ECBM E4040
Neural Networks and Deep Learning

Sequence Modeling:
Recurrent and Recursive Neural Nets

Zoran Kostić
Columbia University
Electrical Engineering Department &
Data Sciences Institute

COLUMBIA ENGINEERING
The Fu Foundation School of Engineering and Applied Science

References and Acknowledgments

- **Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville, http://www.deeplearningbook.org/ , chapter 10.**
- **Lecture material by bionet group / Prof. Aurel Lazar (http://www.bionet.ee.columbia.edu/).**
- **NVIDIA GPU Teaching Kit (NVIDIA and New York University)**

Sequence Modeling:
Recurrent and Recursive Nets (RRNs)

Unfolding Computational Graphs

Recurrent Neural Networks

Encoder-Decoder Sequence-to-Sequence Architectures

Deep Recurrent and Recursive Neural Networks

The Long Short-Term Memory and Other Gated Networks

From MLPs and CNNs to RNNs

MLPs and CNNs are feedforward neural networks whose connections do not form cycles:

- MLPs and CNNs can only map from input to output vectors

A convolutional network is a neural network that is specialized for processing a grid of values X such as an image, can readily scale to images with large width and height, and some convolutional networks can process images of variable size.

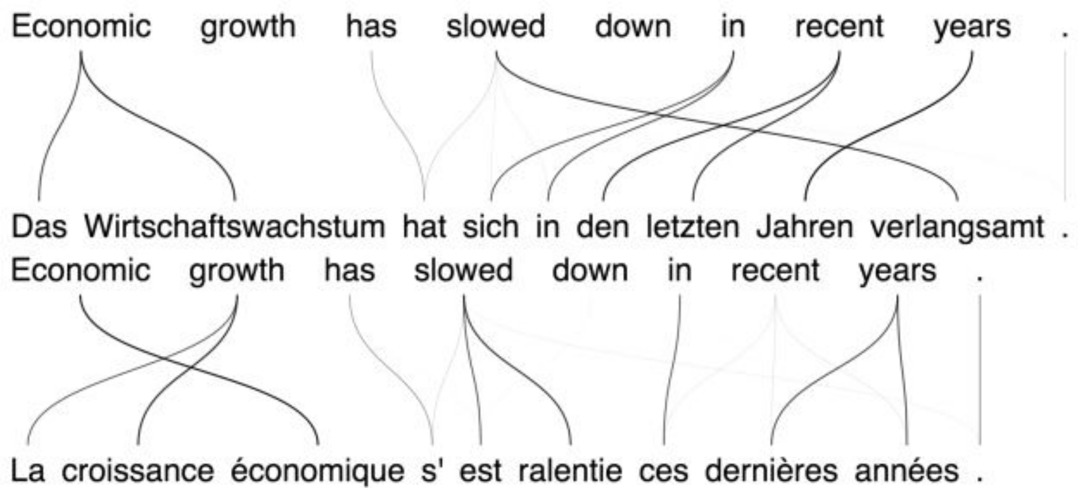
From MLPs and CNNs to RNNs

RNNs are neural networks that allow cyclic connections:

- RNNs can in principle map from the entire history of previous inputs to each output
- Any function computable by a Turing Machine can be computed by a finite size RNN

RNN Key Features

A RNN can be trained to map an input sequence into an output sequence that is not necessarily of the same length.


Economic growth has slowed down in recent years .
Das Wirtschaftswachstum hat sich in den letzten Jahren verlangsamt .
Economic growth has slowed down in recent years .
La croissance économique s' est ralentie ces dernières années .

https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-3/

RNN Key Features

An RNN can be made deep in many ways:

- the hidden recurrent state can be broken down into groups organized hierarchically;
- deeper computation can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps;
- the path-lengthening effect can be mitigated by introducing skip connections.

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures.

Recurrent Neural Networks - Overview

Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data that:

- **are specialized for processing a sequence of values**

  $(x^1, x^2, x^3, \ldots \; x^{\tau-1}, x^{\tau})$.

- **can scale to much longer sequences than would be practical for networks without sequence-based specialization.**
- **most recurrent networks can also process sequences of variable length.**

Computational Graph

...can be Unfolded

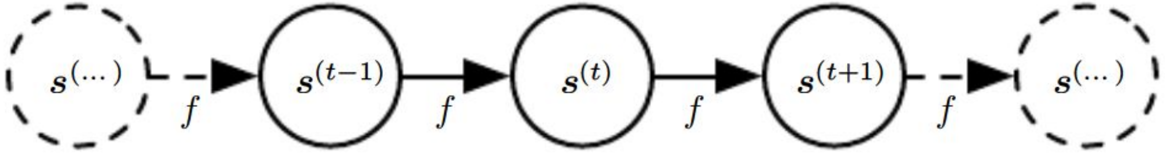
We consider the classical form of a dynamical system

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

where $\boldsymbol{s}^t$ is called the state of the system at time t. For $t = 3$ we have

$$\boldsymbol{s}^3 = f(\boldsymbol{s}^2; \boldsymbol{\theta}) = f(f(\boldsymbol{s}^1; \boldsymbol{\theta}); \boldsymbol{\theta}) = f(f(f(\boldsymbol{s}^0; \boldsymbol{\theta}); \boldsymbol{\theta}); \boldsymbol{\theta}).$$

Unfolding this equation by repeatedly applying the definition in this way yields an expression that does not involve recurrence and can be represented by a traditional directed acyclic computational graph:

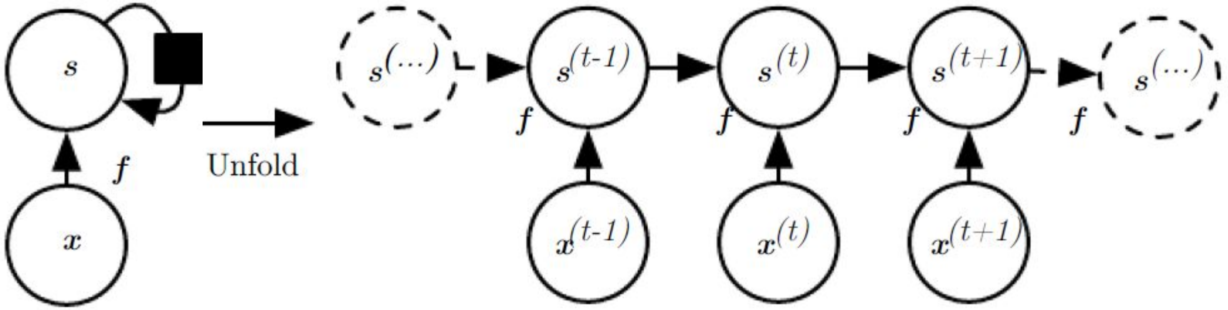


Computational Graph with an External Signal …. can be Unfolded

Consider a dynamical system driven by an external signal $\boldsymbol{x}^t$,

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

Note that the state contains information about the whole past sequence.



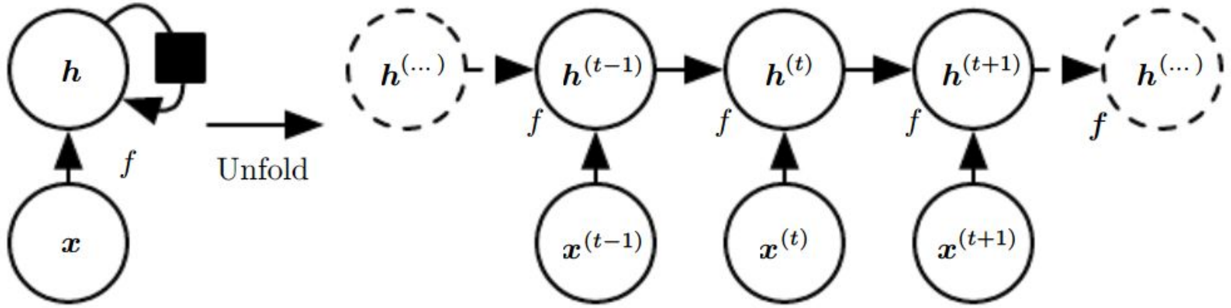
(Left) Circuit diagram. The black square indicates a delay of 1 time step.
(Right) Same network as an unfolded computational graph; each node is now associated with one particular time instance

Computational Graph Unfolding Hidden Nodes

To indicate that the state is in the hidden units of the network, we now use the variable $\boldsymbol{h}$ to represent the state:

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

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



A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time.

Representing Unfolded Recurrence

We can represent the unfolded recurrence after t steps with a function g^t:

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

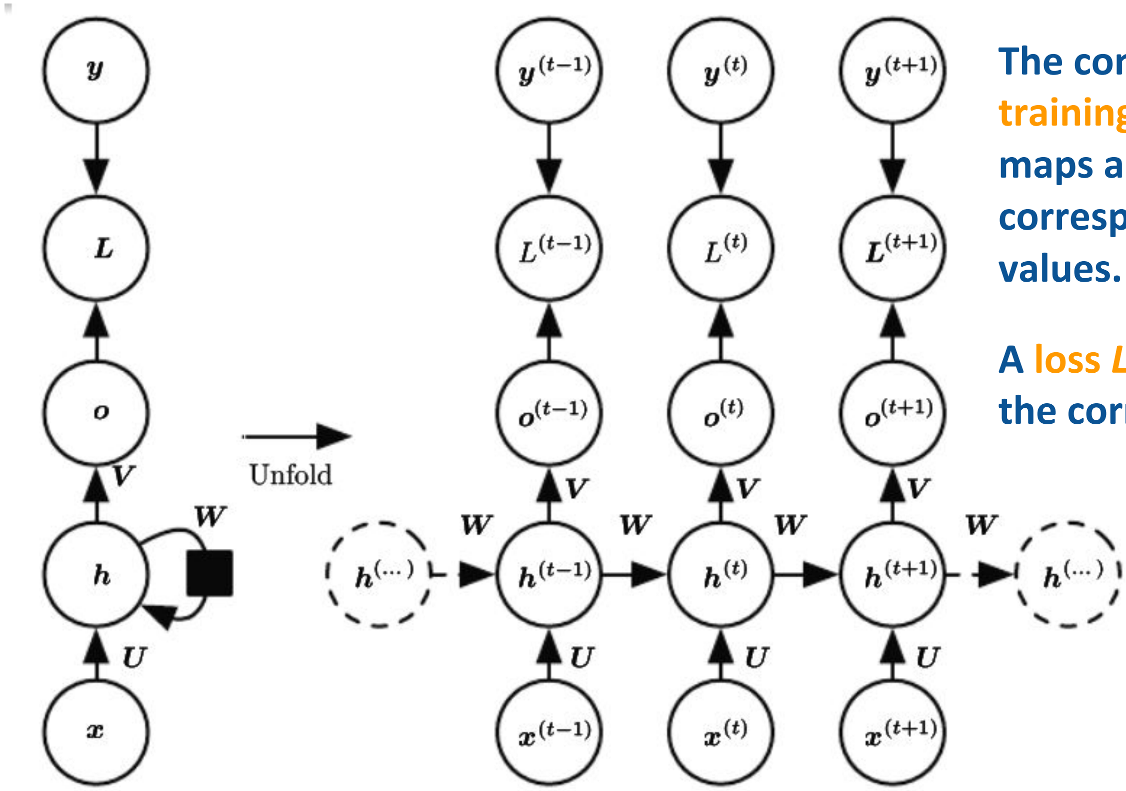
The function g^t takes as input the past sequence in its entirety $(\boldsymbol{x}^t, \boldsymbol{x}^{t-1}, \boldsymbol{x}^{t-2}, \ldots, \boldsymbol{x}^2, \boldsymbol{x}^1)$ and produces the current state. The unfolded recurrent structure allows us to factorize g^t into repeated application of the function f.

