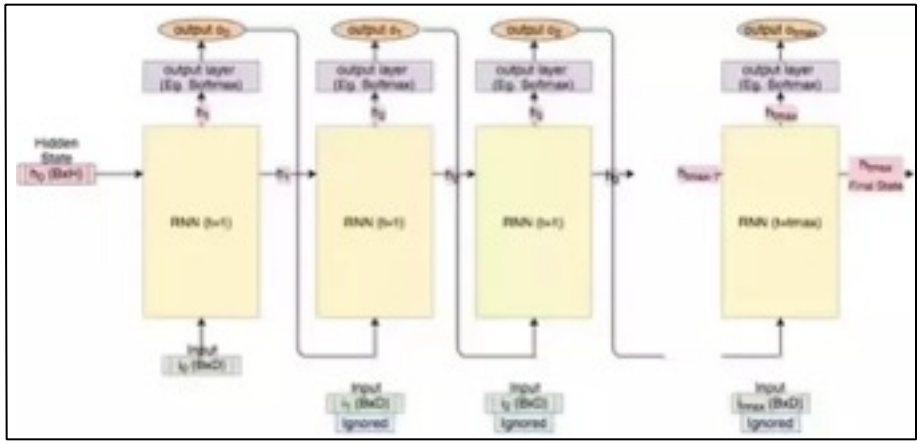
Test time

RNNs without and with teacher forcing

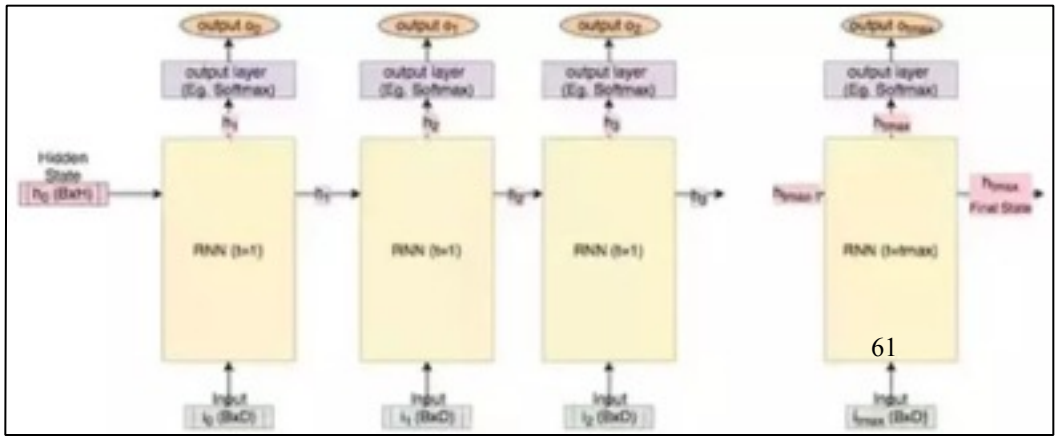
Without Teacher forcing:

Output of previous time step
acts as input during RNN training

RNN learns to generate
next output in a sequence based
upon the previous value



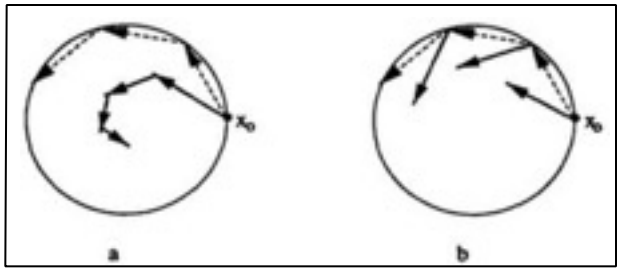
With Teacher forcing:



Source: https://www.quora.com/What-is-the-teacher-forcing-in-RNN

Visualizing Teacher Forcing

- Imagine that the network is learning to follow a trajectory
- It goes astray (because the weights are wrong) but teacher forcing puts the net back on its trajectory
  - By setting the state of all the units to that of teacher's.



(a) Without teacher forcing, trajectory runs astray (solid lines) while the correct trajectory are the dotted lines

(b) With teacher forcing trajectory corrected at each step

Teacher forcing for hidden-to-hidden

- We originally motivated teacher forcing as allowing us to avoid backpropagation through time in models that lack hidden-to-hidden connections
- Teacher forcing can still be applied to models that have hidden-to-hidden connections
  - As long as they have connections from the output at one time step to values computed at the next time step
  - As soon aas the hidden units become functions of earlier time steps, BPTT algorithm is necessary

Training with both Teacher Forcing and BPTT

- Some models may be trained with both Teacher forcing and Backward Propagation through time (BPTT)
  - When there are both hidden-to-hidden recurrences as well as output-to-hidden recurrences

Disadvantage of Teacher Forcing

- If network is to be used in an *open-loop* mode with network outputs (or samples from the output distribution) fed back as inputs
  - In this case the kind of inputs that it will see during training time could be quite different from that it will see at test time
- Solutions to mitigate this problem:
  1. Train with both teacher-forced inputs and free running inputs
     - E.g., predicting the correct target a no of steps in the future through the unfolded recurrent output-to-input paths
     - Thus network can learn to take into account input conditions not seen during training
  2. Mitigate the gap between inputs seen at training time and test time by *generating* values as input
     - This approach exploits a *curriculum learning* strategy to gradually use more of the generated values as input

Professor Forcing

- Teacher Forcing trains RNNs by supplying observed sequence values as inputs during training and using the network's own one-step-ahead predictions to do multi-step sampling
- Professor Forcing algorithm
  - Uses adversarial domain adaptation to encourage the dynamics of the RNN to be the same when training the network and when sampling from the network over multiple time steps.
  - Applied to
    - language modeling
    - vocal synthesis on raw waveforms
    - handwriting generation
    - image generation
- Professor Forcing is an adversarial method for learning generative models
  - It is closely related to Generative Adversarial Networks

Computing the Gradient in an RNN

Topics in Recurrent Neural Networks

1. Overview
2. Teacher forcing and Networks with Output Recurrence
3. Computing the gradient in a RNN
4. RNNs as Directed Graphical Models
5. Modeling Sequences Conditioned on Context with RNNs

General Backpropagation to compute gradient

To compute gradients T of variable z wrt variables in computational graph G

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$.
Initialize `grad_table`, a data structure associating tensors to their gradients
`grad_table`$[z] \leftarrow 1$
for $\mathbf{V}$ in $\mathbb{T}$ **do**
  `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
end for
Return `grad_table` restricted to $\mathbb{T}$

Build-grad function of Generalized Backprop

Algorithm 6.6 The inner loop subroutine `build_grad`($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, `grad_table`) of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: `grad_table`, a data structure mapping nodes to their gradients

```
if V is in grad_table then
  Return grad_table[V]
end if
i ← 1
for C in get_consumers(V, G') do
  op ← get_operation(C)
  D ← build_grad(C, G, G', grad_table)
  G^(i) ← op.bprop(get_inputs(C, G'), V, D)
  i ← i + 1
end for
G ← Σ_i G^(i)
grad_table[V] = G
Insert G and the operations creating it into G
Return G
```

Ob.bprop(inputs,X,$\mathcal{G}$)
returns $\sum_i (\nabla_X \text{op.f(inputs)}_i) G_i$
Which is just an implementation of chain rule

If $\boldsymbol{y}=g(x)$ and $z=f(\boldsymbol{y})$ from chain rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Equivalently, in vector notation

$$\nabla_x z = \left(\frac{\partial \boldsymbol{y}}{\partial x}\right)^T \nabla_y z$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the n x n Jacobian matrix of g

Example for how BPTT algorithm behaves

- BPTT: Back Propagation Through Time
- We provide an example of how to compute gradients by BPTT for the RNN equations

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

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

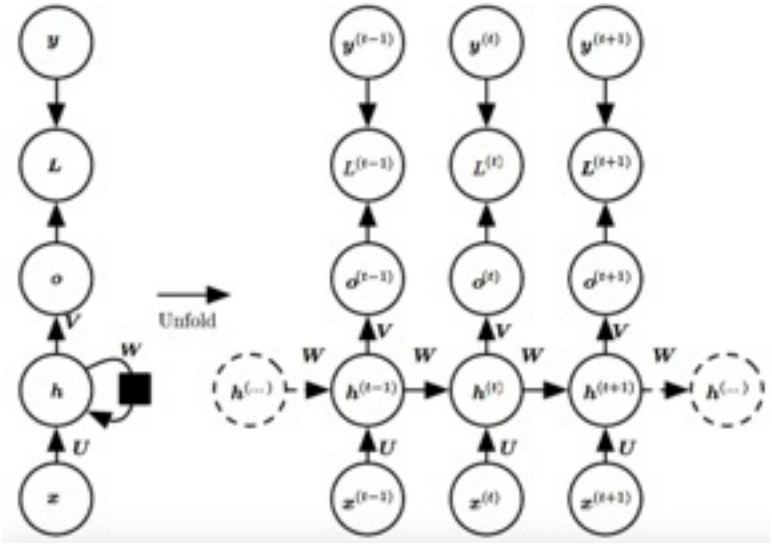
Intuition of how BPTT algorithm behaves

- Ex: Gradients by BPTT for RNN eqns given above

Nodes of our computational graph include the parameters

U, V, W, b, c as well as the sequence of nodes indexed by t for $x^{(t)}$, $h^{(t)}$, $o^{(t)}$ and $L^{(t)}$

For each node **N** we need to compute the gradient $\nabla_N L$ recursively

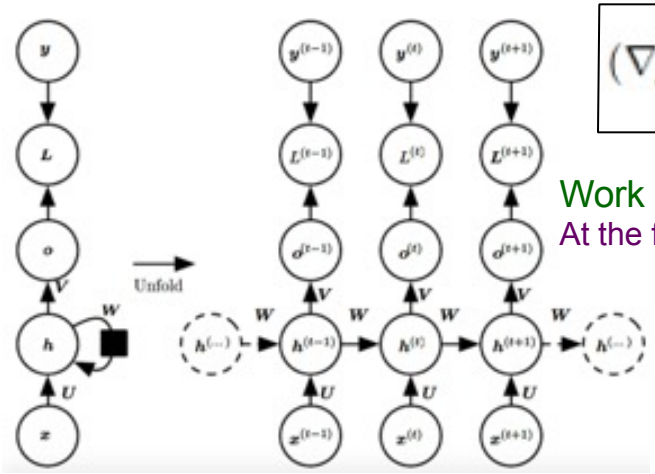


Computing BPTT gradients recursively

- Start recursion with nodes immediately preceding final loss

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

- We assume that the outputs $\boldsymbol{o}^{(t)}$ are used as arguments to softmax to obtain vector $\hat{\boldsymbol{y}}$ of probabilities over the output. Also that loss is the negative log-likelihood of the true target $\boldsymbol{y}^{(t)}$ given the input so far.
- The gradient on the outputs at time step t is



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

Work backwards starting from the end of the sequence.
At the final time step τ, $\boldsymbol{h}^{(\tau)}$ has only $\boldsymbol{o}^{(\tau)}$ as a descendent. So

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

Gradients over hidden units

- Iterate backwards in time iterating to backpropagate gradients through time from $t = \tau\text{-}1$ down to $t = 1$

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} (\nabla_{\boldsymbol{o}^{(t)}} L)$$

$$= \boldsymbol{W}^{\top} (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^{\top} (\nabla_{\boldsymbol{o}^{(t)}} L)$$

  - where $\operatorname{diag}\left(1 - (\boldsymbol{h}^{(t+1)})^2\right)$ is matrix with elements within parentheses. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t+1$

Gradients over parameters

- Once gradients on internal nodes of the computational graph are obtained, we can obtain gradients on the parameters
  - Because parameters are shared across time steps some care is needed

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

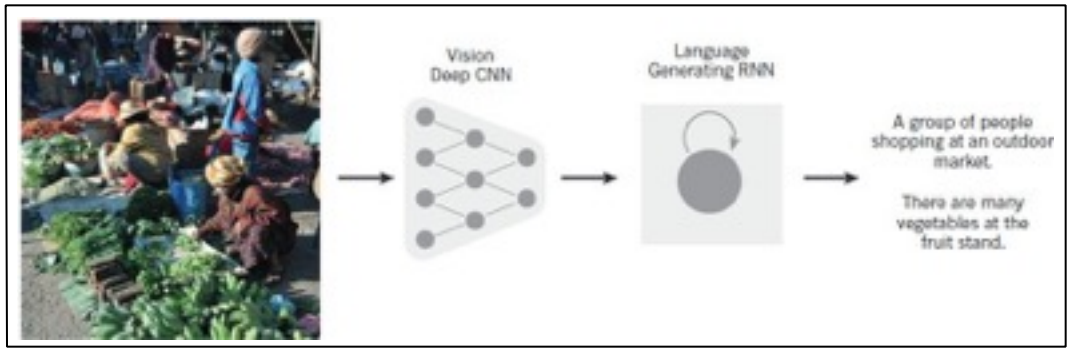
RNNs as Directed Graphical Models

Topics in RNNs as Directed Graphical Models

- Motivation
- What are directed graphical models?
- RNNs as Directed Graphical Models
  - Conditioning Predictions on Past Values
  - Fully connected graphical model for a sequence
  - Hidden units as random variables
  - Efficiency of RNN parameterization
- How to draw samples from the model

