CS 564: Recurrent Neural Network: Vanilla, LSTM, GRU

Asif Ekbal
CSE Dept.
IIT Patna

Hidden Markov Model (HMM)

- Very elegant method for sequence learning

- **Hidden Mark Model parameters**
  - Emission Probability
  - Transition Probability
  - Initial Probability

- **Three issues of HMM**
  - Evaluation Problem [**Probability of observing a sequence, P(O|M)**]
  - Decoding Problem [**Best possible hidden state Sequence, P(S|O, M**)
  - Learning Problem [**Learning the parameters**]

Getting targets when modeling sequences

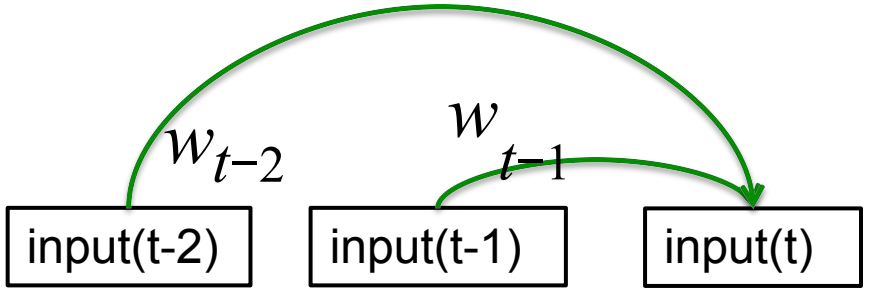
- When applying machine learning to sequences, we often want to turn an input sequence into an output sequence that lives in a different domain
  - *E. g.* turn a sequence of sound pressures into a sequence of word identities.

- When there is no separate target sequence, we can get a teaching signal by trying to predict the next term in the input sequence
  - The target output sequence is the input sequence with an advance of 1 step
  - Seems more natural than trying to predict one pixel in an image from the other pixels, or one patch of an image from the rest of the image
  - For temporal sequences there is a natural order for the predictions

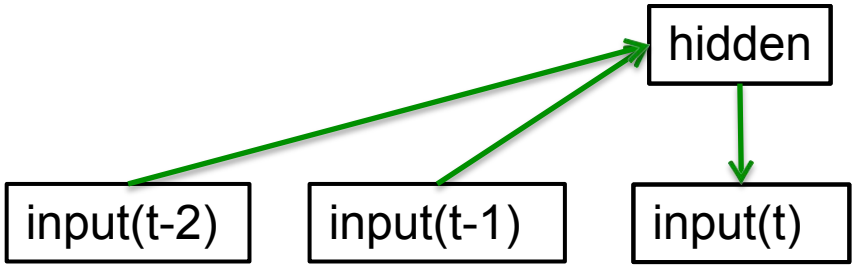
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning
  - It uses the methods designed for supervised learning, but it doesn't require a separate teaching signal

Memoryless models for sequences

- **Autoregressive models**
  Predict the next term in a sequence from a fixed number of previous terms using “delay taps”



- **Feed-forward neural nets**
  These generalize autoregressive models by using one or more layers of non-linear hidden units