Representing Unfolded Recurrence

The unfolding process introduces two major advantages:

- regardless of the sequence length, the learned model always has the 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;
- it is possible to use the same transition function f with the same parameters at every time step.

Universal RNNs



The computational graph to compute the training loss of a recurrent network that maps an input sequence of *x* values to a corresponding sequence of output *o* values.

A loss *L* measures how far each *o* is from the corresponding training target *y*.

Universal RNNs

Any function computable by a Turing machine can be computed by a Finite Size RNN.

RNN Specification

To instantiate the computational graph, assume:

- **a hyperbolic tangent activation function,**
- **the output *o* as giving the unnormalized log probabilities of each possible value of a discrete variable,**
- **the softmax operation as a post-processing step to obtain a vector $\hat{y}$ of normalized probabilities over the output.**

RNN Specification

Forward propagation begins with the initial state $\boldsymbol{h}^0$:

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

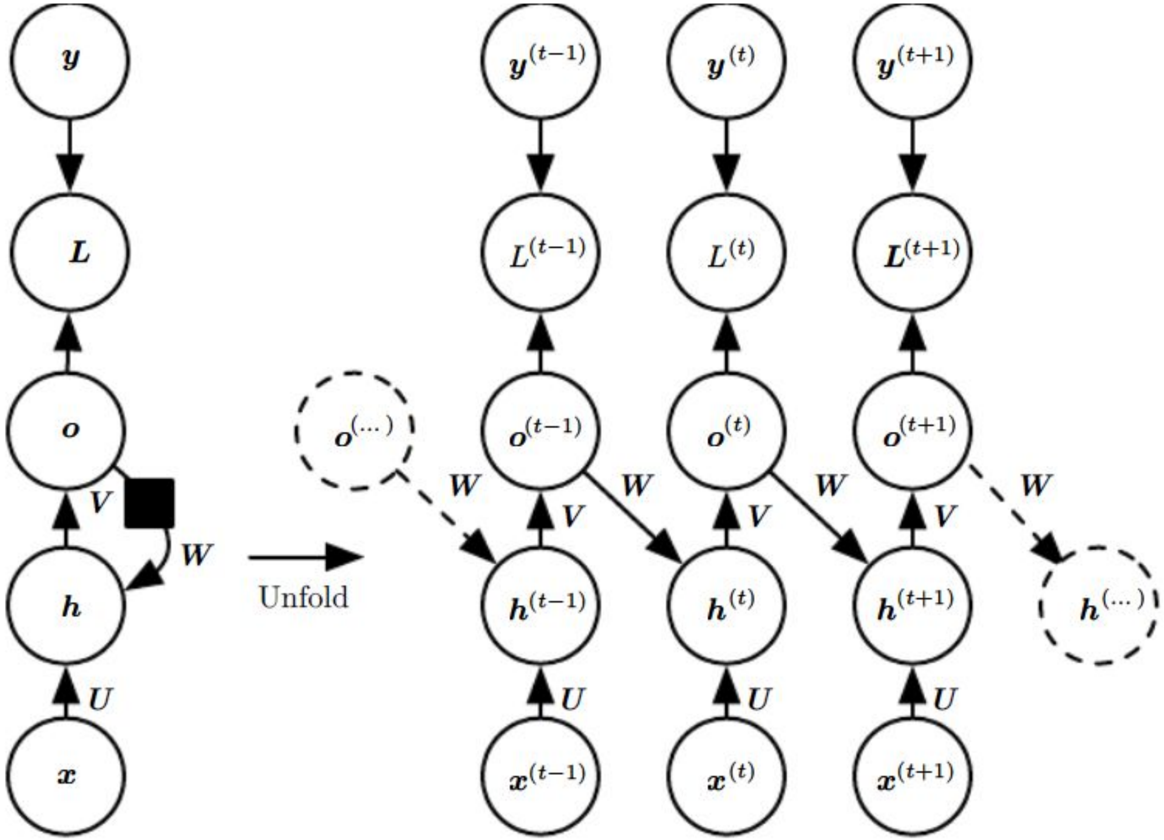
Design Patterns for Building RNNs

Examples of important design patterns for recurrent neural networks:

- **RNNs that produce an output at each time step and have recurrent connections between hidden units (already discussed);**
- **RNNs that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step;**
- **RNNs with recurrent connections between hidden units, that read an entire sequence and then produce a single output.**

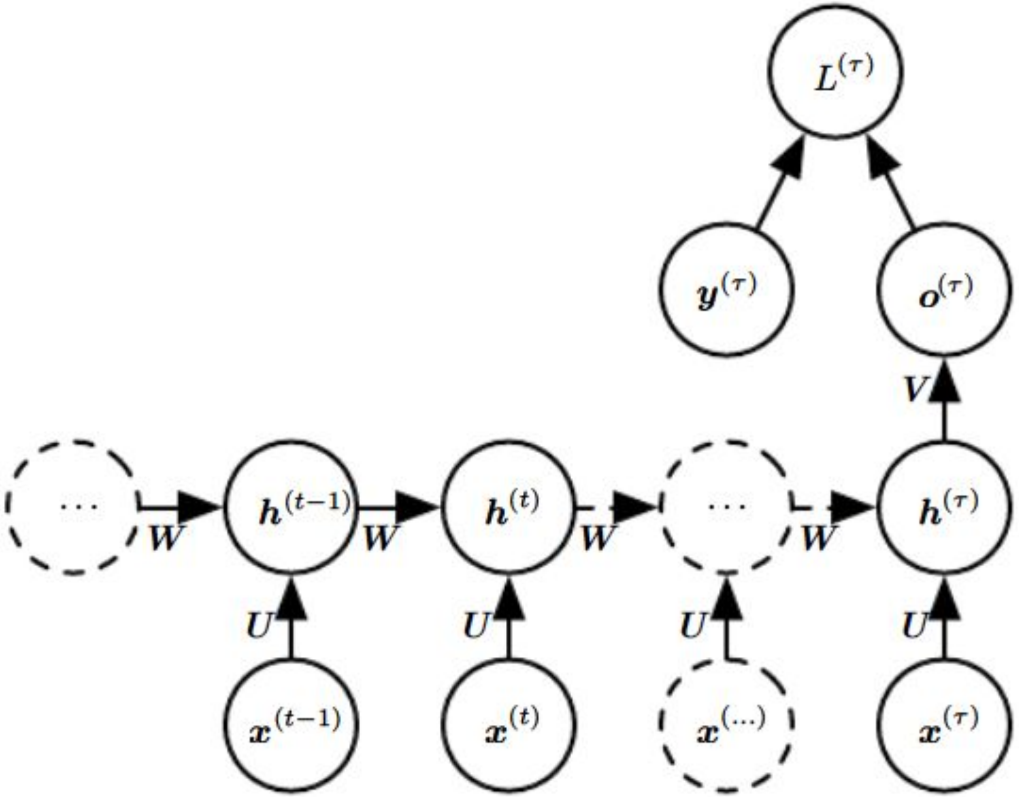
Design Patterns for Building RNNs

RNN whose only recurrence is the feedback from the output to the hidden layer.



Design Patterns for Building RNNs

Time-unfolded recurrent neural network with a single output at the end of the sequence.



RNNs for Estimation of Next Element

We typically train an RNN to estimate the conditional distribution of the next sequence element y^t given the past inputs.

This may mean that we maximize the log-likelihood

$$\log p\,(y^t \mid x^1, x^2, \dots x^t);$$

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

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

RNNs for Estimation of Next Element

Decomposing the joint probability over the sequence of y values as a series of one-step probabilistic predictions is one way to capture the full joint distribution across the whole sequence.

The computation has a high complexity, however.

Fully-Connected Graphical Models

We consider here the case where the RNN models only a sequence of scalar random variables $\mathbb{Y} = (\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^\tau)$, with no additional inputs $\mathbf{x}$.

We parametrize the joint distribution of these observations using the chain rule for conditional probabilities:

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

The loss function is then

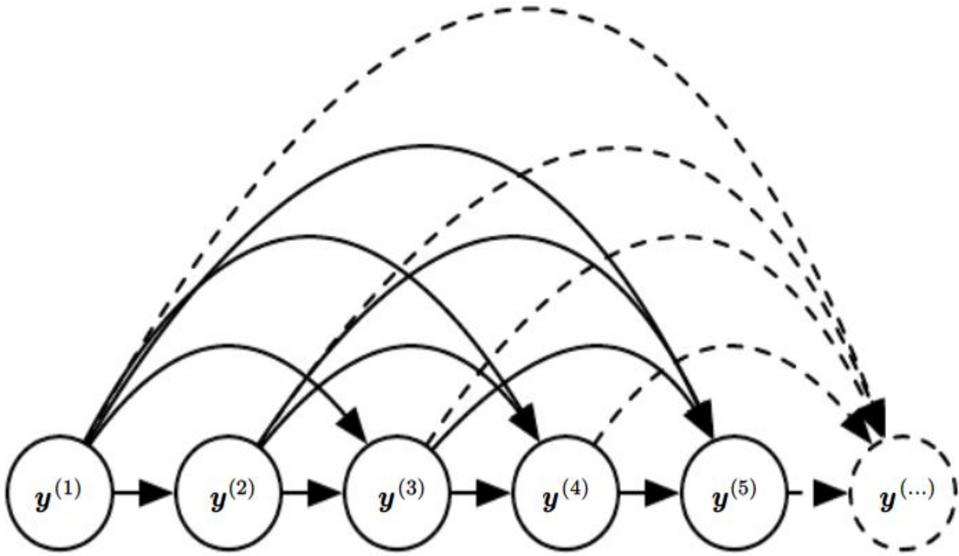
$$L = \sum_t L^t \quad \text{with} \quad L^t = -\log \mathbb{P}(\mathbf{y}^t | \mathbf{y}^{t-1}, \mathbf{y}^{t-2}, \ldots, \mathbf{y}^1).$$

Fully-Connected Graphical Models

Fully connected graphical model for a sequence $y^1, y^2, \dots\ y^t$:

Every past observation y^i may influence the conditional distribution of some y^t, $t > i$ given previous values.