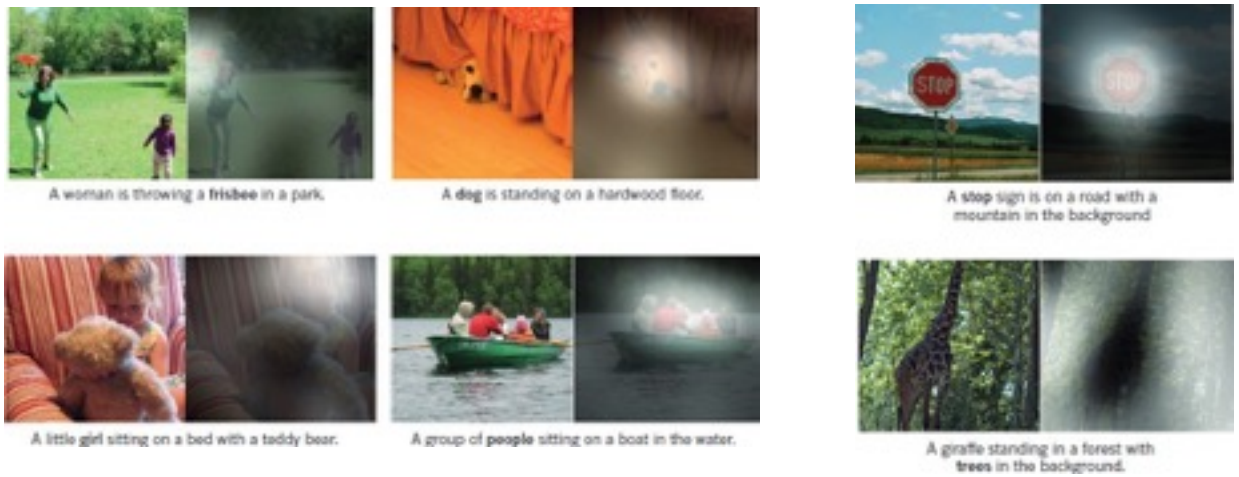
Motivation: From images to text

- Combining ConvNets and Recurrent Net Modules
- Caption generated by a recurrent neural network (RNN) taking as input:
    1. Representation generated by a deep CNN
    2. RNN trained to translate high-level representations of images into captions



Better translation of images to captions

- Different focus (lighter patches given more attention)



- As it generates each word (bold), it exploits it to achieve better translation of images to captions

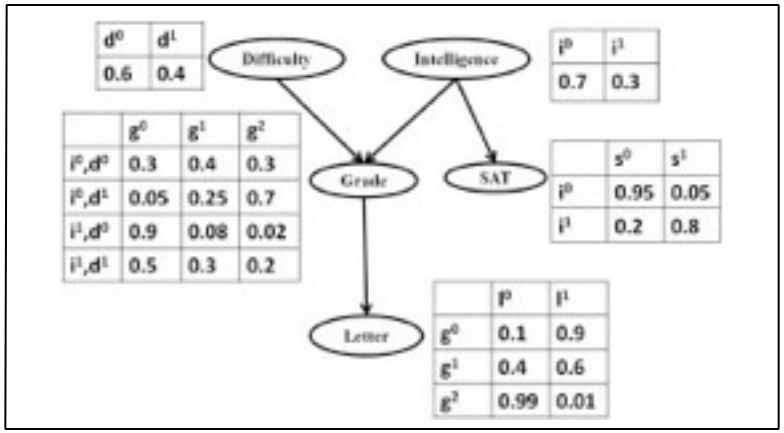
What is a Directed graphical model?

- Specifies a joint distribution $p(\boldsymbol{x}) = p(x_1,..x_n)$ in terms of conditional distributions (called a Bayesian Network)

$$p(\boldsymbol{x}) = \prod_{i=1}^{N} p(x_i \mid pa(x_i))$$

  - where $p(x_i \mid pa(x_i))$ are conditional distributions
    - pa represents parents in a directed graph
  - Computational efficiency is obtained by omitting edges that do not correspond to strong interactions
  - Model is generative:
    - We can easily sample from a BN (ancestral sampling)
- We will see here that RNNs have an interpretation as an efficient directed graphical model

Example of a Directed Graphical Model (Bayesian Net)



Instead of 2x2x3x2x2=48 parameters we need
18 parameters using the directed graphical model
(which makes certain condition independence assumptions)

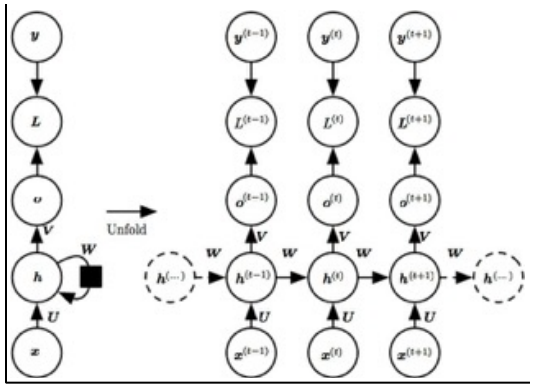
$$P(D,I,G,S,L) = P(D)P(I)P(G \mid D,I)P(S \mid I)P(L \mid G)$$

$$P(i^1,d^0,g^2,s^1,l^0) = P(i^1)P(d^0)P(g^2 \mid i^1,d^0)P(s^1 \mid i^1)P(l^0 \mid g^2)$$
$$= 0.3 \cdot 0.6 \cdot 0.08 \cdot 0.8 \cdot 0.4 = 0.004608$$

CPDs of model depend on RNN Design pattern

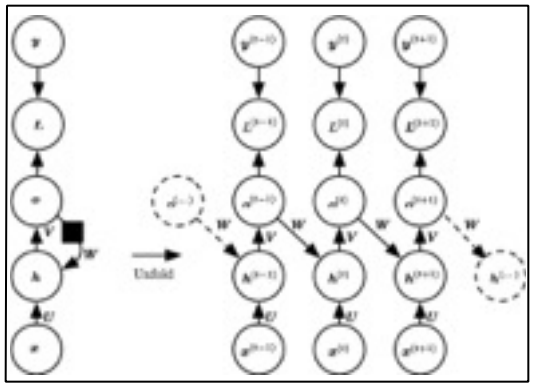
1. Recurrent connections between hidden units

1



2. Recurrent connections only from output at one time step to hidden units at next time step

2



Loss function for RNNs

- In feed-forward networks our goal is to minimize

$$-E_{x\sim\hat{p}_{\text{data}}}\left[\log p_{\text{model}}(\boldsymbol{x})\right]$$

  - Which is the cross-entropy between distribution of training set (data) and probability distribution defined by model
    - *Definition of cross entropy* between distributions p and q is $\mathrm{H}(p,q)=\mathrm{E}_p[-\log\ q]=\mathrm{H}(p)+\mathrm{D}_{\mathrm{KL}}(p||q)$
    - For discrete distributions $\mathrm{H}(p,q)=-\sum_{\boldsymbol{x}} p(\boldsymbol{x})\log\ q(\boldsymbol{x})$
- As with a feedforward network, we wish to interpret the output of the RNN as a probability distribution
  - With RNNs the losses $L^{(\mathrm{t})}$ are cross entropies between training targets $\boldsymbol{y}^{(t)}$ and outputs $\boldsymbol{o}^{(t)}$
    - Mean squared error is the cross-entropy loss associated with an output distribution that is a unit Gaussian

Types of CPDs in an RNN

1. With a predictive log-likelihood training objective:

$$L\left(\{\boldsymbol{x}^{(1)},..\boldsymbol{x}^{(t)}\},\{\boldsymbol{y}^{(1)},..\boldsymbol{y}^{(t)}\}\right)=\sum_t L^{(t)}=-\sum_t \log p_{model}\left(\boldsymbol{y}^{(t)}\,|\,\{\boldsymbol{x}^{(1)},..\boldsymbol{x}^{(t)}\}\right)$$

we train the RNN to estimate the conditional distribution of next sequence element $\boldsymbol{y}^{(t)}$ given past inputs. This means that we maximize log-likelihood

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

2. Alternatively if the model includes connections from output at one time step to the next

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

Conditioning predictions on past y values

- Decomposing joint probability over 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.

1. Do not feed past y values as inputs for next step prediction,

   directed model contains no edges from any $y^{(i)}$ in the past to current $y^{(t)}$. In this case outputs $y^{(i)}$ are conditionally independent given the x values

2. Do feed the actual y values (not their prediction, but the actual observed or generated values) back into the network

   the directed PGM contains edges from all $y^{(i)}$ values in the past to the current $y^{(t)}$ value.

A simple example

- RNN models a sequence of scalar random variables $Y = \{\boldsymbol{y}^{(1)}, .. \boldsymbol{y}^{(\tau)}\}$ with no additional inputs $\boldsymbol{x}$.
  - Input at time step t is simply the output at time step t-1
  - RNN defines a directed graphical model over $\boldsymbol{y}$ variables
- Parameterize joint distribution of observations using the chain rule for conditional probabilities:

$$P(Y) = P(\boldsymbol{y}^{(1)}, .. \boldsymbol{y}^{(\tau)}) = \prod_{t=1}^{\tau} P(\boldsymbol{y}^{(t)} \mid \boldsymbol{y}^{(t-1)}, \boldsymbol{y}^{(t-2)}, .., \boldsymbol{y}^{(1)})$$

where the right hand side of the bar | is empty for t=1

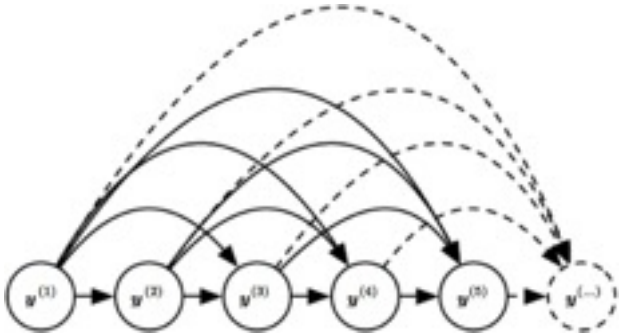
- Hence the negative log-likelihood of a set of values $\{\boldsymbol{y}^{(1)}, .. \boldsymbol{y}^{(\tau)}\}$ according to such a model is

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

$$\text{where } L^{(t)} = -\sum_t \log P\left(\mathrm{y}^{(t)} = \boldsymbol{y}^{(t)} \mid y^{(t-1)}, y^{(t-2)}, .., y^{(1)}\right)$$

Fully connected graphical model for a sequence

- Every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ for $t > 1$
  - Parameterizing this way is inefficient.



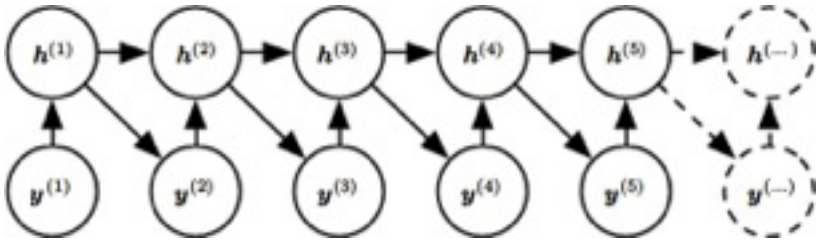
- RNNs obtain the same full connectivity but efficient parameterization as shown next

Introducing the state variable in the PGM of an RNN

Introduce the state variable in the PGM of RNN

even though it is a deterministic function of its inputs, helps see we get efficient parameterization

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

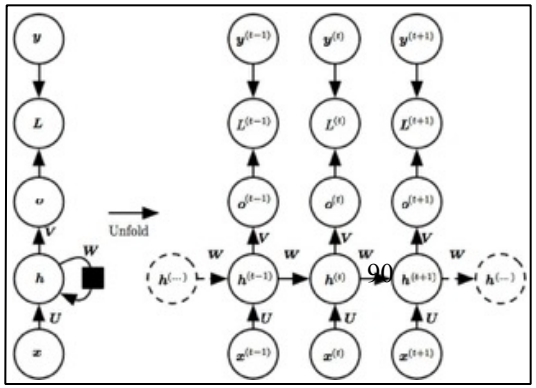


Every stage in the sequence for $\boldsymbol{h}^{(t)}$ and $\boldsymbol{y}^{(t)}$ involves the same structure and can share the parameters with other stages

- If y can take on k different values, the tabular representation would have $O(k^{\tau})$ parameters
  - By comparison because of parameter sharing the no of parameters in the RNN is $O(1)$ as a function of sequence length
- Even with efficient parameterization of PGM, some operations are computationally challenging
  - Difficult to predict missing values in the middle of the sequence
- Price RNNs pay for reduced no of parameters is that oprimizing parameters may be difficult

Stationarity in RNNs

- Parameter sharing in RNNs relies on assuming same parameters can be used in different time steps
- Equivalently CPD over variables at time $t+1$ given the variables at time t is stationary
- In principle it is possible to use t as an extra input at each time step
  - Learner discovers time dependence



How to draw samples from the model

- Simply sample from the conditional distribution at each time step
- One additional complication
  - RNN must have some mechanism for determining the length of the sequence

Methods for predicting end of sequence

- This can be achieved in various ways
  1. Adding a special symbol for end of sequence
     - When the symbol is generated, the sampling process stops
  2. Introduce an extra Bernoulli output
     - It represents the decision to either continue generation or halt generation at each time step
       - More general than previous method as it can be added to any RNN rather than only RNNs that output a sequence of symbols, e.g., RNN that emits sequence of real numbers
  3. Add an extra output to the model to predict τ
     - Model can sample value of τ and then sample τ steps worth of data

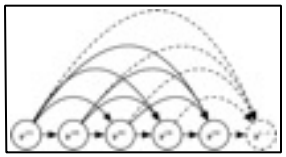
Modeling Sequences Conditioned on Context with RNNs

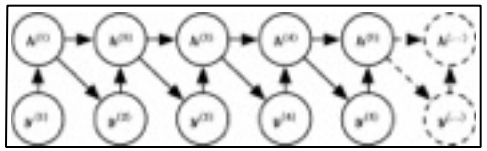
Graphical models of RNNs *without/with* inputs