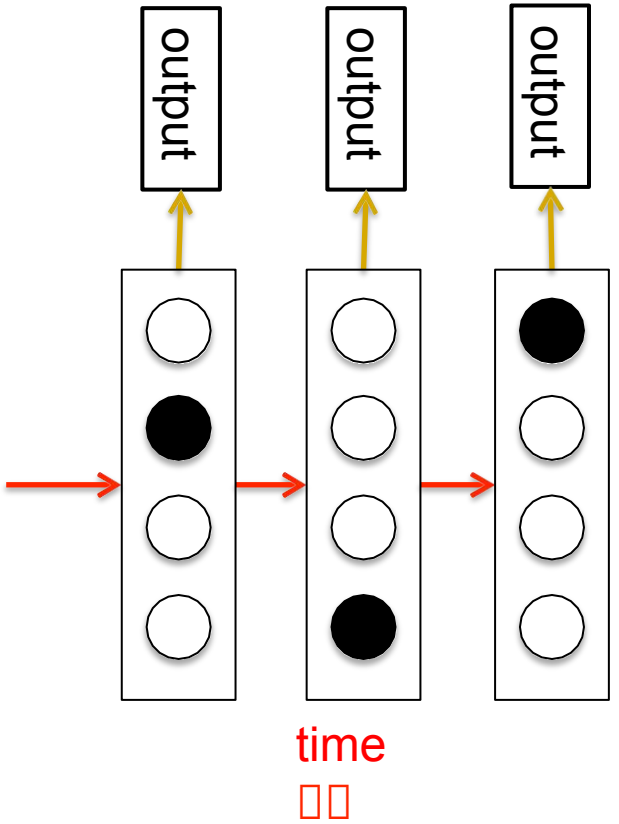
Beyond memoryless models

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model
  - It can store information in its hidden state for a long time
  - If the dynamics is noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state
  - The best we can do is to infer a probability distribution over the space of hidden state vectors

Hidden Markov Models (computer scientists love them!)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic
  - We cannot be sure which state produced a given output. So the state is "hidden"
  - It is easy to represent a probability distribution across N states with N numbers

- To predict the next output we need to infer the probability distribution over hidden states
  - HMMs have efficient algorithms for inference and learning.



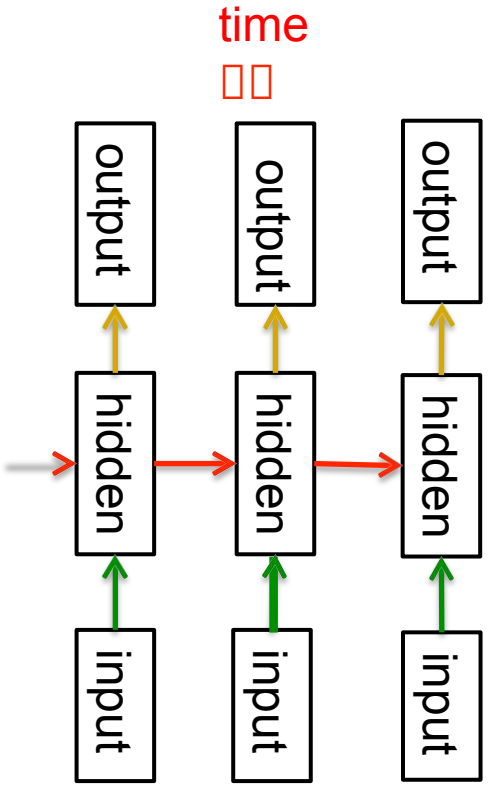
A Fundamental Limitation of HMMs

- Consider what happens when a hidden Markov model generates data
  - At each time step it must select one of its hidden states. So with N hidden states it can only remember log(N) bits about what it generated so far
- Consider the information that the first half of an utterance contains about the second half:
  - The syntax needs to fit (e.g. number and tense agreement).
  - The semantics needs to fit. The intonation needs to fit.
  - The accent, rate, volume, and vocal tract characteristics must all fit.

- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half
  - 2^100 is big!

Recurrent Neural Networks

- RNNs are very powerful, because they combine two properties:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently
  - Nonlinear dynamics that allows them to update their hidden state in complicated ways
- With enough neurons and time, RNNs can compute anything that can be computed by our computer



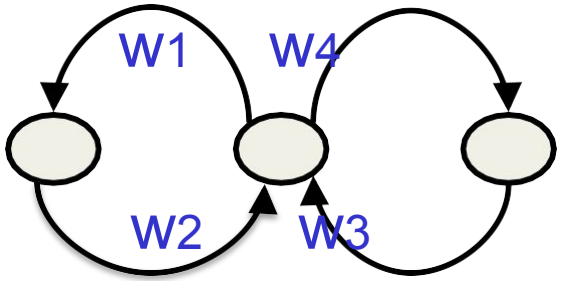
Do Generative Models need to be Stochastic?