Parametrizing the graphical model directly according to this graph might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence.

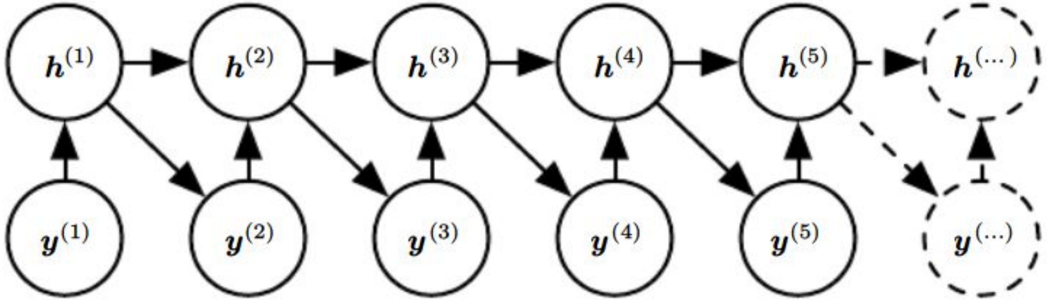


Fully-Connected Graphical Models - Efficiency

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

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

we can obtain a very efficient parametrization:



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

Fully-Connected Graphical Models - Efficiency

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

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

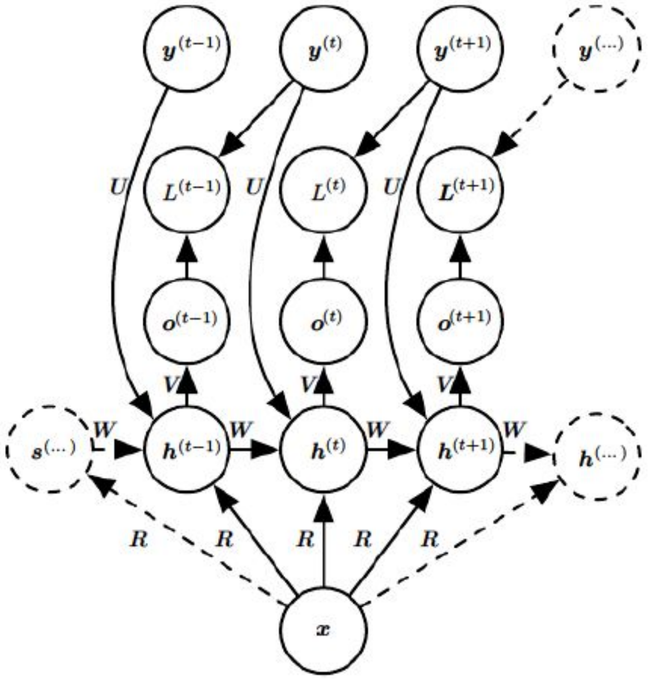
Fully-Connected Graphical Models - Efficiency

Incorporating the h^t nodes in the graphical model decouples the past and the future, acting as an intermediate quantity between them.

A variable y^i in the distant past may influence a variable y^t via its effect on h.

The structure of the previous graph shows that the model can be efficiently parametrized by using the same conditional probability distributions at each time step.

RNNs with a Single Input Vector



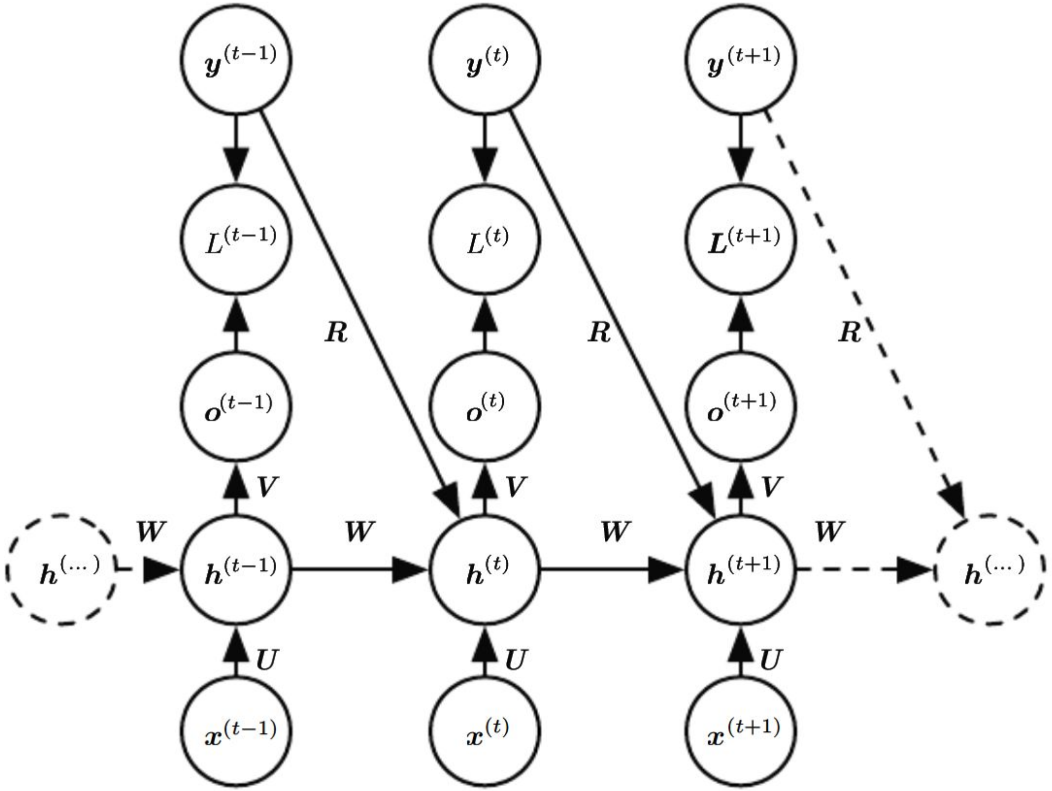
Interaction between input x and each hidden unit vector h^t is parametrized by R.

RNN that maps a fixed-length vector x into a distribution over sequences Y: appropriate for tasks such as image captioning,

where a single image is used as input to a model that then produces a sequence of words describing the image.

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

A Conditional Recurrent Neural Network

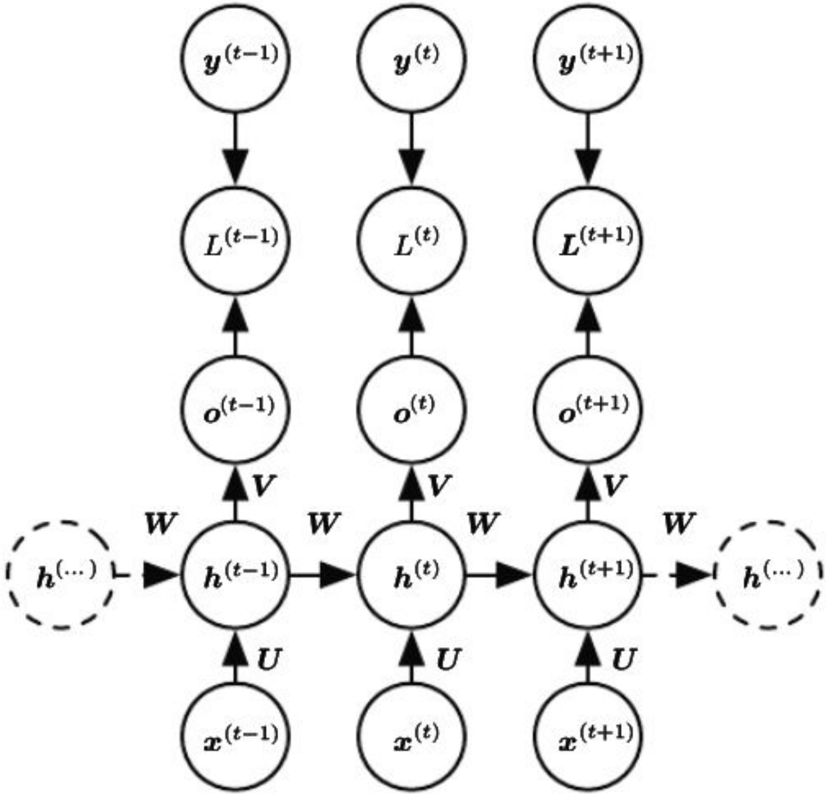


A conditional RNN mapping a variable-length sequence of x values into a distribution over sequences of y values of the same length.

This RNN contains connections from the previous output to the current state.

These connections allow this RNN to model an arbitrary distribution over sequence of y given sequences of x of the same length.

A Conditional Recurrent Neural Network



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

Bidirectional RNN

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

For example, in speech recognition, the correct interpretation of the current sound as a phoneme may depend on the next few phonemes because of co-articulation,

potentially may even depend on the next few words because of the linguistic dependencies between nearby words:

if there are 2 interpretations of the current word that are both acoustically plausible, we may have to look far into future (and past) to disambiguate them.

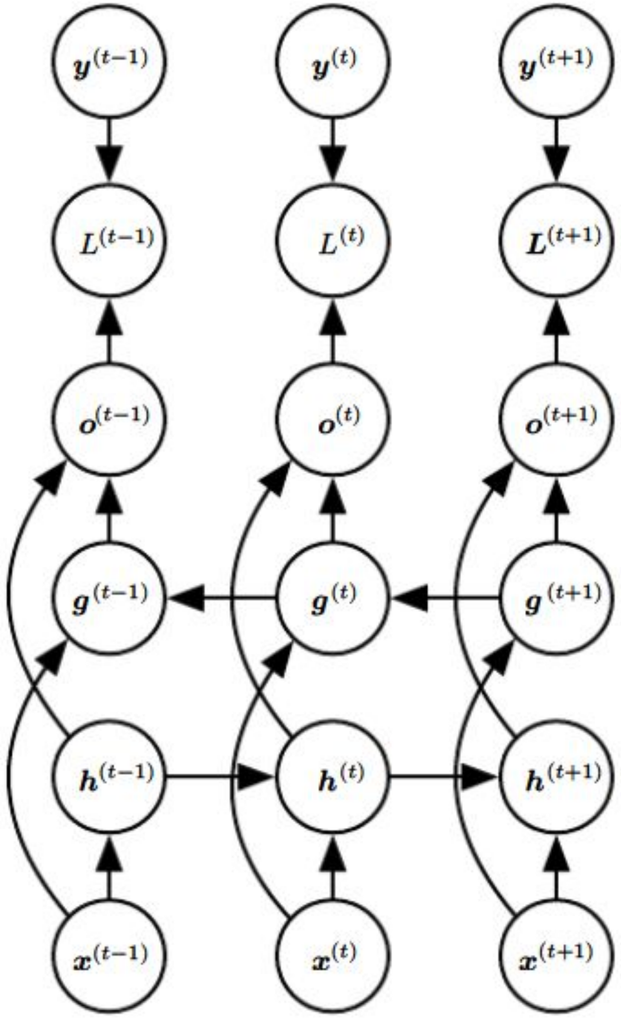
This is also true of handwriting recognition and many other sequence-to-sequence learning tasks.