1. Directed graphical models of RNNs *without* inputs

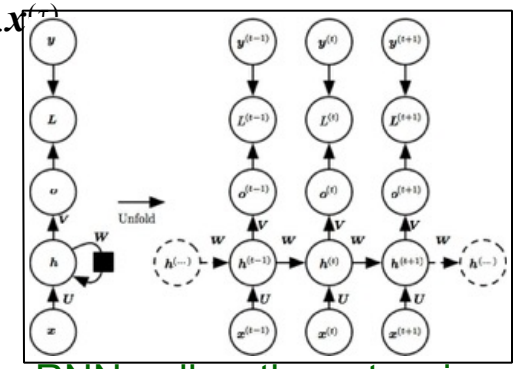
- having a set of random variables $\boldsymbol{y}^{(t)}$:

Fully connected graphical model Efficient parameterization based on $\boldsymbol{h}^{(t)}=f(\boldsymbol{h}^{(t-1)},\boldsymbol{x}^{(t)};\boldsymbol{\theta})$





2. RNNs do include a sequence of inputs $\boldsymbol{x}^{(1)}$, $\boldsymbol{x}^{(2)}$,..$\boldsymbol{x}^{(\tau)}$



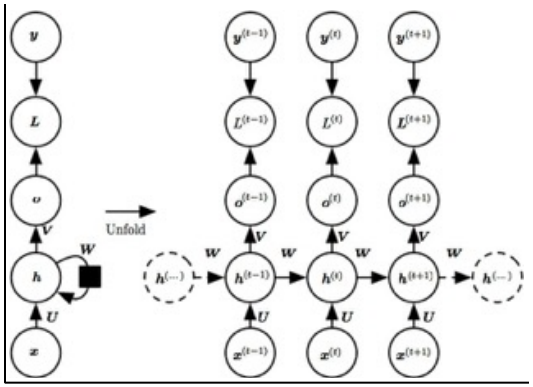
$$\boldsymbol{o}^{(t)}=\boldsymbol{c}+V\boldsymbol{h}^{(t)}$$
$$\boldsymbol{h}^{(t)}=\tanh(\boldsymbol{a}^{(t)})$$
$$\boldsymbol{a}^{(t)}=\boldsymbol{b}+W\boldsymbol{h}^{(t-1)}+U\boldsymbol{x}^{(t)}$$

- RNNs allow the extension of the graphical model view to represent not only the joint distribution view over y variables but also a conditional distribution over y given x

CPDs of model depend on RNN Design pattern

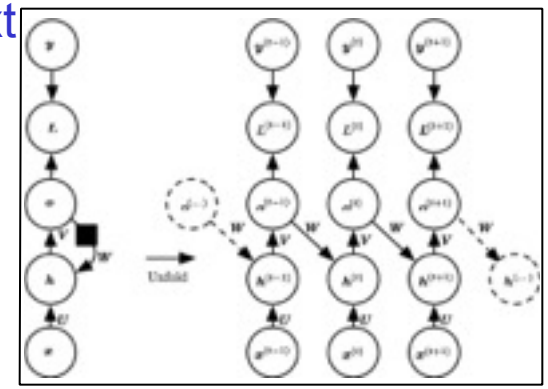
1. Recurrent connections between hidden units

1



2. Recurrent connections only from output at one time step to hidden units at next time step

2



Extending RNNs to represent conditional $P(\boldsymbol{y}|\boldsymbol{x})$

- A model representing a variable $P(\boldsymbol{y};\ \boldsymbol{\theta})$ can be reinterpreted as a model representing a conditional distribution $P(\boldsymbol{y}|\boldsymbol{\omega})$ with $\boldsymbol{\omega}=\boldsymbol{\theta}$
- We can extend such a model to represent a distribution $P(\boldsymbol{y}|\boldsymbol{x})$ by using the same $P(\boldsymbol{y}|\boldsymbol{\omega})$ as before but making $\boldsymbol{\omega}$ a function of $\boldsymbol{x}$
- In the case of an RNN this can be achieved in several ways
  - Most common choices are described next

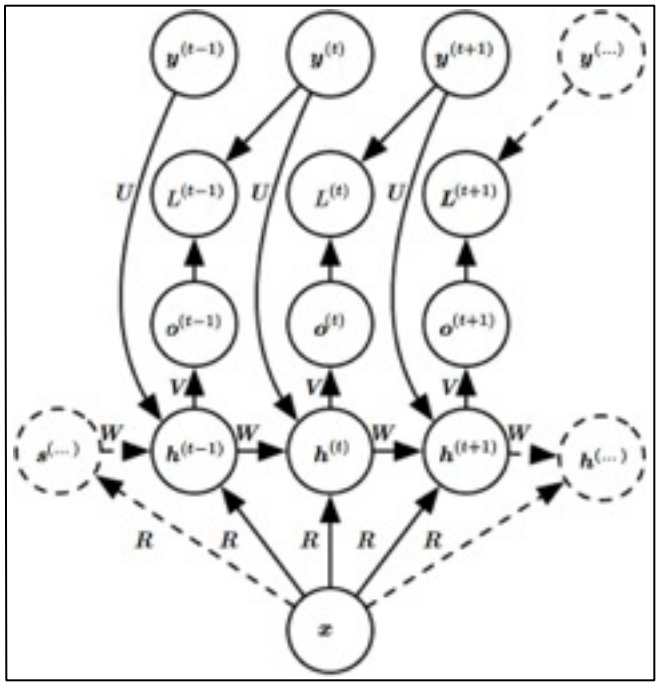
Taking a single vector $\boldsymbol{x}$ as an extra input

- Instead of taking a sequence $\boldsymbol{x}^{(t)},\ t = 1,..,\tau$ as input we can take a single vector $\boldsymbol{x}$ as input
- When $\boldsymbol{x}$ is a fixed-size vector we can simply make it an extra input of the RNN that generates the $\boldsymbol{y}$ sequence
- Common ways of providing an extra input to RNN are
  1. An extra input at each time step, or
  2. As the initial state $\boldsymbol{h}^{(0)}$, or
  3. Both
- The first and common approach is illustrated next

Mapping vector $\boldsymbol{x}$ into distribution over sequences $\mathbf{Y}$

An extra input at each time step

Interaction between input $\boldsymbol{x}$ and hidden unit vector $\boldsymbol{h}^{(t)}$
is parameterized by a newly introduced weight matrix R
that was absent from the model with only y values



Appropriate for tasks such as image captioning
where a single image $\boldsymbol{x}$ is input which produces a sequence of words describing the image

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

RNN to receive a sequence of vectors $\boldsymbol{x}^{(t)}$ as input

- RNN described by $\boldsymbol{a}^{(t)}=\boldsymbol{b}+W\boldsymbol{h}^{(t-1)}+U\boldsymbol{x}^{(t)}$ corresponds to a conditional distribution $P(\boldsymbol{y}^{(1)},\ldots,\boldsymbol{y}^{(\tau)}|\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(\tau)})$
- It makes a conditional independence assumption that this distribution factorizes as

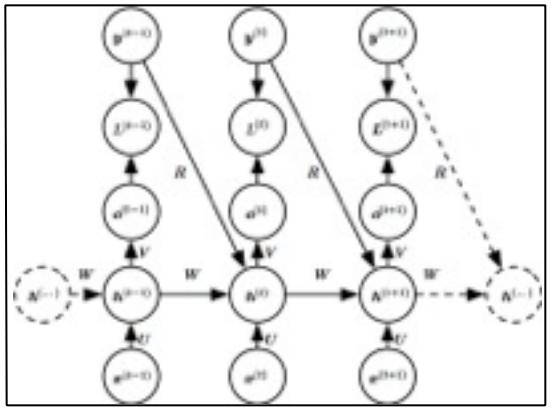
$$\prod_t P(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(1)},..,\boldsymbol{x}^{(t)})$$

- To remove the conditional independence assumption, we can add connections from the output at time t to the hidden unit at time $t+1$ (see next slide)
  - The model can then represent arbitrary probability distributions over the $\boldsymbol{y}$ sequence
- Limitation: both sequences must be of same length
  - Removing this restriction is discussed in Section 10.4

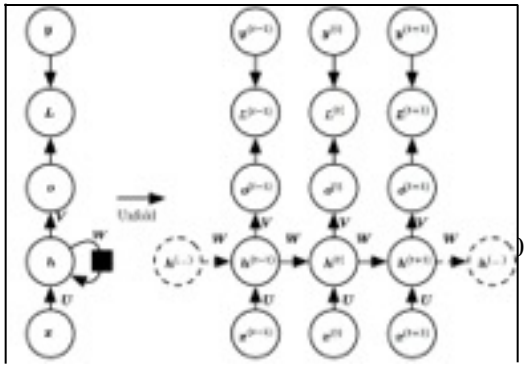
Removing conditional independence assumption

Connections from previous output to current state allow RNN to model arbitrary distribution over sequences of y

Conditional RNN mapping a variable-length sequence of x values into a distribution over sequences of y values of same length



Compare it to model that is only able to represent distributions in which the y values are conditionally independent from each other given x values



BidirectionalRNNs

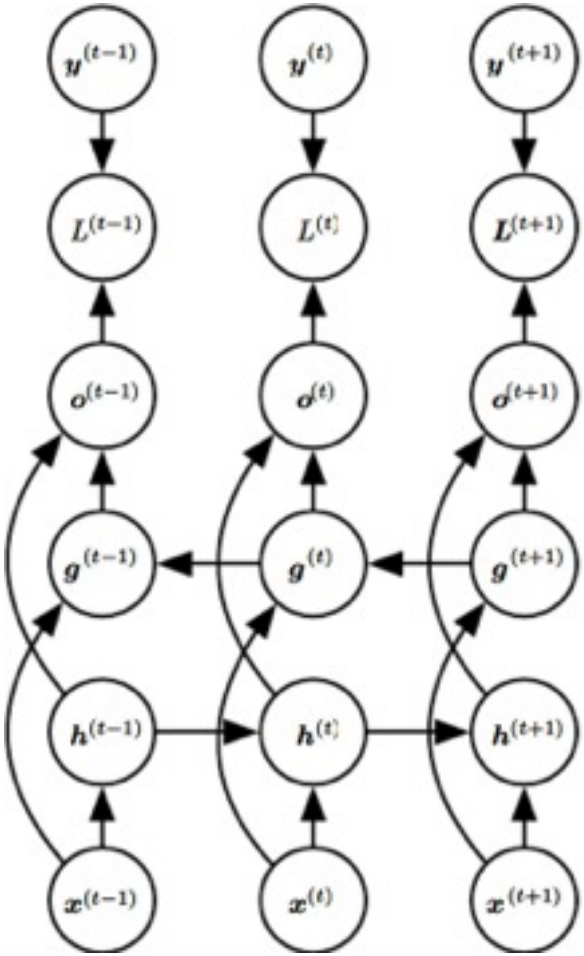
Need for bidirectionality

- In speech recognition, the correct interpretation of the current sound may depend on the next few phonemes because of co-articulation and the next few words because of linguistic dependencies
- Also true of handwriting recognition

A birectional RNN

- Combine an RNN that moves forward through time from the start of the sequence
- Another RNN that moves backward through time beginning from the end of the sequence
- A bidirectional RNN consists of two RNNs which are stacked on the top of each other.
  - The one that processes the input in its original order and the one that processes the reversed input sequence.
  - The output is then computed based on the hidden state of both RNNs.

A typical bidirectional RNN



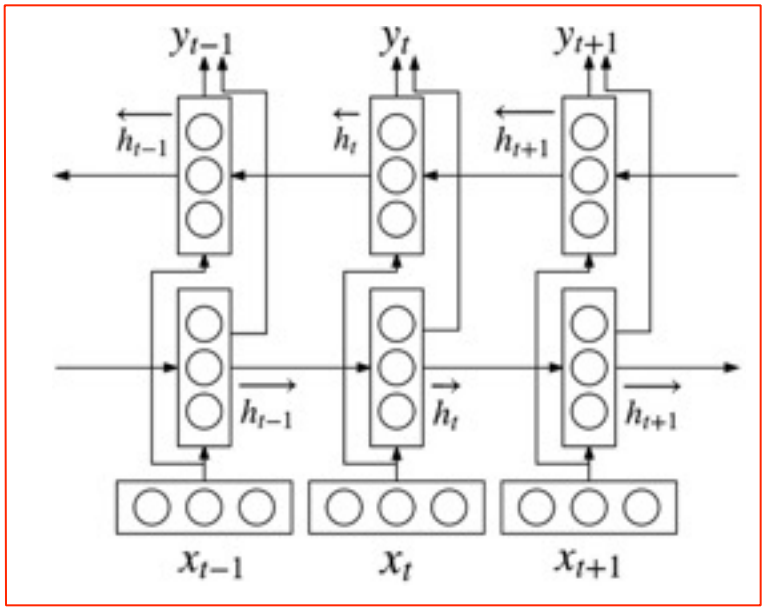
Maps input sequences $\boldsymbol{x}$ to target sequences $\boldsymbol{y}$ with loss $L^{(t)}$ at each step $\boldsymbol{t}$

$\boldsymbol{h}$ recurrence propagates to the right
$\boldsymbol{g}$ recurrence propagates to the left.