- Linear dynamical systems and hidden Markov models are stochastic models
  - But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data
- Recurrent neural networks are deterministic
  - Think hidden state of a RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or hidden Markov model

Recurrent Neural Networks

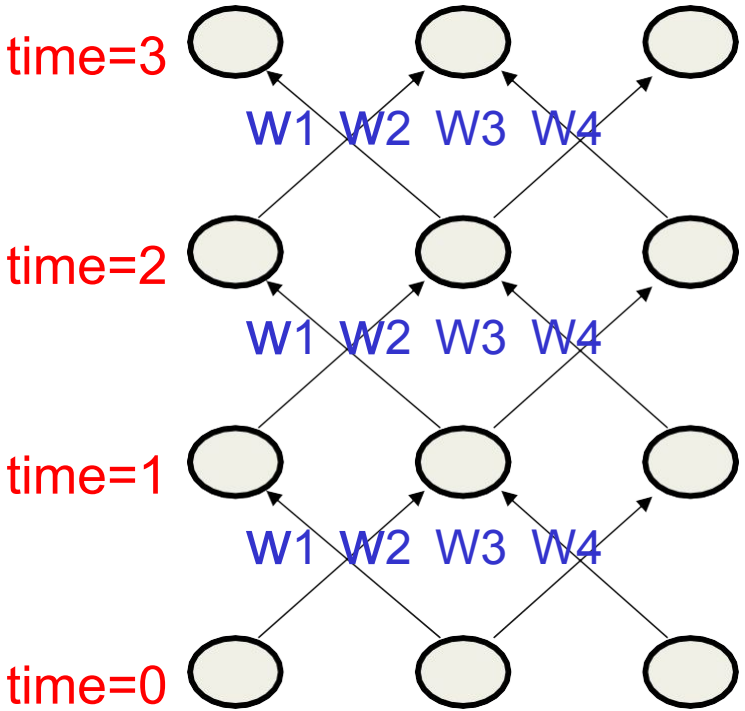
- What kinds of behaviour can RNNs exhibit?
  - They can oscillate. Good for motor control?
  - They can settle to point attractors. Good for retrieving memories?
  - They can behave chaotically. Bad for information processing?
  - RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects
- But the computational power of RNNs makes them very hard to train
  - For many years we could not exploit the computational power of RNNs despite some heroic efforts (e.g. Tony Robinson's speech recognizer).

The equivalence between feedforward nets and recurrent nets



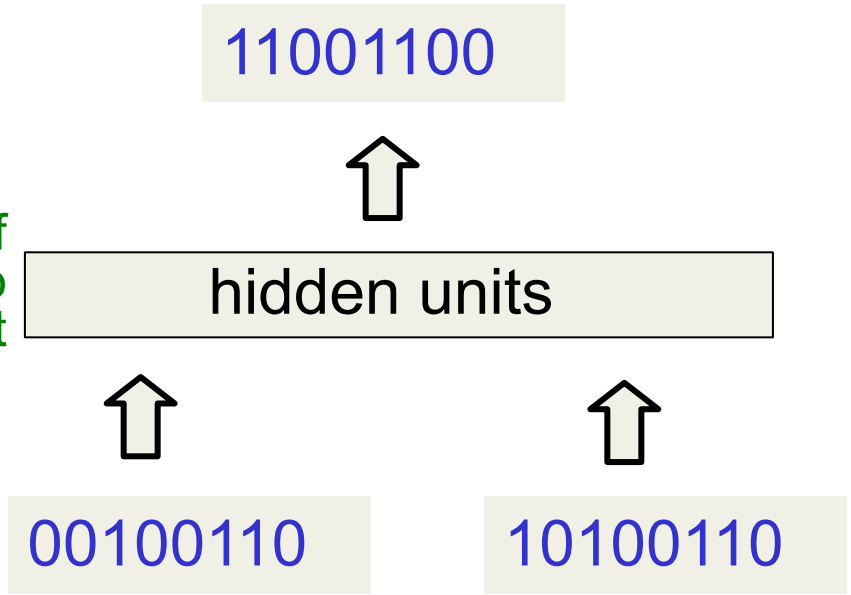
Assume that there is a time delay of 1 in using each connection.

The recurrent net is just a layered net that keeps reusing the same weights.