Bidirectional RNN

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

The h recurrence propagates information forward in time (towards the right) while the g recurrence propagates information backward in time (towards the left).

Thus at each point t, the output units o^t can benefit from a relevant summary of the past in its h^t input and from a relevant summary of the future in its g^t input.



Bidirectional RNN

In the typical bidirectional RNN, h^t stands for the state of the sub-RNN that moves forward through time and g^t stands for the state of the sub-RNN that moves backward through time.

This allows the output units o^t to compute a representation that depends on both the past and the future

but is most sensitive to the input values around time t, without having to specify a fixed-size window around t

(as one would have to do with a feedforward network, a convolutional network, or a regular RNN with a xed-size look-ahead buffer).

Bidirectional RNN in Multiple Dimensions

This idea can be naturally extended to 2-dimensional input, such as images, by having four RNNs, each one going in one of the four directions: up, down, left, right.

At each point (i, j) of a 2-D grid, an output O_{ij} could then compute a representation that would capture mostly local information but could also depend on long-range inputs, if the RNN is able to learn to carry that information.

Compared to a convolutional network, RNNs applied to images are typically more expensive but allow for long-range lateral interactions between features in the same feature map.

Encoder-Decoder RNN Architecture

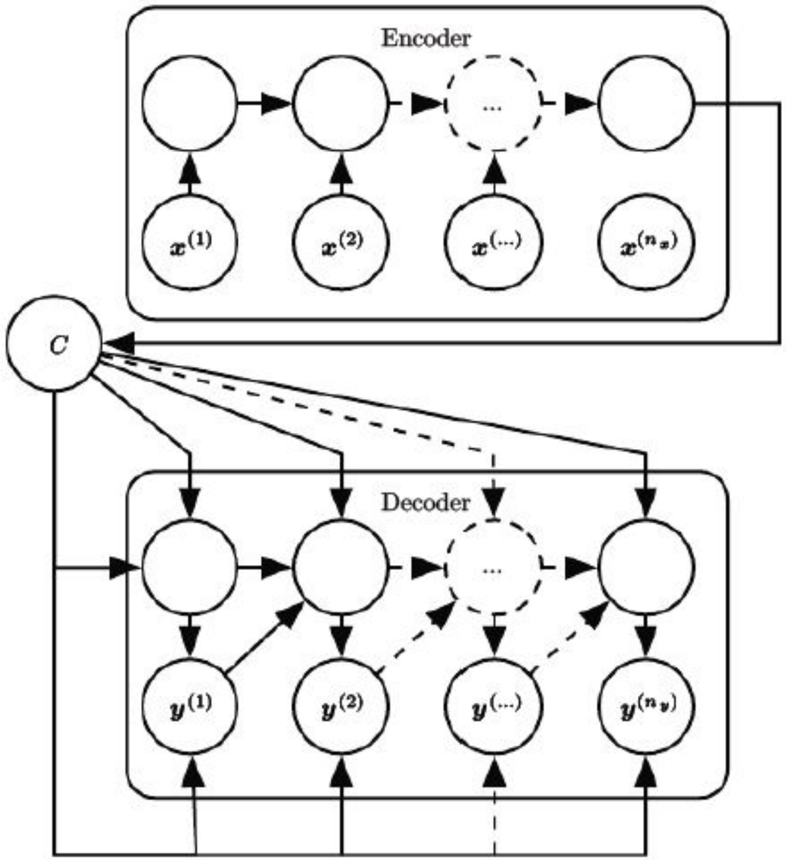
A RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length.

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

The input to an RNN is often call the context. We want to produce a representation of this context, C. The context C might be a vector or sequence of vectors that summarizes the input sequence

$$\mathbf{X} = (\mathbf{x}^1, \ldots, \mathbf{x}^{n_x})$$

Encoder-Decoder RNN Architecture



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

A decoder or writer or output RNN is conditioned on that fixed-length vector to generate the output sequence

$$\mathbf{Y} = (\mathbf{y}^1, \ldots, \mathbf{y}^{n_y})$$

Encoder-Decoder RNN Architecture

The two RNNs are trained jointly to maximize the average of

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

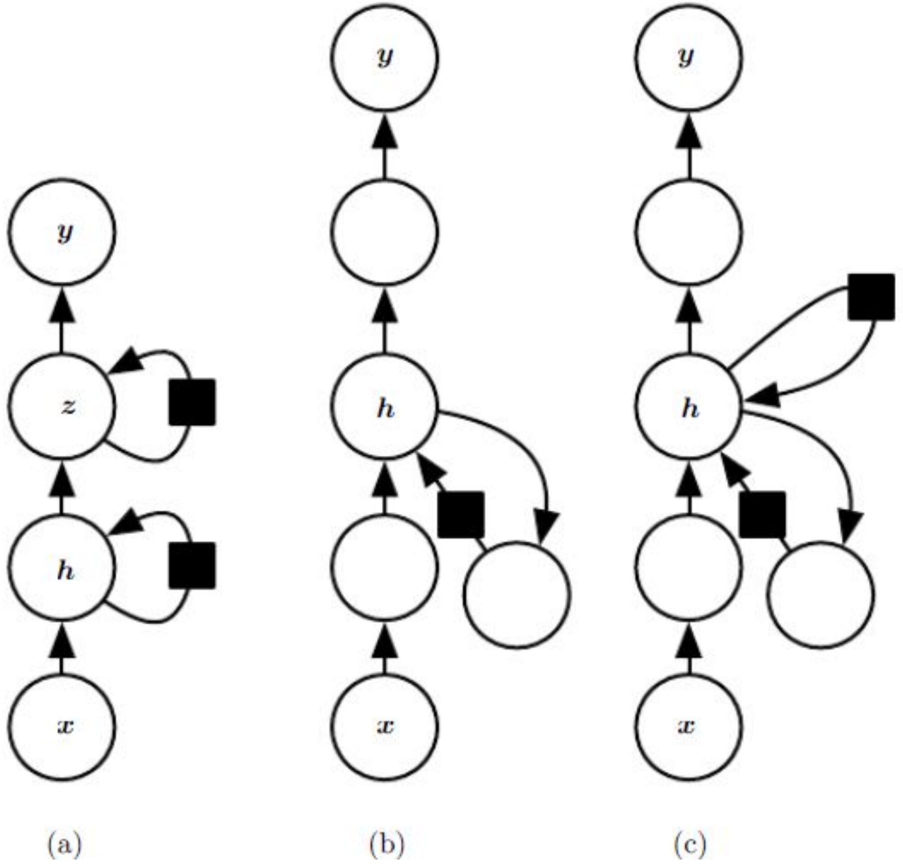
over all the pairs of x and y sequences in the training set.

The last state $\boldsymbol{h}^{n_x}$ of the encoder RNN is typically used as a representation C of the input sequence that is provided as input to the decoder RNN.

Deep Recurrent Networks

A recurrent neural network can be made deep in many ways:

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



Recursive Neural Networks

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

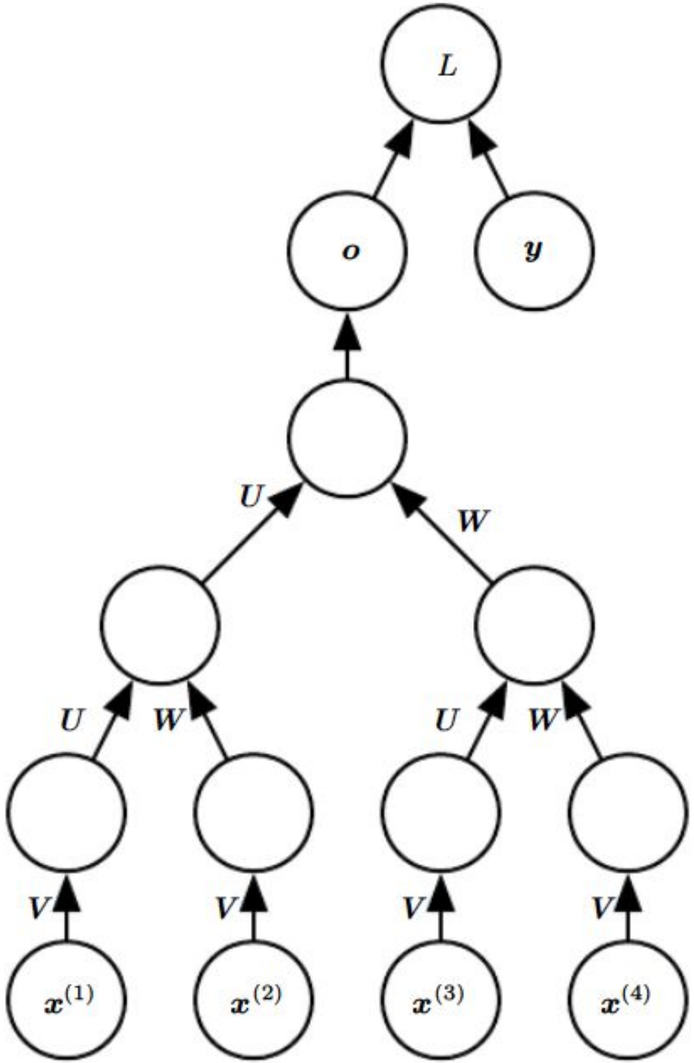
Recursive networks have been successfully applied to processing data structures as input to neural nets, in natural language processing as well as in computer vision.