This allows output units $\boldsymbol{o}^{(t)}$ to compute a representation that depends both the past and the future

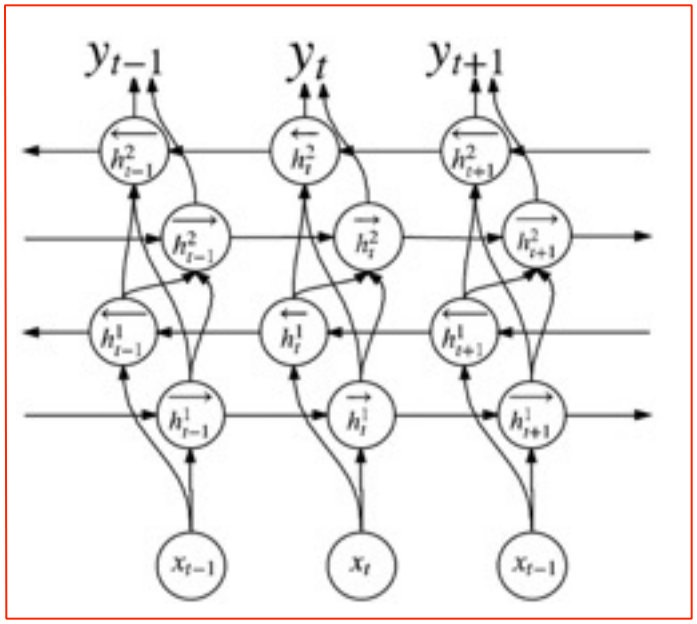
Parameters of a Bidirectional RNN

Bidirectional RNN



Deep Bidirectional RNN

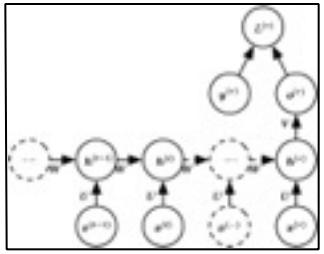
Has multiple layers per time step

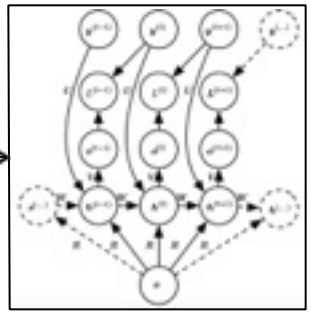


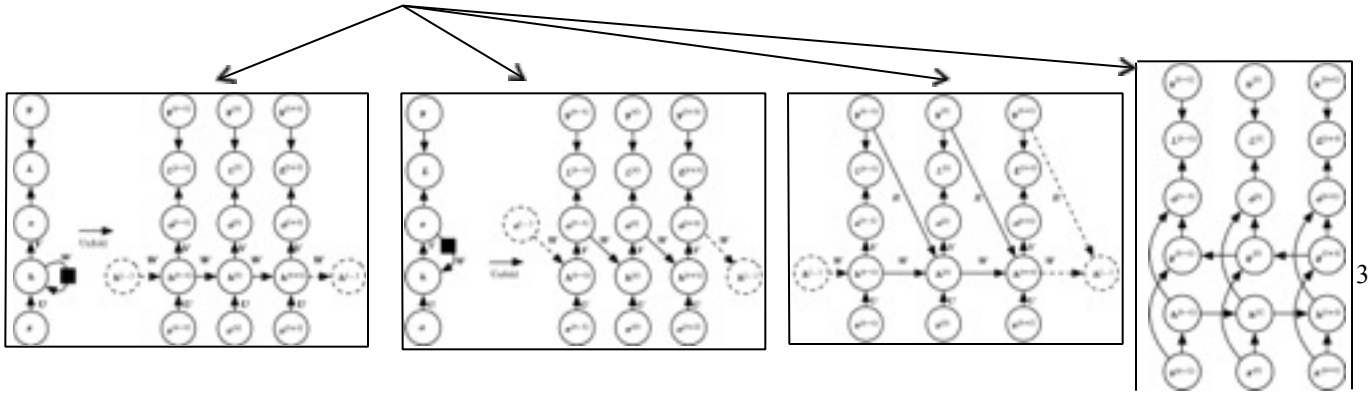
Encoder-Decoder Sequence-to-Sequence Architectures

Previously seen RNNs

1. An RNN to map an input sequence to a fixed-size vector
2. RNN to map a fixed-size vector to a sequence
3. How an RNN can map an input sequence to an output sequence of the same length



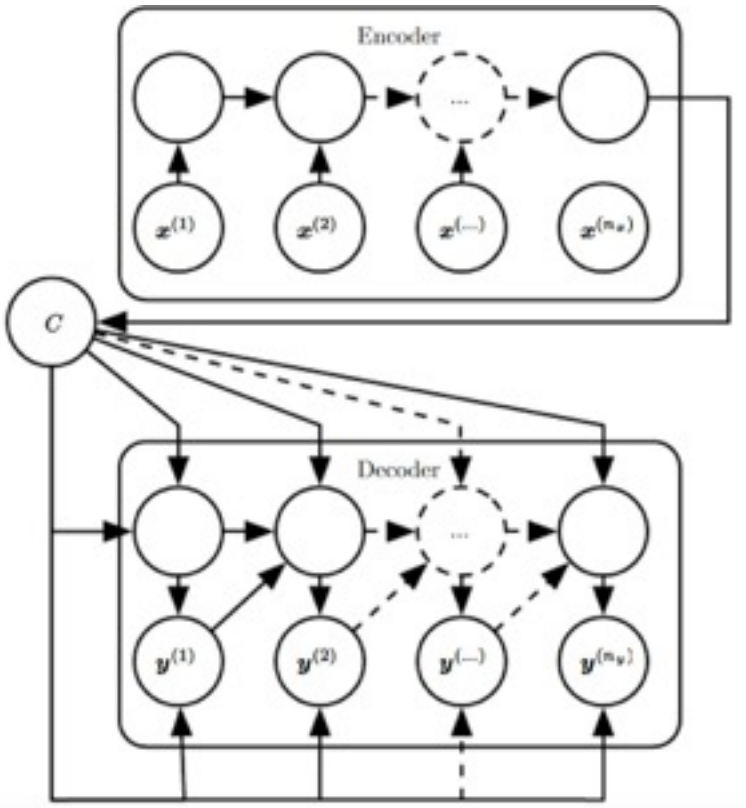




RNN when input/output are not same length

- Here we consider how an RNN can be trained to map an input sequence to an output sequence which is not necessarily the same length
- Comes up in 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 (although their lengths might be related)

An Encoder-Decoder or Sequence-to-Sequence RNN



Learns to generate an output sequence

$(y^{(1)},..,y^{(n_y)})$

given an input sequence

$(x^{(1)},..,x^{(n_x)})$

It consists of an encoder RNN that reads an input sequence and a decoder RNN that generates the output sequence or computes the probability of a given output sequence)

The final hidden state of the encoder RNN Is used to compute a fixed size context C which represents a semantic summary of the input sequence and is given as input to the decoder

Deep Recurrent Networks

Topics

- Recurrent Neural Networks

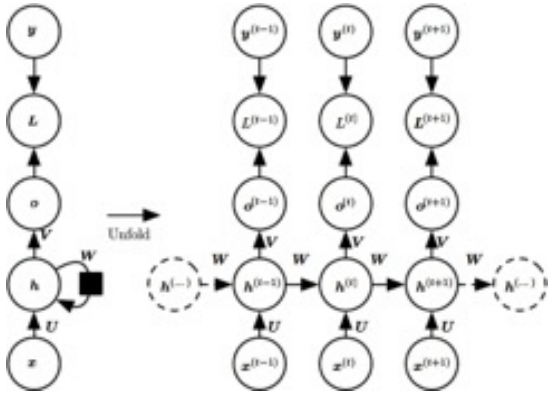
1. Unfolding Computational Graphs
2. Recurrent Neural Networks
3. Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent Networks
6. Recursive Neural Networks
7. The Challenge of Long-Term Dependencies
8. Echo-State Networks
9. Leaky Units and Other Strategies for Multiple Time Scales
10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Computation in RNNs: parameter blocks

- The computation in most recurrent neural networks can be decomposed into three blocks of parameters and associated transformations:
  1. From the input to the hidden state
  2. From the previous hidden state to the next hidden state
  3. From the hidden state to the output

Blocks of parameters as a shallow transformation

- With the RNN architecture shown each of these three blocks is associated with a single weight matrix, i.e.,
  - When the network is unfolded, each of these corresponds to a shallow transformation.
  - By a shallow Transformation we mean a transformation that would be represented a single layer within a deep MLP.
  - Typically this is a transformation represented by a learned affine transformation followed by a fixed nonlinearity



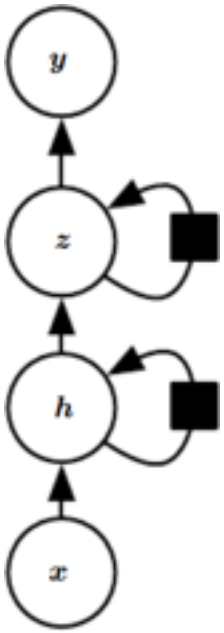
- Would it be advantageous to introduce depth into each of these operations?
  - Experimental evidence strongly suggests so.
    - That we need enough depth in order to perform the required transformations

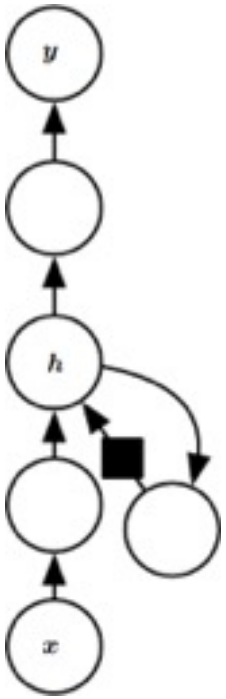
Ways of making an RNN deep

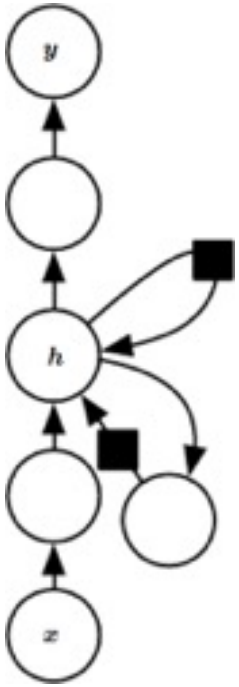
1. Hidden recurrent state can be broken down into groups organized hierarchically

2. Deeper computation can be introduced in the input-hidden, hidden-hidden and hidden-output parts. This may lengthen the shortest path linking different time steps

3. The path-lengthening effect can be mitigated by introducing skip connections.

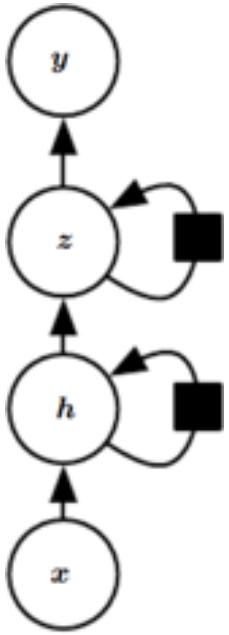






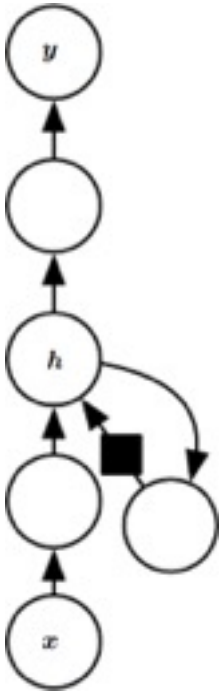
1. Recurrent states broken down into groups

We can think of lower levels of the hierarchy play a role of transforming the raw input into a representation that is more appropriate at the higher levels of the hidden state



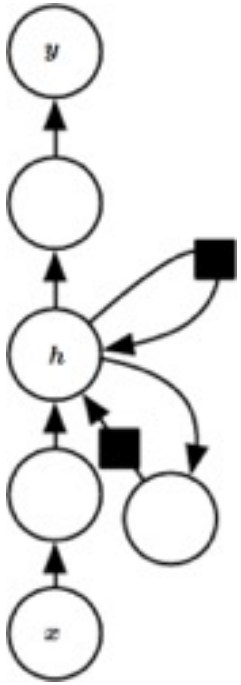
2. Deeper computation in hidden-to-hidden

- Go a step further and propose to have a separate MLP (possibly deep) for each of the three blocks:
  1. From the input to the hidden state
  2. From the previous hidden state to the next hidden state
  3. From the hidden state to the output
- Considerations of representational capacity suggest that to allocate enough capacity in each of these three steps
  - But doing so by adding depth may hurt learning by making optimization difficult
  - In general it is easier to optimize shallower architectures
  - Adding the extra depth makes the shortest time of a variable from time step t to a variable in time step $t+1$ beome longer



3. Introducing skip connections

- For example, if an MLP with a single hidden layer is used for the state-to-state transition, we have doubled the length of the shortest path between variables in any two different time steps compared with the ordinary RNN.
- This can be mitigated by introducing skip connections in the hidden-to-hidden path as illustrated here



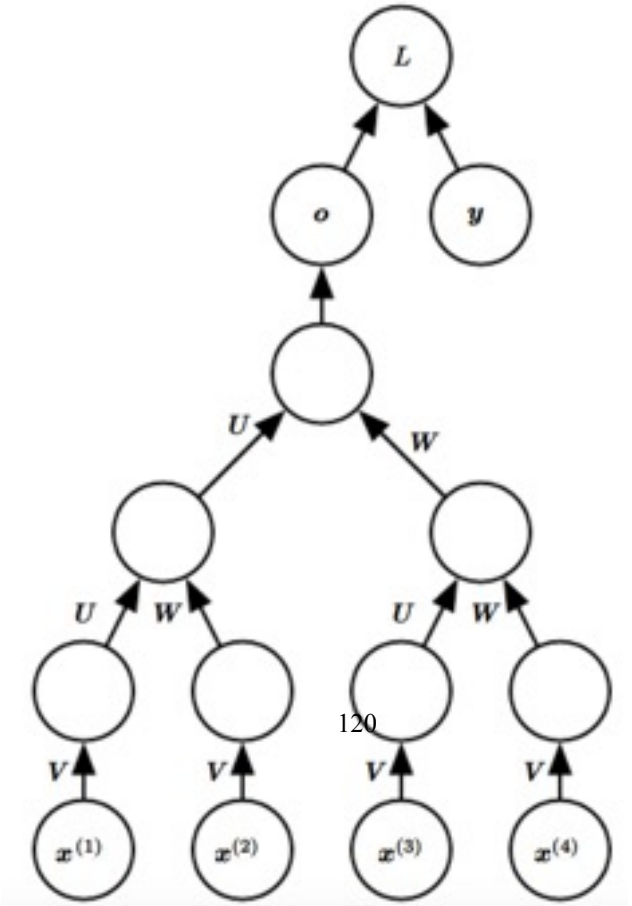
Recursive Neural Networks

Recursive Neural Networks

- They are yet another generalization of recurrent networks with a different kind of computational graph
- It is structured as a deep tree, rather than the chain structure of RNNs
- The typical computational graph for a recursive network is shown next