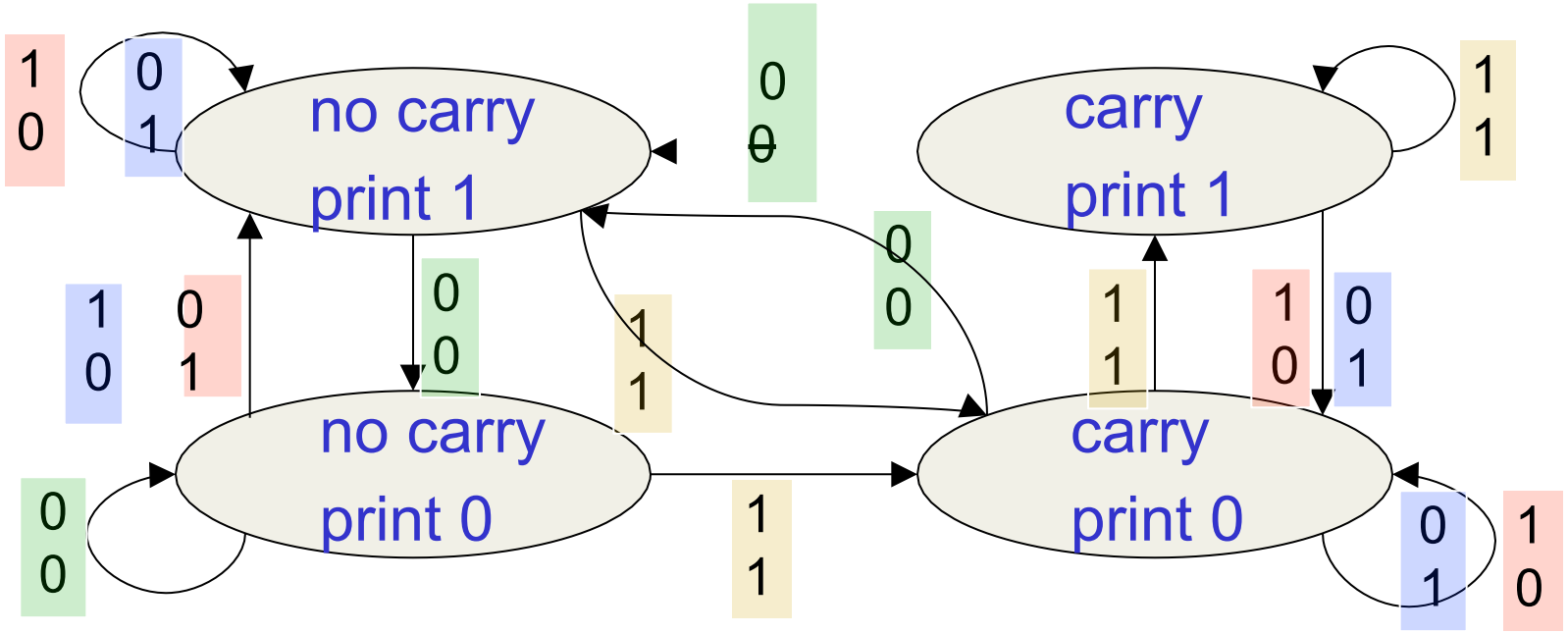
A good toy problem for a recurrent network

- Train a Feedforward NN to perform binary addition?
  - MUST decide in advance the maximum number of digits in each number
  - Processing applied to the beginning of a long number does not generalize to the end of the long number because it uses different weights



- **As a result, Feedforward NN does not generalize well on the binary addition task**

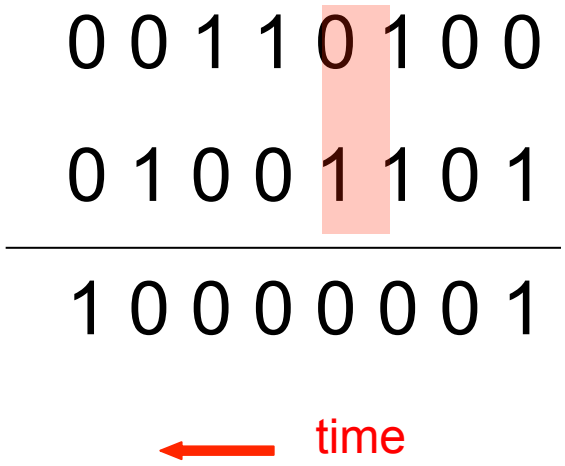
The algorithm for binary addition



This is a finite state automaton. It decides what transition to make by looking at the next column. It prints after making the transition. It moves from right to left over the two input numbers.