Recursive Neural Networks

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

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

The graph illustrates a supervised learning case in which some target y associated with the whole sequence is provided



RNNs: The Challenge of Long-Term Dependencies
Extreme Gradient Behavior

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

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

RNNs: Training = Backpropagation + Parameter Update

Backpropagation:

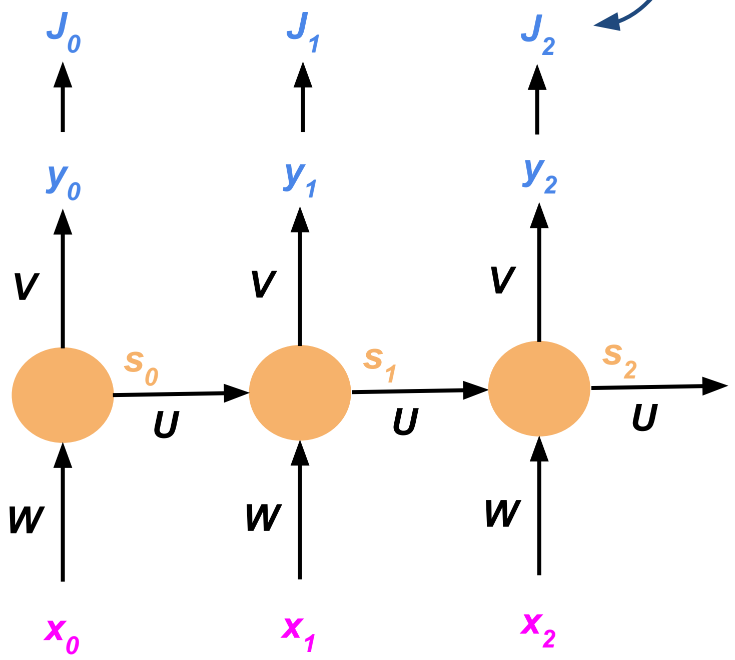
- **Starting with the loss, going backwards through the net**
- **Take partial derivatives with respect to all parameters**
- **Apply the chain rule for partial derivatives**

Parameter Update:

- **Modify the value of every parameter in direction opposite to the gradient raise**

RNNs: Incorporates an additional component of partial derivative chain propagation through time.

RNNs: Backpropagation



Unfolded RNN
There is loss J_i at each time step

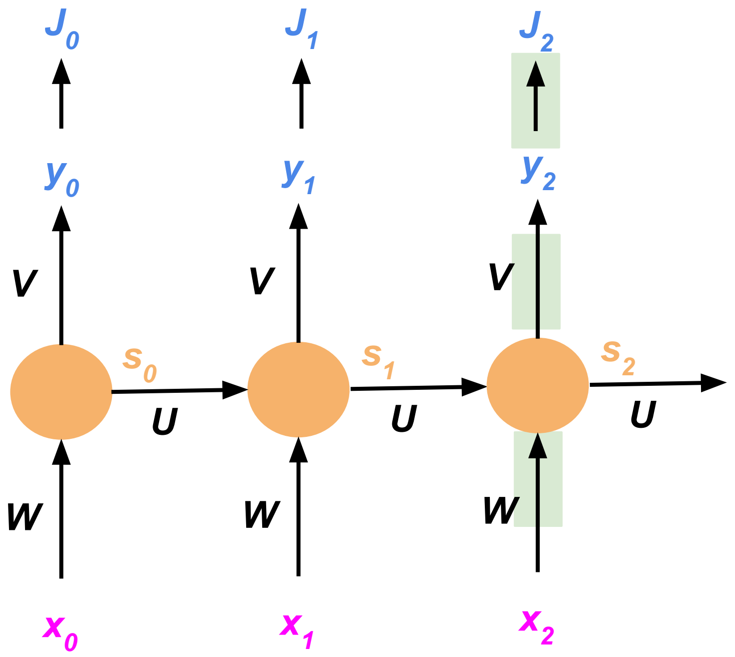
Sum all loss to get total loss

$$J(\Theta) = \sum_t J_t(\Theta)$$

Summing the gradients

$$\frac{\partial J}{\partial P} = \sum_t \frac{\partial J_t}{\partial P}$$

RNNs: Backpropagation



Example for parameter W

$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

For step 1

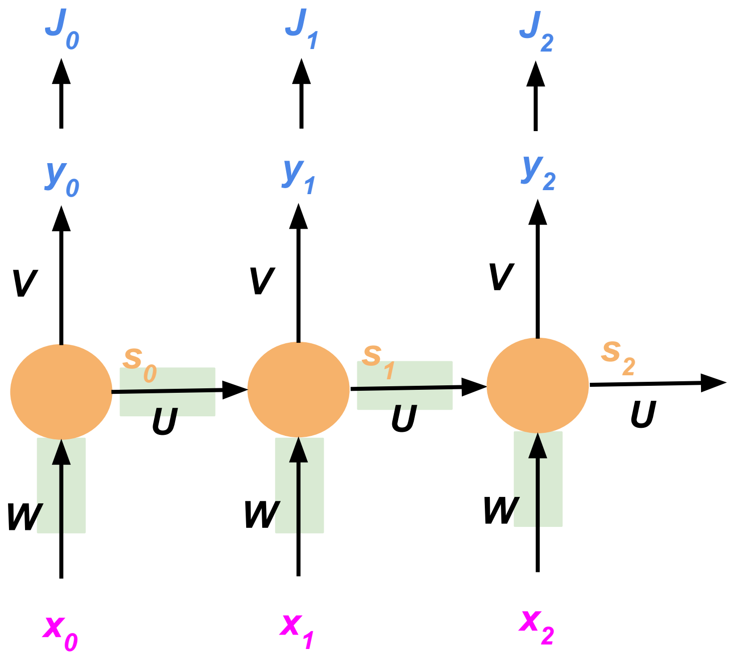
$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial W}$$

But s_2 depends on s_1 as well

$$s_2 = tanh(Us_1 + Wx_2)$$

And s_1 depends on W

RNNs: Backpropagation



S_2 dependencies:

$$\frac{\partial s_2}{\partial W} = \frac{\partial s_2}{\partial W} + \frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W} + \frac{\partial s_2}{\partial s_0}\frac{\partial s_0}{\partial W}$$

RNNs: Backpropagation

Contribution of W in previous time steps to the error in time step 2

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^{2} \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \boxed{\frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}}$$

Generalized to any time step

$$\frac{\partial J_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$$

RNNs are Hard to Train: Vanishing Gradient

$$\frac{\partial J_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\boxed{\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \frac{\partial s_{n-2}}{\partial s_{n-3}} \cdots \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}}$$

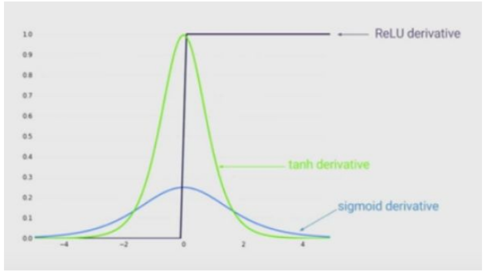
The product gets longer and longer

$$\frac{\partial s_n}{\partial s_{n-1}} = W^T diag[f'(W_{s_{j-1}} + Ux_j)],$$

- ***where W is randomly sampled from standard distribution (<1)***
- ***where f is tanh, and f' is <1***

→ Many small numbers multiply!

VANISHING GRADIENTS: OLD VALUES GET NO CONSIDERATION



The Challenge of Long-Term Dependencies

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

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

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

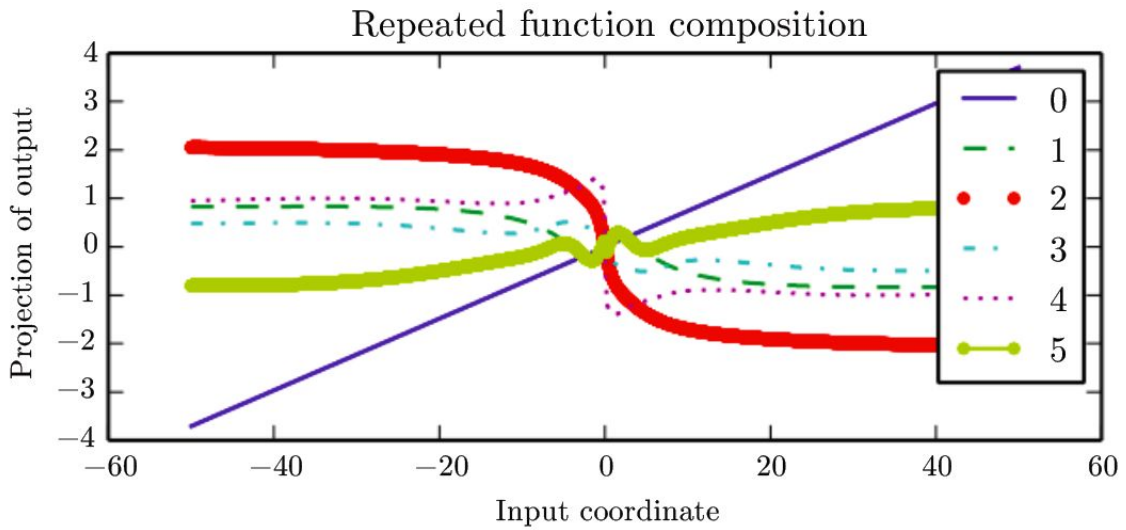
$$\boldsymbol{h}^t = (\mathbf{W}^t)^T \boldsymbol{h}^0,$$

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

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

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

The Challenge of Long-Term Dependencies



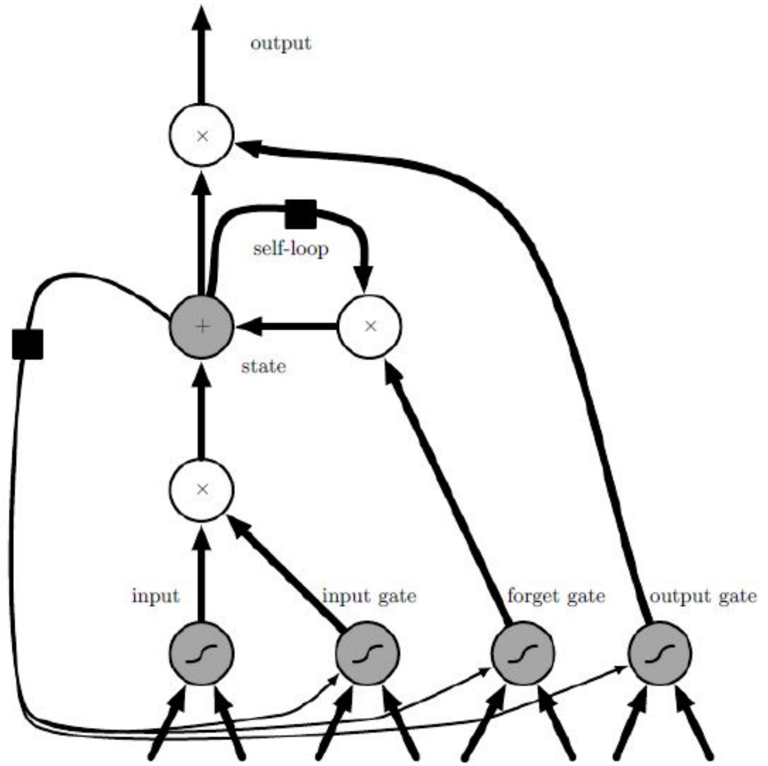
A linear projection of a 100-dimensional hidden state down to a single dimension, plotted on the y-axis. The x-axis is the coordinate of the initial state along a random direction in the 100-dimensional space. We can thus view this plot as a linear cross-section of a high-dimensional function. The plots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

The Long Short-Term Memory Networks (LSTM)

LSTM recurrent networks have LSTM cells that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN.

Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information.

The Long Short-Term Memory Networks (LSTM)



LSTM recurrent network cell. The gradient can flow for a long time within the self-loop.

Cells are connected recurrently to each other, replacing the hidden units of ordinary RNN.

Input feature is computed with a regular artificial neuron. Its value is accumulated into the state if the sigmoidal input gate allows it.

The state unit has a linear self-loop whose weight is controlled by the forget gate.

The output of the cell can be shut off by the output gate.

All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity.

The state unit can also be used as an extra input to the gating units.

The Long Short-Term Memory Networks (LSTM)

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

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

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

The Long Short-Term Memory Networks (LSTM)

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

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

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

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

The Long Short-Term Memory Networks (LSTM)

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

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

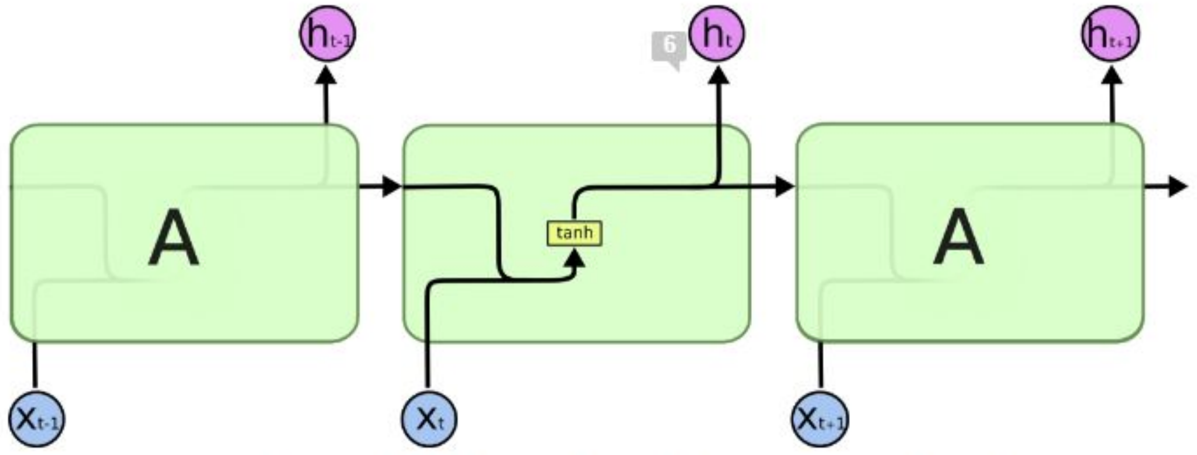
which has parameters $\mathbf{b}^o$, $\mathbf{U}^o$, $\mathbf{W}^o$ for its biases, input weights and recurrent weights, respectively.

The Long Short-Term Memory Networks (LSTM)

LSTM networks have been shown to learn long-term dependencies more easily than the simple recurrent architectures

- first on artificial data sets designed for testing the ability to learn long-term dependencies,
- then on challenging sequence processing tasks where state-of-the-art performance was obtained.

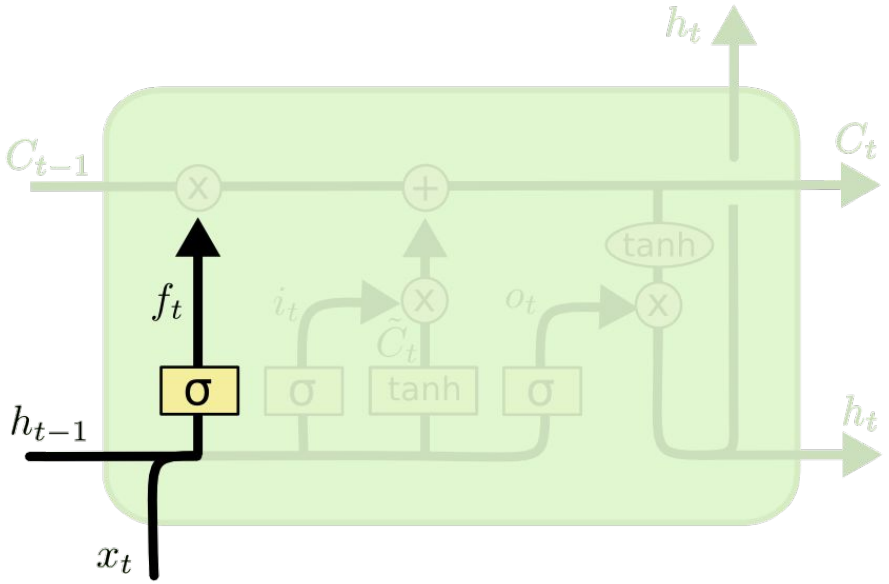
RNN



The repeating module in a standard RNN contains a single layer.

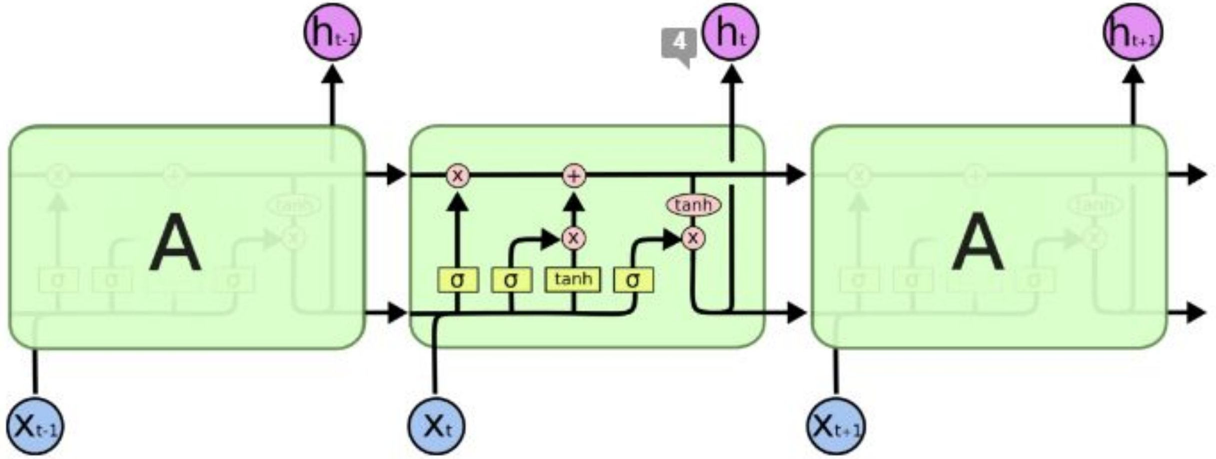
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM

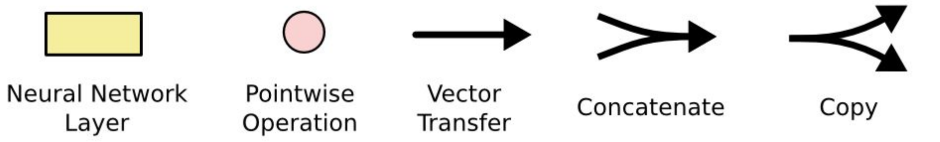


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

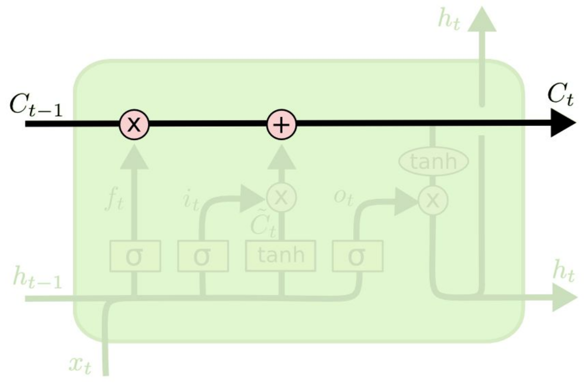
RNN with LSTM



The repeating module in an LSTM contains four interacting layers.



RNN with LSTM



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

LSTM Formulas

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

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

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

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

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

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTMs and Gradient Behavior

LSTMs help preserve the error that can be back-propagated through time and layers.

LSTM contains self-loops which can produce paths where the gradient can flow for long durations.

The weight imposed on this self-loop is conditioned on the context, rather than fixed.

By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically.

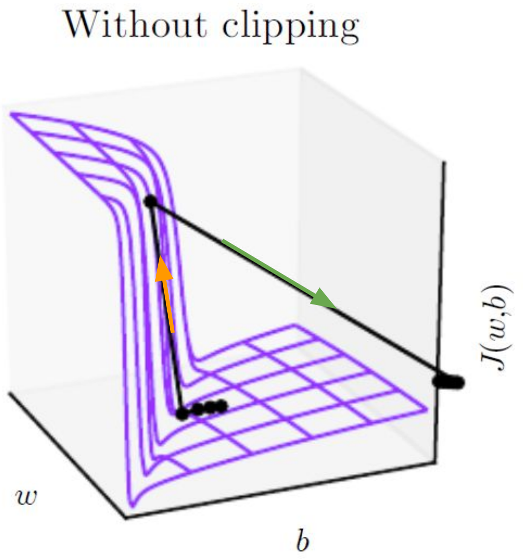
LSTMs and Gradient Behavior

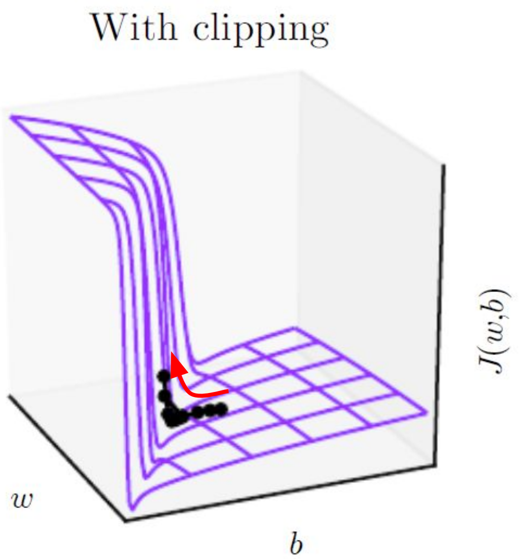
It's important to note that LSTMs' memory cells give different roles to addition and multiplication in the transformation of input.