Computational graph of a Recursive Network

- It generalizes a recurrent network from a chain to a tree
- A variable sequence $\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)},,\boldsymbol{x}^{(t)}$ can be mapped to a fixed size representation (the output $\boldsymbol{o}$), with a fixed set of parameters (the weight matrices U, V, W)
- Figure illustrates supervised learning case in which target $\boldsymbol{y}$ is provided that is associated with the whole sequence



Advantage of Recursive over Recurrent Nets

- For a sequence of the same length τ, the depth (measured as the no. of compositions of nonlinear operations) can be reduced from τ to $O(\log \tau)$, which might help deal with long-term dependencies
- An open question is how best to structure the tree

Need for Recursive nets in NLP

- Deep learning based methods learn low-dimensional, real-valued vectors for word tokens, mostly from a large data corpus, successfully capturing syntactic and semantic aspects of text
- For tasks where the inputs are larger text units, e.g., phrases, sentences or documents, a compositional model is first needed to aggregate tokens into a vector with fixed dimensionality that can be used for other NLP tasks
- Models for achieving this fall into two categories: recurrent models and recursive models

Recurrent Model for NLP

- Recurrent models deal successfully with time series data
- The recurrent models generally consider no linguistic structure aside from the word order
- They were applied early on to NLP by modeling a sentence as tokens processed sequentially and at each step combining the current token with previously built embeddings
- Recurrent models can be extended to bidirectional ones from both left to right and right to left
- These models consider no linguistic structure aside from word order

Recursive Models for NLP

- Recursive neural models (also referred to as tree models) by contrast are structured by syntactic parse trees
- Instead of considering tokens sequentially, recursive models combine neighbors based on the recursive structure of parse trees, starting from the leaves and proceeding recursively in a bottom-up fashion until the root of the parse tree is reached
  - Ex: for the phrase the food is delicious, following the operation sequence ((the food) (is delicious)) rather than the sequential order (((the food) is) delicious)

Advantage of Recursive Model for NLP

- They have the potential of capturing long-distance dependencies
- Two tokens may be structurally closer to each other even though they are far away in word sequence
- Ex: a verb and its corresponding direct object can be far away in terms of tokens if many adjectives lie inbetween, but they are adjacent in the parse tree
- However parsing is slow and domain dependent
- See performance comparison with LSTM on four NLP tasks at https://nlp.stanford.edu/pubemnlp2015_2_jiwei.pdf

Structure of the Tree

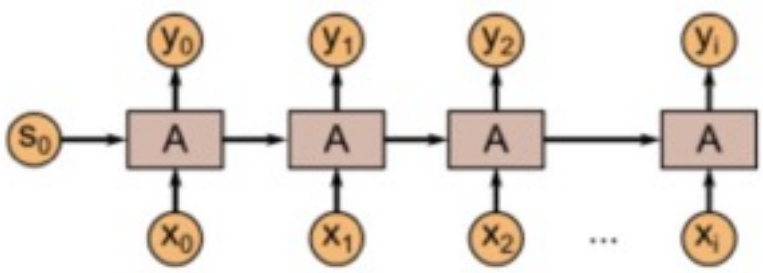
- One option is to have a tree structure that does not depend on the data, such as a balanced binary tree
- In some application domains, external methods can suggest the appropriate tree structure
  - Ex: when processing natural language sentences, the tree structure for the recursive network can be fixed to the structure of the parse tree of the sentence provided by a natural language parse
- Ideally, one would like the learner itself to discover and infer the tree structure that is appropriate for any given input

Variants of Recursive Net idea

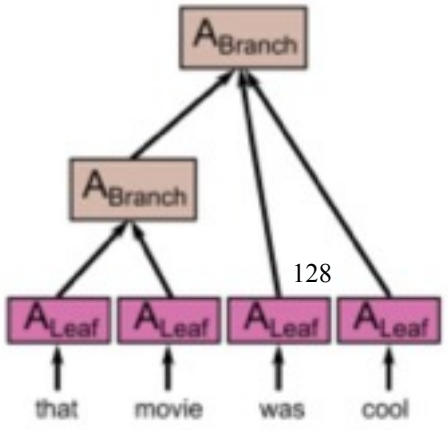
- Associate data with a tree structure and associate inputs and targets with individual nodes of the tree
  - The computation performed for each node does not have to be the artificial neuron computation (affine transformation of all inputs followed by a monotone nonlinearity)
  - Can use a tensor operations of bilinear forms
    - Previously found useful to model linear relationships between concepts when the concepts are represented by continuous vectors

Recursive NeuralNetworks

- Recursive neural networks are also called Tree Nets
  - Useful for learning tree-like structures
  - They are highly useful for parsing natural scenes and language



Recurrent Neural Net



Recursive Neural Net

Unrolling Recurrent and Tree Nets

- In RNNs, at each time step the network takes as input its previous state $s^{(t-1)}$ and its current input $x^{(t)}$ and produces an output $y^{(t)}$ and a new hidden state $s^{(t)}$.
- TreeNets, on the other hand, don't have a simple linear structure like that.
- With RNNs, you can 'unroll' the net and think of it as a large feedforward net with inputs $x^{(0)}, x^{(1)}, \ldots, x^{(T)}$, initial state $s^{(0)}$, and outputs $y^{(0)}, y^{(1)}, \ldots, y^{(T)}$, with T varying depending on the input data stream, and the weights in each of the cells tied with each other.
- You can also think of TreeNets by unrolling them – the weights in each branch node are tied with each other, and the weights in each leaf node are tied with each other.

Advantage of Recursive Nets

- The advantage of Recursive Nets is that they can be very powerful in learning hierarchical, tree-like structure.
- The disadvantages are, firstly, that the tree structure of every input sample must be known at training time.

Long-Term Dependencies: The Challenge

Topics in Sequence Modeling

- Recurrent Neural Networks

1. Unfolding Computational Graphs
2. Recurrent Neural Networks
3. Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent Networks
6. Recursive Neural Networks
7. The Challenge of Long-Term Dependencies
8. Echo-State Networks
9. Leaky Units and Other Strategies for Multiple Time Scales
10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Challenge of Long-Term Dependencies

- Neural network optimization face a difficulty when computational graphs become deep, e.g.,
  - Feedforward networks with many layers
  - RNNs that repeatedly apply the same operation at each time step of a long temporal sequence
- Gradients propagated over many stages tend to either vanish (most of the time) or explode (damaging optimization)
- The difficulty with long-term dependencies arise from exponentially smaller weights given to long-term interactions (involving multiplication of many Jacobians)

Vanishing and Exploding Gradient Problem

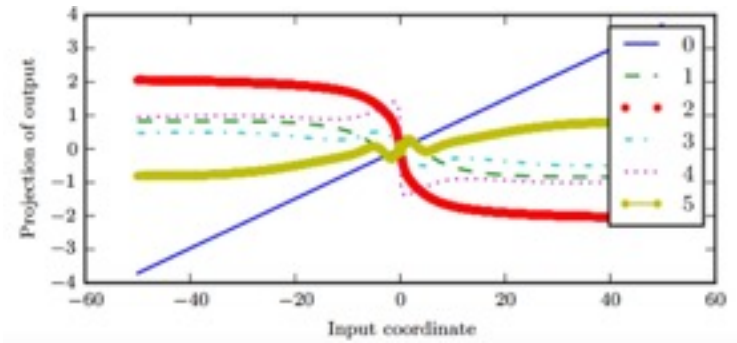
- Suppose a computational graph consists of repeatedly multiplying by a matrix W
- After t steps this is equivalent to multiplying by W^t
- Suppose W has an eigendecomposition $W = V\text{diag}(\lambda)V^{-1}$
  - In this case it is straightforward to see that

    $$W^t = (V\text{diag}(\lambda)V^{-1})^t = V\text{diag}(\lambda)^t V^{-1}$$

  - Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude and vanish if they are less than 1 in magnitude
- Vanishing gradients make it difficult to know which direction the parameters should move to improve cost
- Exploding gradients make learning unstable

Function Composition in RNNs

- RNNs involve composition of the same function multiple times, one per step
- These compositions can result in extremely nonlinear behavior



Composing many nonlinear functions:
tanh here
$\boldsymbol{h}$ has 100 dimensions mapped to a single dimension
Most of the space it has a small derivative and highly nonlinear elsewhere

- Problem particular to RNNs

Leaky Units and Multiple Time Scales

Topics

- Recurrent Neural Networks

1. Unfolding Computational Graphs
2. Recurrent Neural Networks
3. Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent Networks
6. Recursive Neural Networks
7. The Challenge of Long-Term Dependencies
8. Echo-State Networks
9. Leaky Units and Other Strategies for Multiple Time Scales

10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Multiple Time Scales

- Goal: to deal with long-term dependencies
- Design model so that
  1. Some model parts operate at fine-grained time scales
     - Handle small details
  2. Other parts operate at coarse time scales
     - Transfer information from the distant past to the present more efficiently

Building Fine and Coarse Time Scales

- Strategies to build fine and coarse time scales
  1. Adding skip connections through time
  2. Leaky units and a spectrum of different time scales
     - To integrate signals with different time constants
  3. Removal of some connections to model fine-grained time scales

Adding skip connections through time

- Add direct connections from variables in the distant past to variables in the present
- In an ordinary RNN, recurrent connection goes from time t to time $t+1$. Can construct RNNs with longer delays
- Gradients can vanish/explode exponentially wrt no. of time steps
- Introduce time delay of d to mitigate this problem
- Gradients diminish as a function of τ/d rather than τ
- Allows learning algorithm to capture longer dependencies
  - Not all long-term dependencies can be captured this way

Leaky units and a spectrum of time scales

- Rather than an integer skip of d time steps, the effect can be obtained smoothly by adjusting a real-valued α
- Running Average
  - Running average $\mu^{(t)}$ of some value $v^{(t)}$ is $\mu^{(t)} \leftarrow \alpha\mu^{(t-1)} + (1-\alpha)v^{(t)}$
  - Called a linear self-correction
  - When α is close to 1, running average remembers information from the past for a long time and when it is close to 0, information is rapidly discarded.
- Hidden units with linear self connections behave similar to running average. They are called *leaky units*.
- Can obtain product of derivatives close to 1 by having *linear* self-connections and a weight near 1 on those connections

Removing Connections

- Another approach to handle long-term dependencies
- Organize state of the RNN at multiple time scales
  - Information flowing more easily through long distances at the slower time scales
- It involves actively removing length one connections and replacing them with longer connections
- Skip connections add edges

Long-Short Term Memory and Other Gated RNNs

Topics in Sequence Modeling

- Recurrent Neural Networks

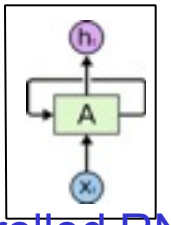
1. Unfolding Computational Graphs
2. Recurrent Neural Networks
3. Bidirectional RNNs
4. Encoder-Decoder Sequence-to-Sequence Architectures
5. Deep Recurrent Networks
6. Recursive Neural Networks
7. The Challenge of Long-Term Dependencies
8. Echo-State Networks
9. Leaky Units and Other Strategies for Multiple Time Scales
10. LSTM and Other Gated RNNs
11. Optimization for Long-Term Dependencies
12. Explicit Memory

Topics in LSTM and Other Gated RNNs

- From RNN to LSTM
- Problem of Long-Term Dependency
- BPTT and Vanishing Gradients
- Key intuition of LSTM as “state”
- LSTM Cell (three gates and two activations)
- LSTM usage example
- Equations for: forget gate, state update, output gate
- Step-by-Step LSTM Walk-through
- Gated RNNs

Recurrent Neural Networks

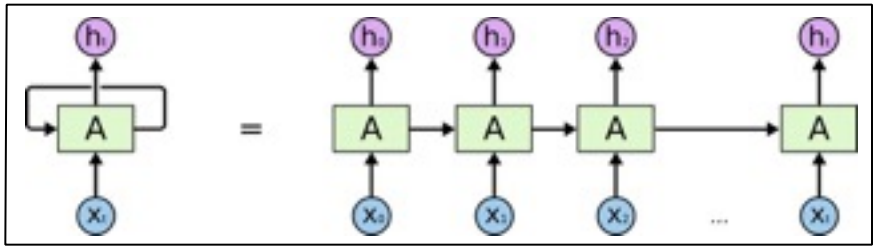
- RNNs have loops



A chunk of neural network A looks at some input x_t and outputs a value h_t

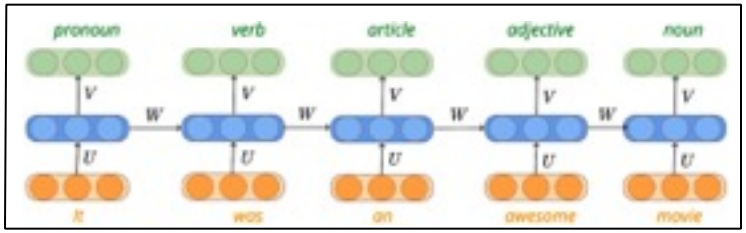
A loop allows information to be passed from one step of the network to the next

- An unrolled RNN



Chain-like structure reveals that RNNs are intimately related to sequences and lists