The central plus sign is essentially the secret of LSTMs. Stupidly simple as it may seem, this basic change helps them preserve a constant error when it must be backpropagated at depth.

Instead of determining the subsequent cell state by multiplying its current state with new input, they add the two, and that quite literally makes the difference. (The forget gate still relies on multiplication, of course.)

Gradient Clipping in Recurrent Networks





(Left) Gradient descent clipping overshoots the bottom of small ravine, then receives a very large gradient from the cliff face. The large gradient propels the parameters outside the axes of the plot.

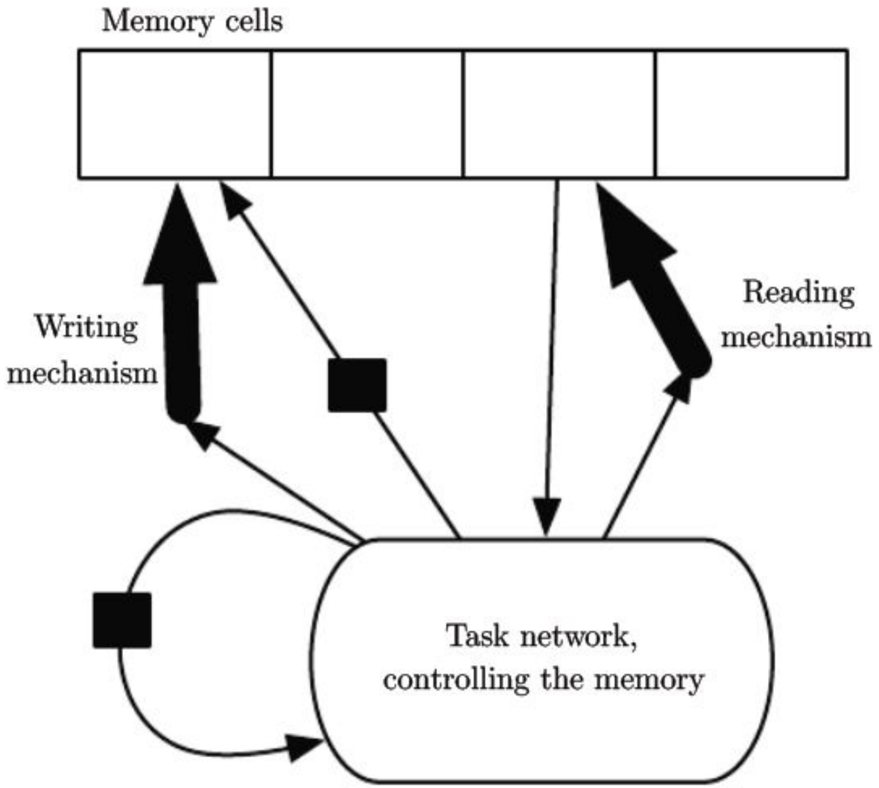
(Right) Gradient descent clipping has a moderate reaction to the cliff. While it ascends the cliff face, the step size is restricted so that it cannot be propelled away from steep region near the solution.

Explicit Memory

An example of a network with an explicit memory, capturing some of the key design elements of the Neural Turing Machine.

Here, we distinguish the representation part of the model (the task network, here a recurrent net at the bottom) from the memory part of the model (the set of cells), which can store facts.

The task network learns to control the memory, deciding where to read from and where to write to within the memory (through the reading and writing mechanisms, indicated by bold arrows pointing at the reading and writing addresses).



Backup Slides

Various

For vanishing, choose higher valued derivatives

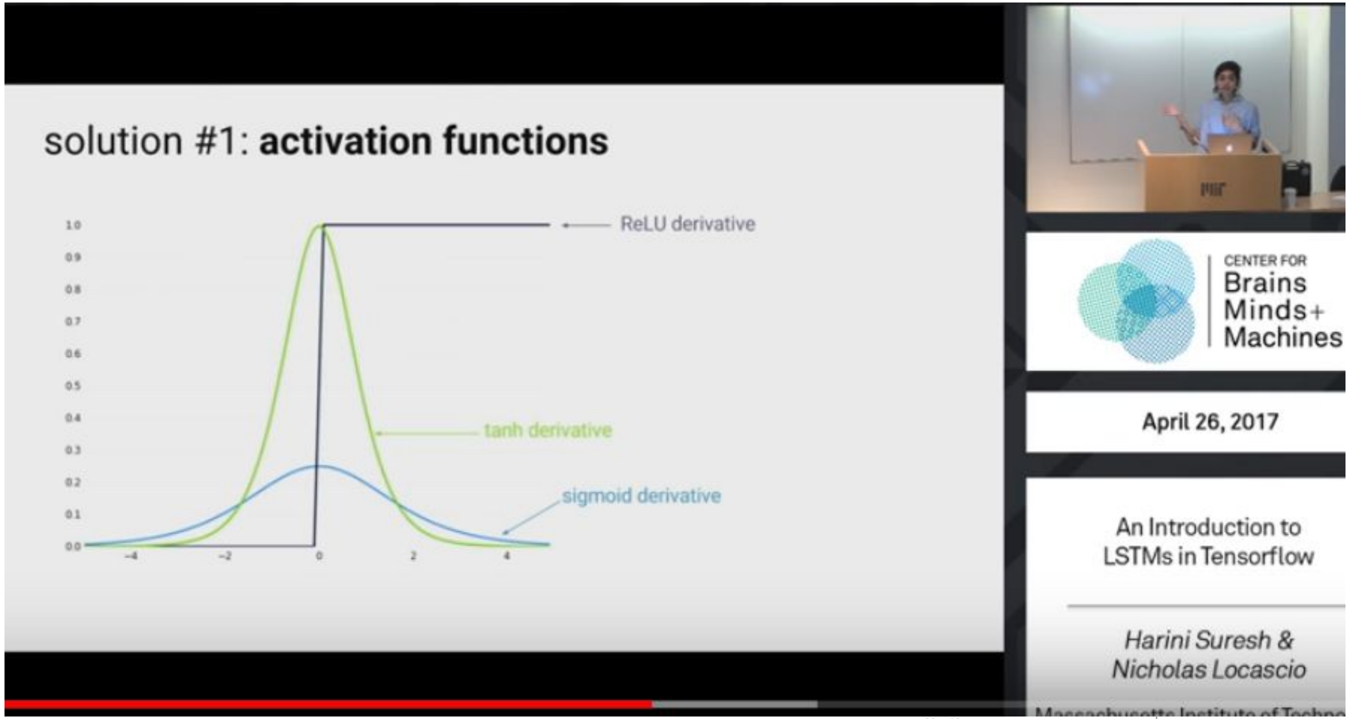
https://www.youtube.com/watch?v=l4X-kZjl1gs