- Application to Part of Speech (POS) Tagging



Different Types of RNNs

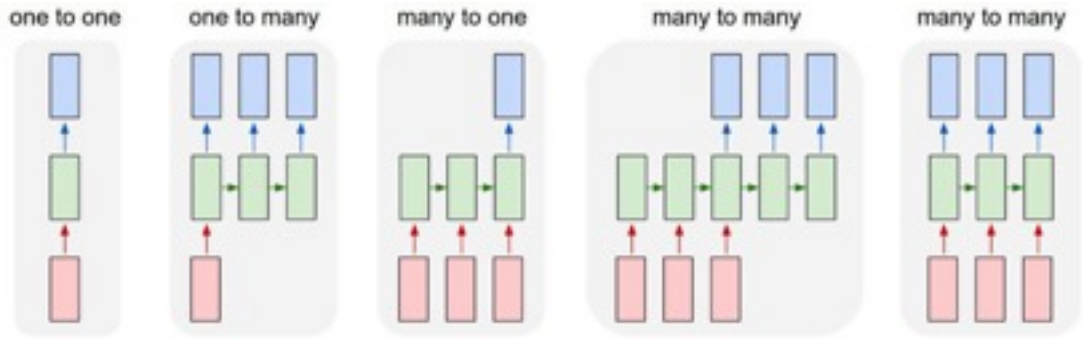


Image captioning
takes image and outputs a sentence

Sentiment Analysis
sentence classified as positive or negative

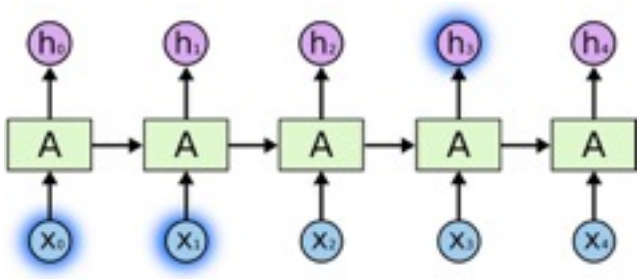
Machine Translation

Synced sequence
input and output
(e.g. video classification label each frame of video)

RNNs mainly used for
- Sequence Classification
  - Sentiment & Video Classification
- Sequence Labeling
  - POS & NE Tagging
- Sequence Generation
  - MT & Transliteration

Prediction with only recent previous information

- RNNs connect previous information to the present task
  - Previous video frames may help understand present frame
- Sometimes we need only look at recent previous information to predict
  - To predict the last word of "The clouds are in the *sky*" we don't need any further context. It is obvious that the word is "sky"



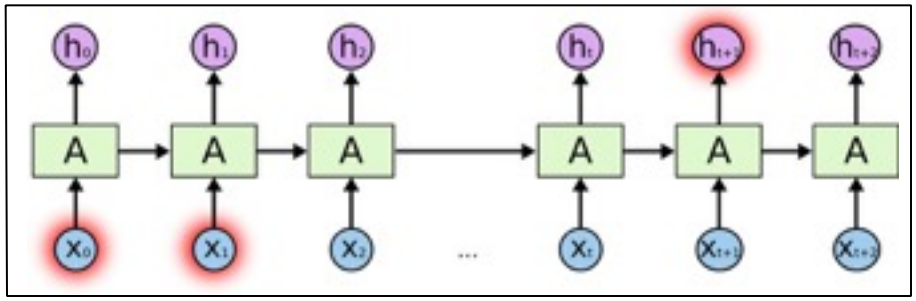
h_3 depends on x_0 and x_1

Long-Term Dependency

- RNNs work upon the fact that the result of an information is dependent on its previous state or previous n time steps.
- They have difficulty in learning long range dependencies.
- Example sentence: The man who ate my pizza has purple hair
  - Note: purple hair is for the man and not the pizza.
  - So this is a long dependency

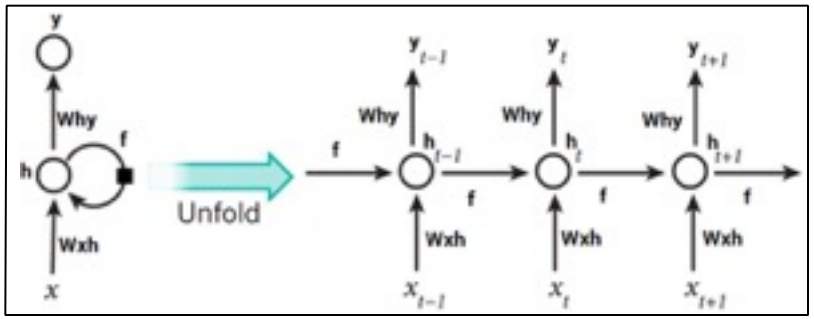
Long-term dependency and Backprop

- There are cases where we need even more context
  - To predict last word in I grew up in France……I speak *French*
  - Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is *French*
- Gap between relevant information and where it is needed is large



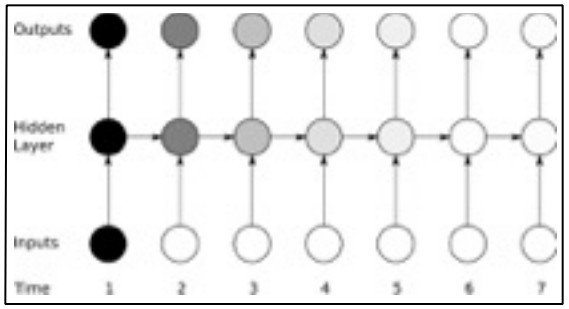
- Learning to store information over extended time intervals via RNN backpropagation takes a very long time due to decaying error backflow (discussed next)

Backpropagation Through Time (BPTT)



If y_t is predicted and
$\bar{y}_t$ is training value
the cross entropy loss is
$E_t(\bar{y}_t, y_t) = -\bar{y}_t \log(y_t)$

Total error is the sum
$E(\bar{y}, y) = -\sum \bar{y}_t \log(y_t)$



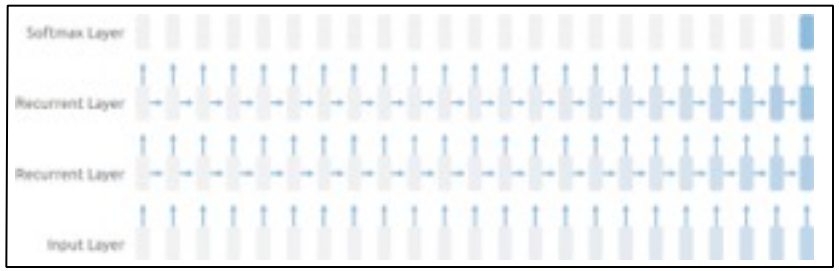
To backpropagate error, apply the chain rule.
The error after the third time step wrt the first:
$\partial E/\partial W = \partial E/\partial y_3 * \partial y_3/\partial h_3 * \partial h_3/\partial y_2 * \partial y_2/\partial h_1$ ..
and there is a long dependency.

Vanishing gradient problem

If a single gradient approached 0, all gradients rush to zero exponentially fast due to the multiplication.
Such states would no longer help the network to learn anything.

Vanishing and Exploding Gradients

- Vanishing gradient problem

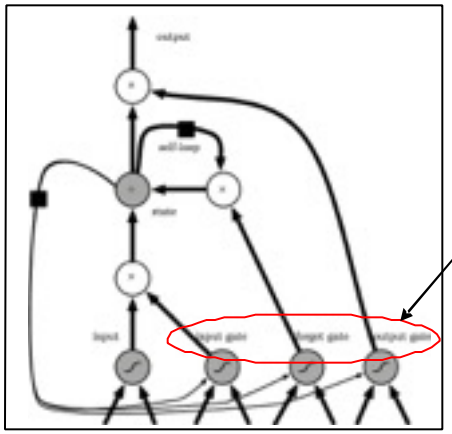


Contribution from the earlier steps becomes insignificant in the gradient

- Exploding gradient problem:
  - Gradients become very very large due to a single or multiple gradient values becoming very high.
- Vanishing gradient is more concerning
  - Exploding gradient easily solved by clipping the gradients at a predefined threshold value.
- Solution: gated RNNs
  - LSTM, GRU (Gated Recurrent Units), a variant of LSTM

Key intuition of LSTM is "State"

- A persistent module called the cell-state
- Note that "State" is a representation of past history
- It comprises a common thread through time
- Cells are connected recurrently to each other
  - Replacing hidden units of ordinary recurrent networks

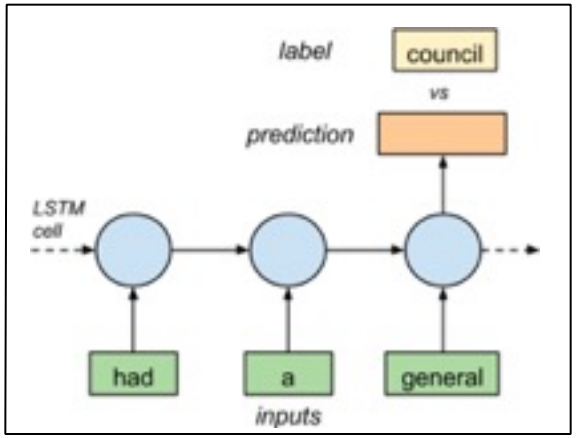


There are three sigmoid gates:
An input gate (i),
A forget gate (f)
An output gate (o)

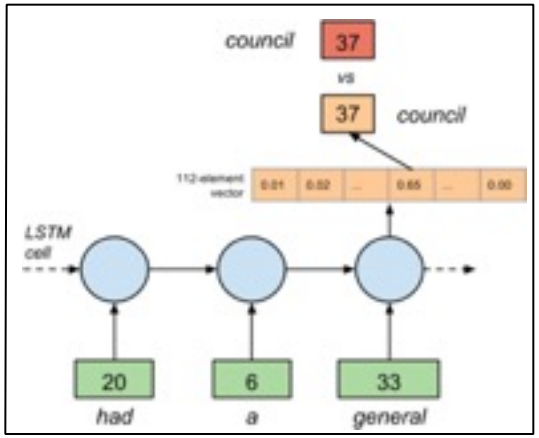
LSTM usage example

- “long ago , the mice had a general council to consider what measures they could take to outwit their common enemy , the cat……” Aesop’s fables story with 112 words

Training



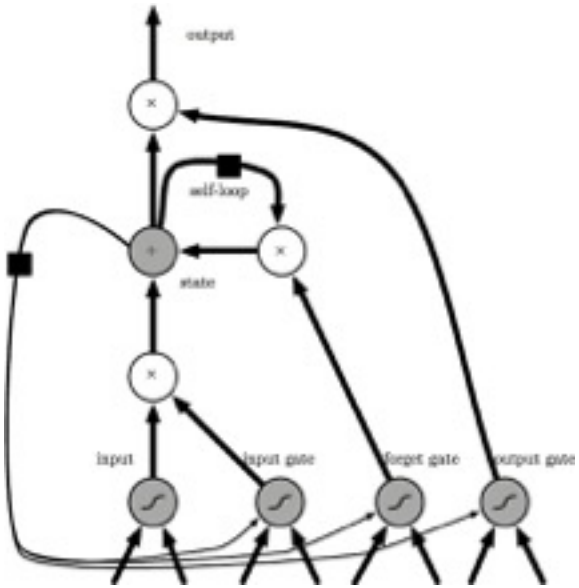
Prediction



https://towardsdatascience.com/lstm-by-example-using-tensorflow-feb0c1968537

LSTM Recurrent Network Cell

- Input feature computed with regular artificial neuron unit
  - Its value can be accumulated into the state
  - If the sigmoidal input gate allows it
- State unit has linear self-loop
  - Weight controlled by forget gate
- Output of cell can be shut off by output gate
- All units: sigmoid nonlinearity
  - Input gate can be any squashing
- State unit can also be used as an extra input to gating units
- Black square indicates delay of single time step

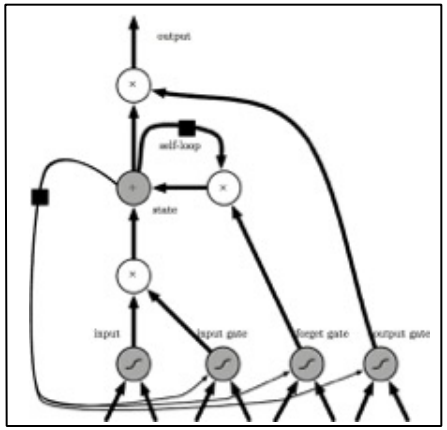


Core contribution of LSTM

- Clever idea of introducing self-loops to produce paths where the gradient can flow for long durations
- A crucial addition: make weight on this self-loop conditioned on the context, rather than fixed
  - By making weight of this self-loop gated (controlled by another hidden unit), time-scale can be changed dynamically
  - Even for an LSTM with fixed parameters, time scale of integration can change based on the input sequence
    - Because time constants are output by the model itself

State perturbed by linear operations

- Cell-state perturbed only by a few linear operations at each time step



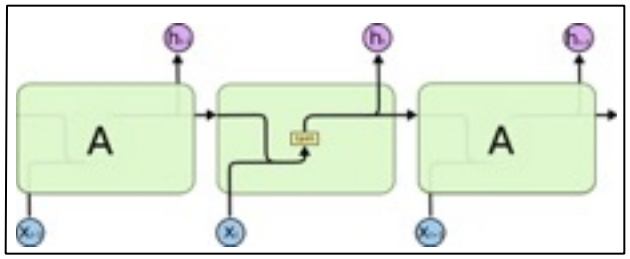
- Since cell state connection to previous cell states is interrupted only by the linear operations of multiplication and addition, LSTMs can remember short-term memories (*i.e.* activity belonging to the same "episode") for a very long time

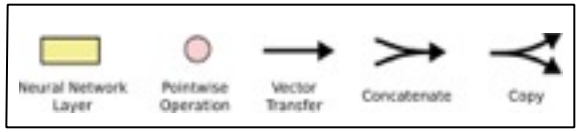
LSTM success

- LSTM found extremely successful in:
  - Unconstrained handwriting recognition
  - Speech recognition
  - Handwriting generation, Machine Translation
  - Image Captioning, Parsing

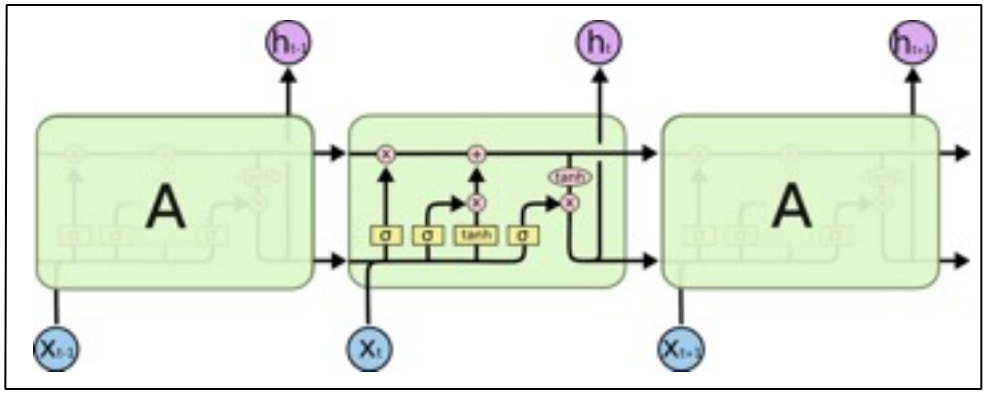
RNN vs LSTM cell

- RNNs have the form of a repeating chain structure
- The repeating module has a simple structure such as tanh



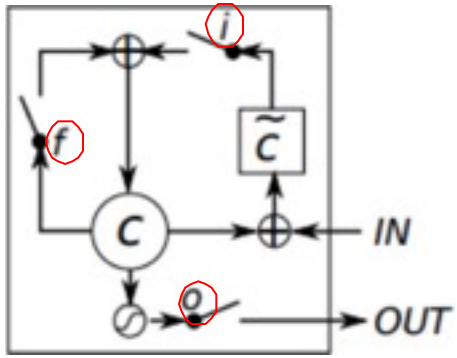


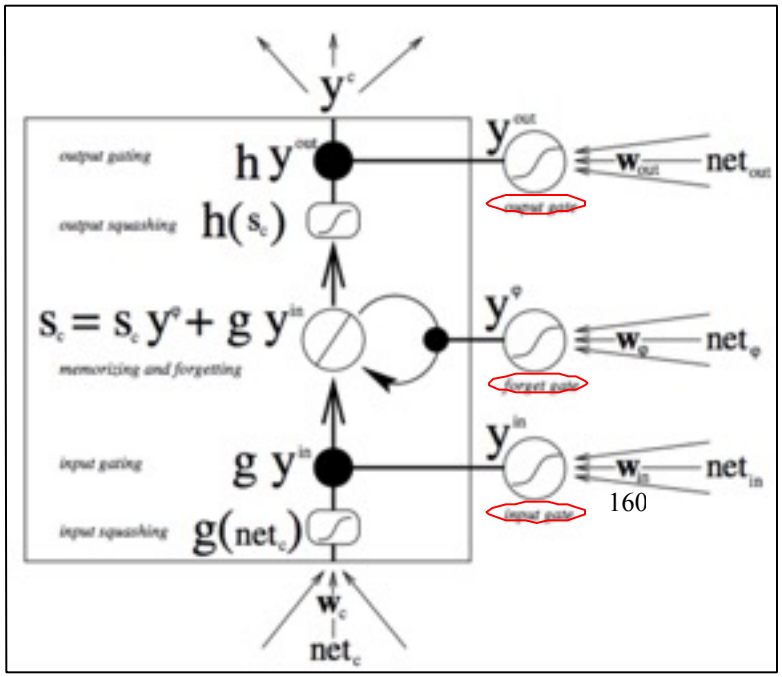
- LSTMs also have a chain structure
  - but the repeating module has a different structure



LSTM cell

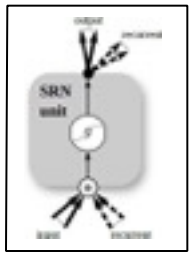
- Like RNNs , LSTMs also have a chain like structure
- But repeating module has a slightly different structure
- Instead of a single layer, multiple layers interact
  - They have three sigmoid gates:
    - An input gate (i),
    - A forget gate (f)
    - An output gate (o)

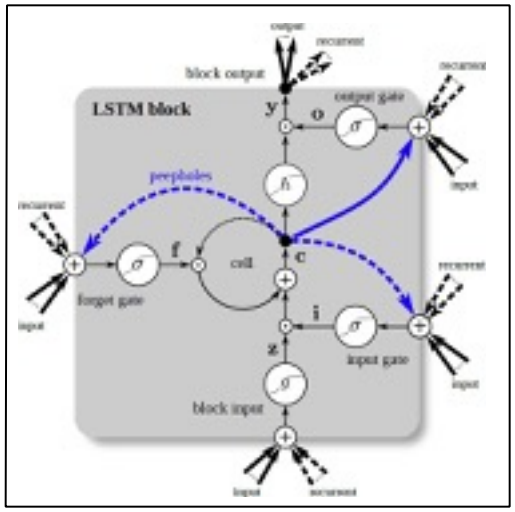


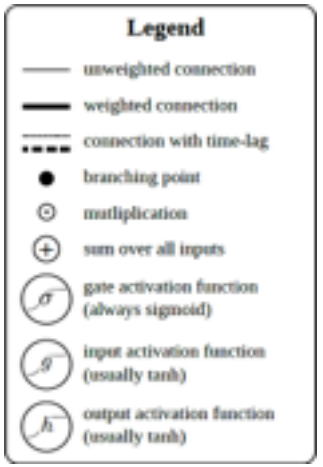


LSTM Cell equations

Simple RNN unit







Weights for an LSTM layer:

- Input weights: $\mathbf{W}_z, \mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o \in \mathbb{R}^{N \times M}$
- Recurrent weights: $\mathbf{R}_z, \mathbf{R}_i, \mathbf{R}_f, \mathbf{R}_o \in \mathbb{R}^{N \times N}$
- Peephole weights: $\mathbf{p}_i, \mathbf{p}_f, \mathbf{p}_o \in \mathbb{R}^N$
- Bias weights: $\mathbf{b}_z, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^N$

Vector formulas for LSTM forward pass:

$$
\begin{aligned}
\bar{\mathbf{z}}^t &= \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z \\
\mathbf{z}^t &= g(\bar{\mathbf{z}}^t) && \textit{block input} \\
\bar{\mathbf{i}}^t &= \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i \\
\mathbf{i}^t &= \sigma(\bar{\mathbf{i}}^t) && \textit{input gate} \\
\bar{\mathbf{f}}^t &= \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f \\
\mathbf{f}^t &= \sigma(\bar{\mathbf{f}}^t) && \textit{forget gate} \\
\mathbf{c}^t &= \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t && \textit{cell} \\
\bar{\mathbf{o}}^t &= \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o \\
\mathbf{o}^t &= \sigma(\bar{\mathbf{o}}^t) && \textit{output gate} \\
\mathbf{y}^t &= h(\mathbf{c}^t) \odot \mathbf{o}^t && \textit{block output}
\end{aligned}
$$

BPTT deltas inside the LSTM block:

$$
\begin{aligned}
\delta \mathbf{y}^t &= \Delta^t + \mathbf{R}_z^T \delta \mathbf{z}^{t+1} + \mathbf{R}_i^T \delta \mathbf{i}^{t+1} + \mathbf{R}_f^T \delta \mathbf{f}^{t+1} + \mathbf{R}_o^T \delta \mathbf{o}^{t+1} \\
\delta \bar{\mathbf{o}}^t &= \delta \mathbf{y}^t \odot h(\mathbf{c}^t) \odot \sigma'(\bar{\mathbf{o}}^t) \\
\delta \mathbf{c}^t &= \delta \mathbf{y}^t \odot \mathbf{o}^t \odot h'(\mathbf{c}^t) + \mathbf{p}_o \odot \delta \bar{\mathbf{o}}^t + \mathbf{p}_i \odot \delta \bar{\mathbf{i}}^{t+1} \\
&\quad + \mathbf{p}_f \odot \delta \bar{\mathbf{f}}^{t+1} + \delta \mathbf{c}^{t+1} \odot \mathbf{f}^{t+1} \\
\delta \bar{\mathbf{f}}^t &= \delta \mathbf{c}^t \odot \mathbf{c}^{t-1} \odot \sigma'(\bar{\mathbf{f}}^t) \\
\delta \bar{\mathbf{i}}^t &= \delta \mathbf{c}^t \odot \mathbf{z}^t \odot \sigma'(\bar{\mathbf{i}}^t) \\
\delta \bar{\mathbf{z}}^t &= \delta \mathbf{c}^t \odot \mathbf{i}^t \odot g'(\bar{\mathbf{z}}^t)
\end{aligned}
$$

Sigmoid used for gate activation, tanh used as input and output activation, point-wise multiplication of vectors is $\odot$

Source: Greff et.al., LSTM: a search space odyssey, Transactions on Neural networks and Learning Systems, arXiv 2017

Accumulating Information over Longer Duration

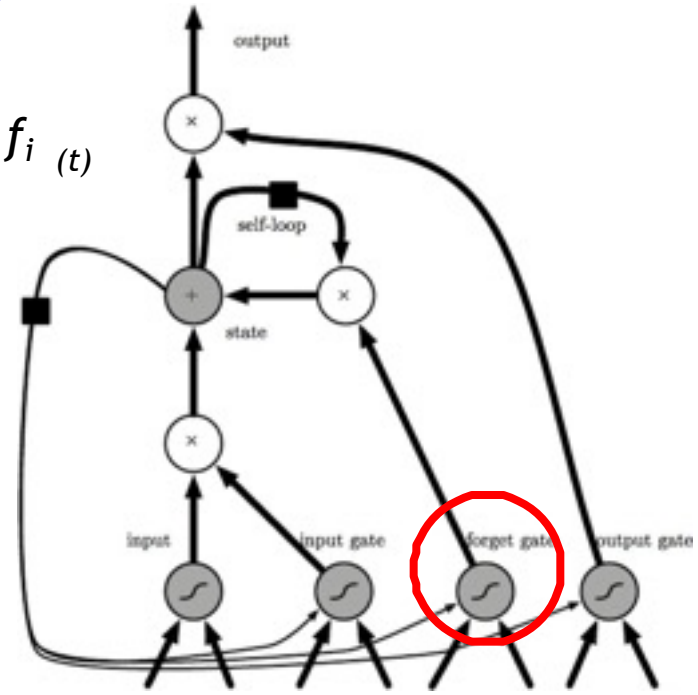
- Leaky Units Allow the network to accumulate information
  - Such as evidence for a particular feature or category
  - Over a long duration
- However, once the information has been used, it might be useful to forget the old state
  - E.g., if a sequence is made of sub-sequences and we want a leaky unit to accumulate evidence inside each sub-sequence, we need a mechanism to forget the old state by setting it to zero
- Gated RNN:
  - Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it

Description of the LSTM cell

- Forward propagation equations for the LSTM cell are given in the next slide
  - In the case of a shallow recurrent network architecture
  - Deeper architectures have also been successively used
- Instead of a unit that simply applies an elementwise nonlinearity to the affine transformation of inputs and recurrent units LSTM recurrent units 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 RNN
  - But has more parameters and a system of gating units that controls the flow of information

Equation for LSTM forget gate

- The most important component is the state unit $s_i^{(t)}$ that has a linear self-loop similar to the leaky units
- However the self-loop weight (or the associated time constant) is controlled by a forget gate unit $f_i^{(t)}$ (for time step t and cell i) that sets this weight to a value between 0 and 1 via a sigmoid unit



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

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

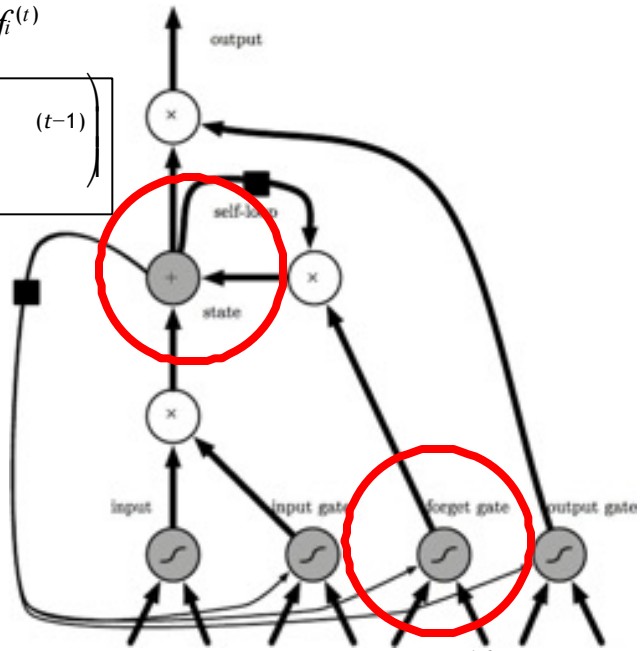
Equation for LSTM internal state update

- The LSTM cell internal state is updated as follows
  - But with conditional self-loop weight $f_i^{(t)}$

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j}^{f} x_j^{(t)} + \sum_j W_{i,j}^{f} h_j^{(t-1)}\right)$$

  - where $\boldsymbol{b}$, $\boldsymbol{U}$ and $\boldsymbol{W}$ respectively denote the biases, input weights and recurrent weights into the LSTM cell
- External input gate unit $g_i^{(t)}$ is
  - computed similar to forget gate
  - With a sigmoid unit to obtain a gating value between 0 and 1 but with its own parameters

$$g_i^{(t)} = \sigma\left(b_i^{g} + \sum_j U_{i,j}^{g} x_j^{(t)} + \sum_j W_{i,j}^{g} h_j^{(t-1)}\right)$$