min 28

(1) vanishing

(2) good initilialization

(3)gated cells


solution #1: **activation functions**

April 26, 2017

An Introduction to LSTMs in Tensorflow

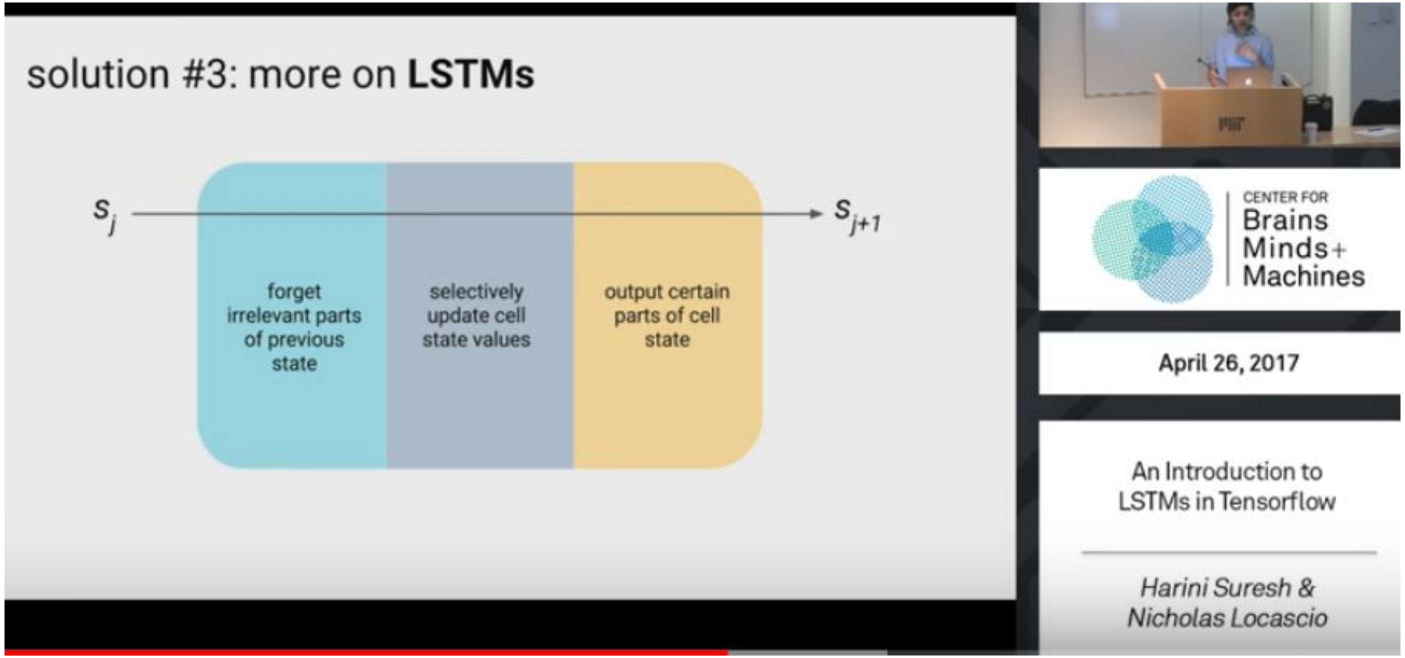
Harini Suresh & Nicholas Locascio

LSTM big picture

https://www.youtube.com/watch?v=l4X-kZjl1gs

min 31

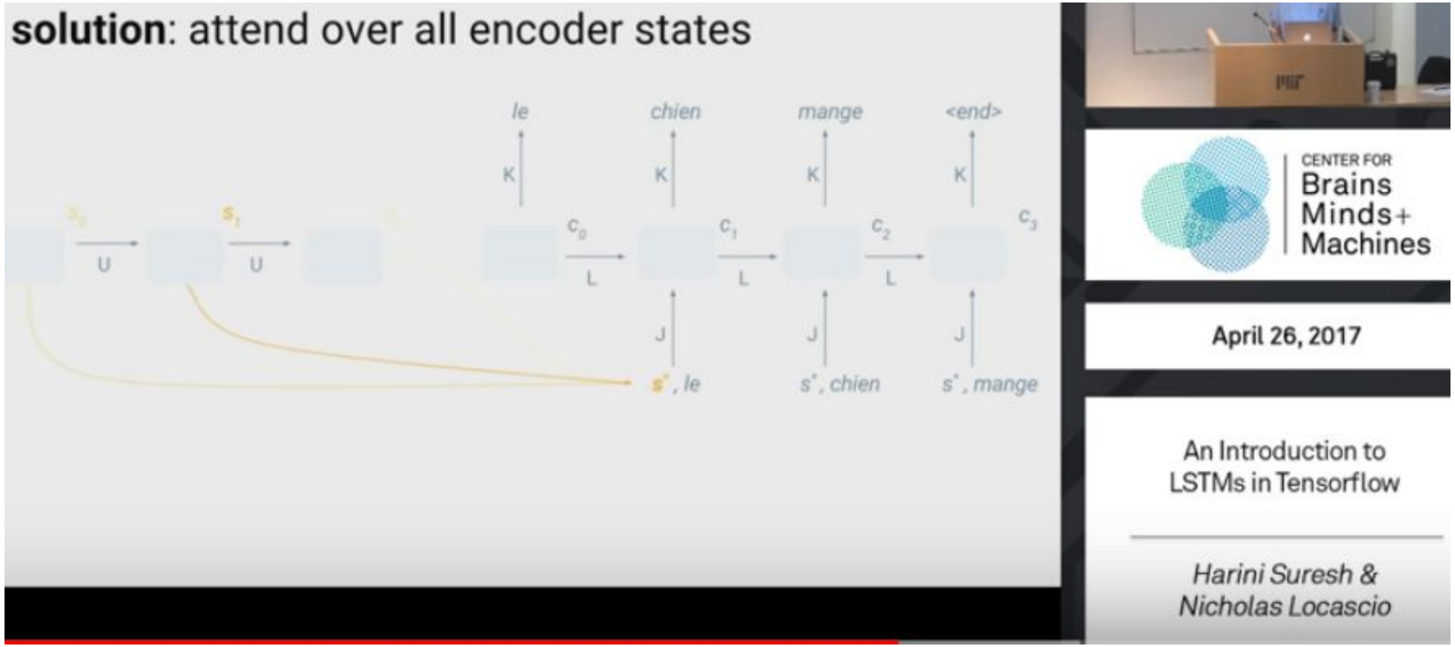
(3)gated cells


solution #3: more on **LSTMs**

s_j
s_{j+1}
forget irrelevant parts of previous state
selectively update cell state values
output certain parts of cell state
CENTER FOR Brains Minds+ Machines
April 26, 2017
An Introduction to LSTMs in Tensorflow
Harini Suresh & Nicholas Locascio

Attention

- https://www.youtube.com/watch?v=l4X-kZjl1gs
- min 36.48
- attention


solution: attend over all encoder states
le
chien
mange
<end>
K
K
K
K
s_0
s_1
s_2
U
U
c_0
c_1
c_2
c_3
L
L
L
J
J
J
s^*, le
s^*, chien
s^*, mange
CENTER FOR Brains Minds+ Machines
April 26, 2017
An Introduction to LSTMs in Tensorflow
Harini Suresh & Nicholas Locascio

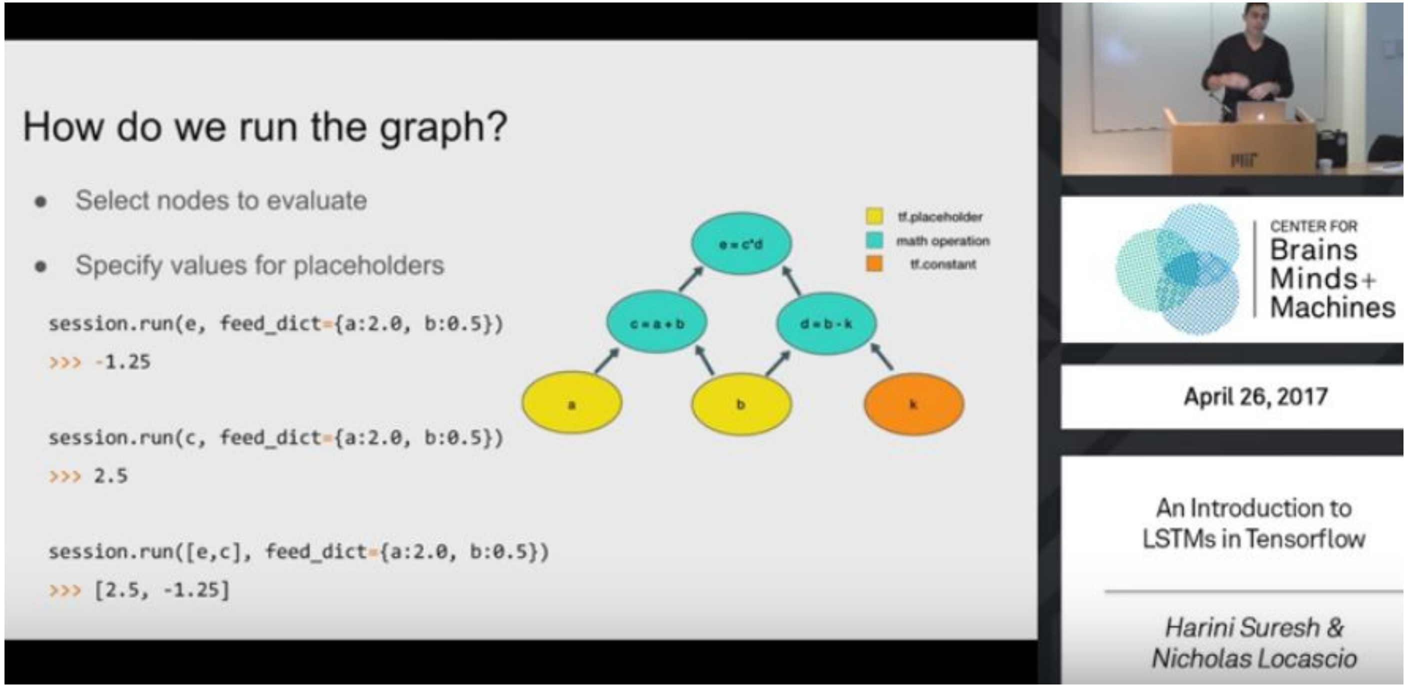
Tensorflow - running the graph

https://www.youtube.com/watch?v=l4X-kZjl1gs

min 41:25

tensorflow summary

min 42:40



How do we run the graph?

- Select nodes to evaluate
- Specify values for placeholders

```
session.run(e, feed_dict={a:2.0, b:0.5})
>>> -1.25

session.run(c, feed_dict={a:2.0, b:0.5})
>>> 2.5

session.run([e,c], feed_dict={a:2.0, b:0.5})
>>> [2.5, -1.25]
```

April 26, 2017

An Introduction to LSTMs in Tensorflow

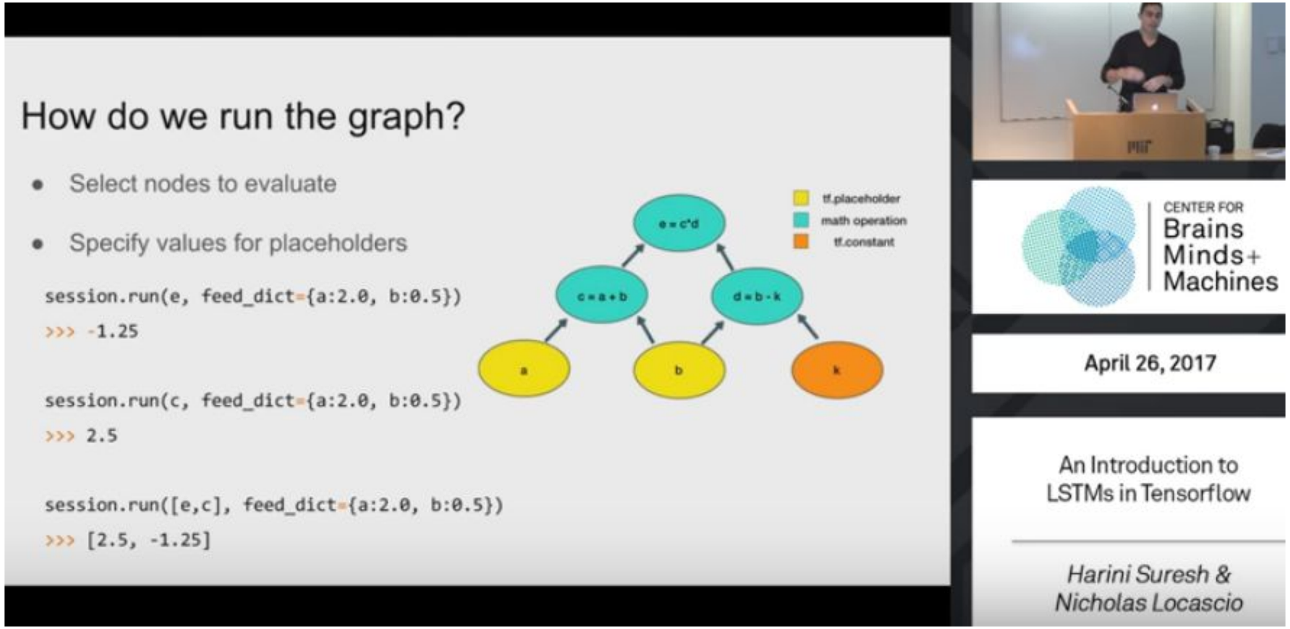
Harini Suresh & Nicholas Locascio

LSTM Tensorflow

https://github.com/nicholaslocascio/bcs-lstm

min 48:42

min 54:31



How do we run the graph?

- Select nodes to evaluate
- Specify values for placeholders

```
session.run(e, feed_dict={a:2.0, b:0.5})
>>> -1.25

session.run(c, feed_dict={a:2.0, b:0.5})
>>> 2.5

session.run([e,c], feed_dict={a:2.0, b:0.5})
>>> [2.5, -1.25]
```

CENTER FOR Brains Minds+ Machines

April 26, 2017

An Introduction to LSTMs in Tensorflow

Harini Suresh & Nicholas Locascio

Representing Sequences, RNN motivation

A bag of words:

This is a good sandwich, not bad at all]

[0 1 0 0 2 0 0 0 2 1]

does not preserve order

- **One more where multiple slots**
- **Markov model**
- **long term dependencies**
- **Goals: maintain word order, share parameters across the sequences, Keep track of long term dependencies**