Equation for output of LSTM cell

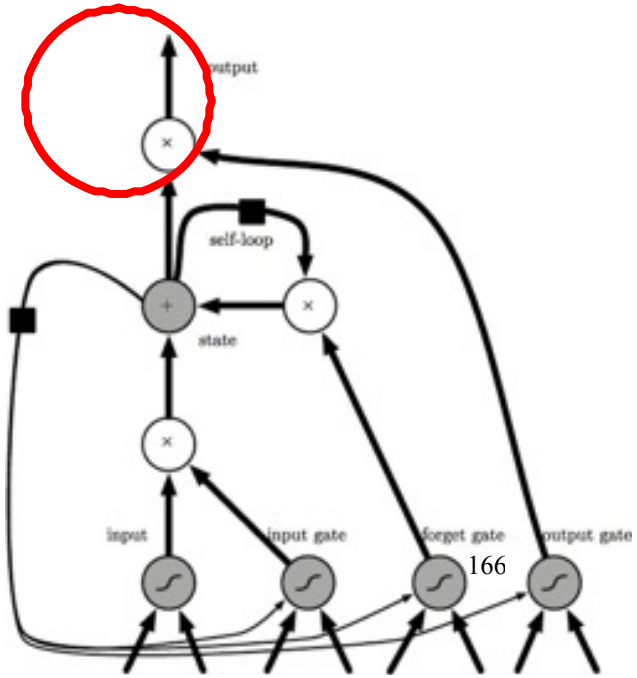
- Output $h_i^{(t)}$ of the LSTM cell can be shut off via the output gate $q_i^{(t)}$ which also uses a sigmoid unit for gating

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right)$$

$$q_i^{(t)} = \text{sigm}\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

$\boldsymbol{b}^0$, $\boldsymbol{U}^0$ and $\boldsymbol{W}^0$ are biases, input weights and recurrent weights

- Among the variants one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit This would require three additional parameters



Power of LSTMs

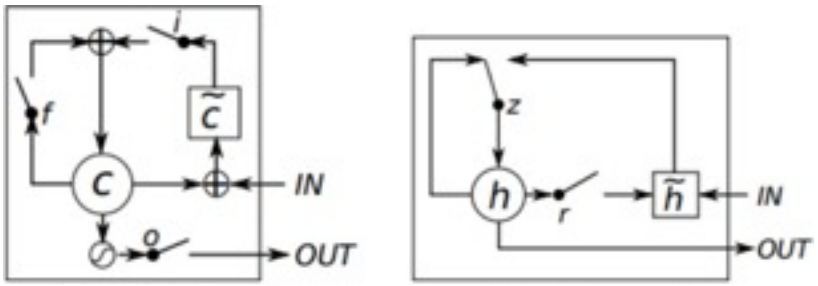
- LSTM networks have been shown to learn long-term dependencies more easily than simple recurrent architectures
  - First on artificial data sets
  - Then on challenging sequential tasks
- Variants and alternatives to LSTM have been studied and used and are discussed next

Other gated RNNs

- What pieces of the LSTM architecture are actually necessary?
- What other successful architectures could be designed that allow the network to dynamically control the time scale and forgetting behavior of different units?
- Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as gated recurrent units or GRUs
- Main difference with LSTM
  - Single gating unit simultaneously controls the forgetting factor and decision to update state unit

GRUs

- GRUs have simpler structure and are easier to train.
- Their success is primarily due to the gating network signals that control how the present input and previous memory are used, to update the current activation and produce the current state.
- These gates have their own sets of weights that are adaptively updated in the learning phase.
- We have just two gates here, the reset and the update gate



(a) Long Short-Term Memory

(b) Gated Recurrent Unit

Idea of Gated RNNs

- Like leaky units, gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode
- Leaky units do this with connection weights that are manually chosen or were parameters α

  generalizing the concept of discrete skipped connections
- Gated RNNs generalize this to connection weights that may change with each time step

Update equations for Gated RNNs

- Update equations are

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right)$$

- Where $\boldsymbol{u}$ stands for the update gate and $\boldsymbol{r}$ for reset gate. Their value is defined as usual:

$$u_i^{(t)} = \sigma\left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)} \right)$$

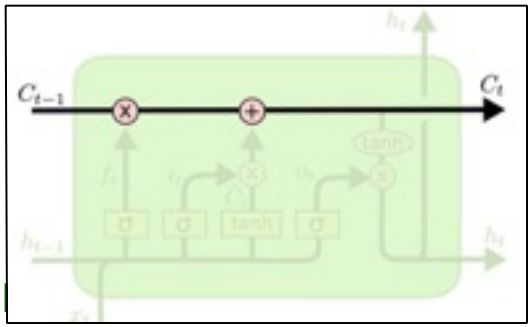
and

$$r_i^{(t)} = \sigma\left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)} \right)$$

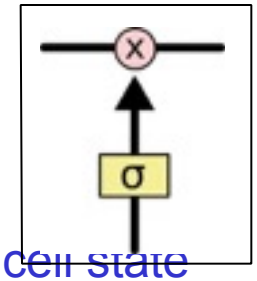
- Reset and update gates can individually ignore parts of the state vector
  - Update gates act conditional leaky integrators that can linearly gate any dimension thus choosing to copy it (at one extreme of sigmoid) or completely ignore it (at the other extreme) by replacing it by the new target state value (towards which the leaky integrator converges
  - Reset gates control which parts of the state get used to compute next target state, introducing an additional nonlinear effect in the relationship between past state and future state

Core idea behind LSTM

- The key to LSTM is the cell state, $\boldsymbol{C}_t$, the horizontal line running through the top of the diagram
- Like a conveyor belt
  - Runs through entire chain with minor interactions
  - LSTM does have the ability to remove/add information to cell state regulated

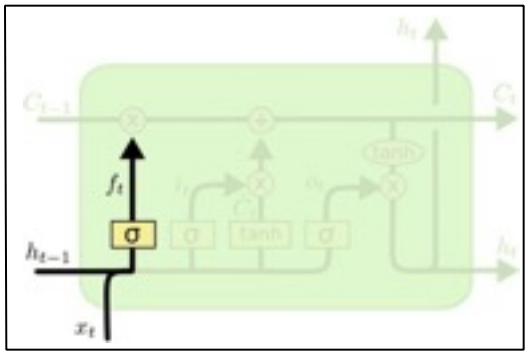


- Gates are an optional way to let information through
- Consist of a sigmoid and a multiplication operation
- Sigmoid outputs a value between 0 and 1
  - 0 means let nothing through
  - 1 means let everything through
- LSTM has three of these gates, to protect and control cell state



Step-by-step LSM walk through

- Example of language model: predict next word based on previous ones
  - Cell state may include the gender of the present subject
- First step: information to throw away from cell state



Called *forget gate layer*

It looks at $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ and outputs a number between 0 and 1 for each member of $\boldsymbol{C}_{t-1}$ for whether to forget

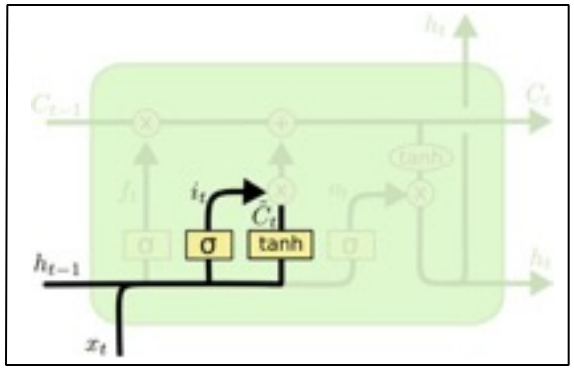
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

In language model

consider trying to predict the next word based on all previous ones
The cell state may include the gender of the present subject
so that the proper pronouns can be used
When we see a new subject we want to forget old subject

LSM walk through: Second step

- Next step is to decide as to what new information we're going to store in the cell state



In the Language model, we'd want to add the gender of the new subject to the cell state, to replace the old One we are forgetting

This has two parts:

first a sigmoid layer called *Input gate layer*: decides which values we will update

Next a tanh layer creates a vector of new candidate values $\tilde{C}_t$ that could be added to the state.

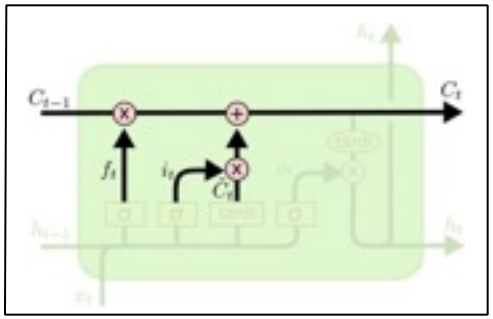
In the third step we will combine these two to create an update to the state

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

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

LSTM walk-through: Third Step

- It's now time to update old cell state C_{t-1} into new cell state C_t
  - The previous step decided what we need to do
  - We just need to do it



We multiply the old state by f_t, forgetting the things we decided to forget earlier.

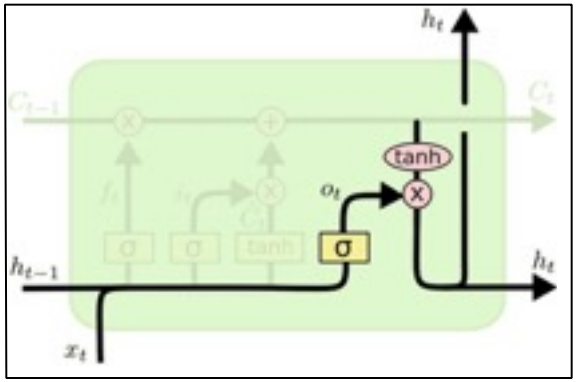
Then we add $i_t * \tilde{C}_t$
This is the new candidate values, scaled by how much we decided to update each state value

In the Language model,
this is where we'd actually drop the information about the old subject's gender and add the new information,
as we decided in previous steps

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

LSTM walk-through: Fourth Step

- Finally we decide what we are going to output



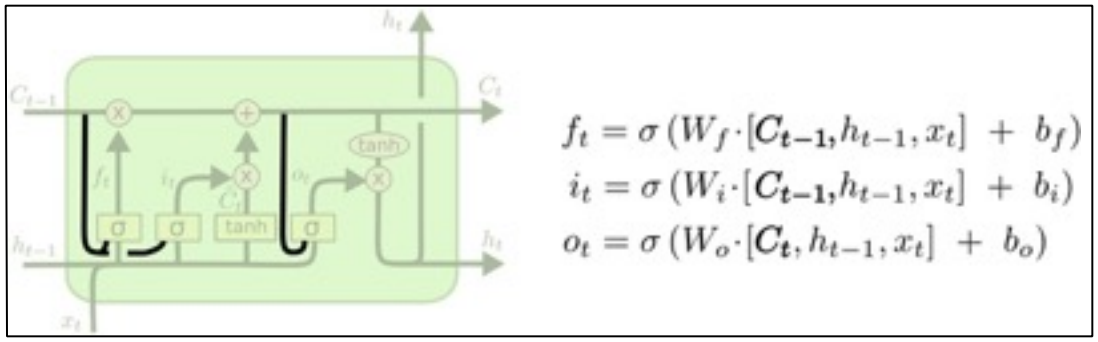
This output will be based on our cell state, but will be a filtered version.
First we run a sigmoid layer which decides what parts of cell state we're going to output.
Then we put the cell state through tanh (to push values to be between -1 and 1)
and multiply it by the output of the sigmoid gate, so that we output only the parts we decided to

For the Language model,
Since it just saw a subject it might want to output
relevant to a verb, in case that is what is coming next, e.g., it might output whether the subject is singular or plural so that we know what form a verb should be conjugated into if that's what follows next.

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

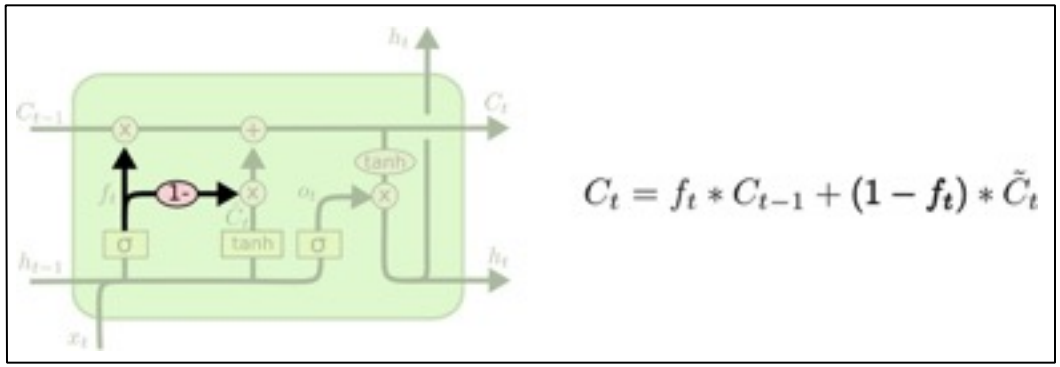
Variants of LSTM

- There are several minor variants of LSTM
  - LSTM with “peephole” connections



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

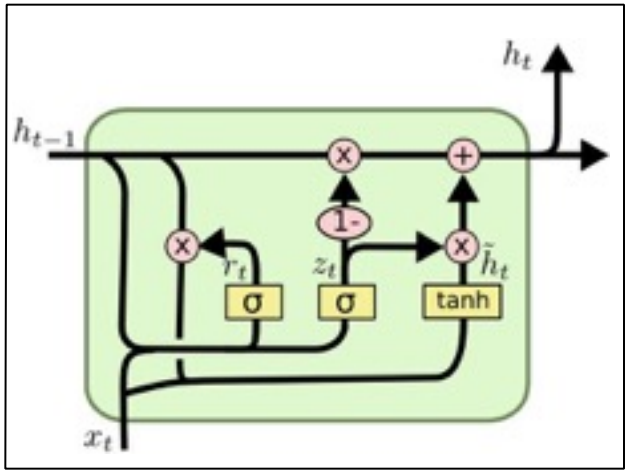
  - Coupled forget and input gates



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

Gated Recurrent Unit (GRU)

- A dramatic variant of LSTM
  - 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 LSTM models
  - Has become increasingly popular



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

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

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

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