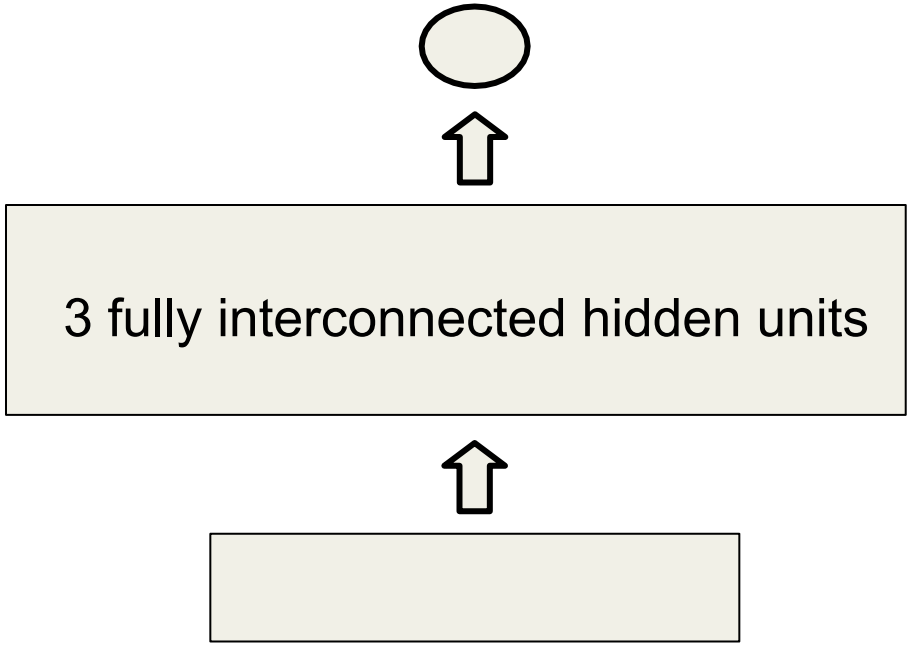
A recurrent net for binary addition

- Network has two input units and one output unit
- Subjected two input digits at each time step
- The desired output at each time step is the output for the column that was provided as input two time steps ago
  - Takes one time step to update the hidden units based on the two input digits
  - Takes another time step for the hidden units to cause the output



The connectivity of the network

- The 3 hidden units are fully interconnected in both directions
  - This allows a hidden activity pattern at one time step to vote for the hidden activity pattern at the next time step
- The input units have feedforward connections that allow then to vote for the next hidden activity pattern



What the network learns?

- Learns four distinct patterns of activity for the 3 hidden units
- Patterns correspond to the nodes in the finite state automaton
  - Do not confuse units in a NN with nodes in a finite state automaton. Nodes are like activity vectors.
  - The automaton is restricted to be in exactly one state at each time. The hidden units are restricted to have exactly one vector of activity at each time

- A recurrent network can emulate a finite state automaton, but it is exponentially more powerful
- With N hidden neurons it has 2^N possible binary activity vectors (but only N^2 weights)
  - This is important when the input stream has two separate things going on at once
  - A finite state automaton needs to square its number of states
  - An RNN needs to double its number of units

Recurrent Neural Network

- An effective neural network model for learning sequences
- Have memory to store information of the past
  - Same weights are shared across the layers
- Can have many configurations
  - One-to-many, many-to-one, many-to-many

- Have shown success in solving many problems such as Sentiment Analysis, Machine Translation, Image Captioning etc.

Recurrent Neural Networks (RNNs)

- RNNs are neural networks, specialized for processing a sequence of values $x^{(1)}, \ldots, x^{(\tau)}$.

Recurrent Neural Networks (RNNs)

- RNNs are neural networks, specialized for processing a sequence of values $x^{(1)}, \ldots, x^{(\tau)}$
- Can scale to much longer sequences that would be practical for networks without sequence-based specialization
- Most networks can process sequences of variable length
  - But does not do so by sharing parameters

Recurrent Neural Networks

We can process a sequence of vectors x by applying a recurrence formula at each step:

Recurrent Neural Networks

We can process a sequence of vectors x by applying a recurrence formula at each step:

$$h_t = f_W(h_{t-1}, x_t)$$

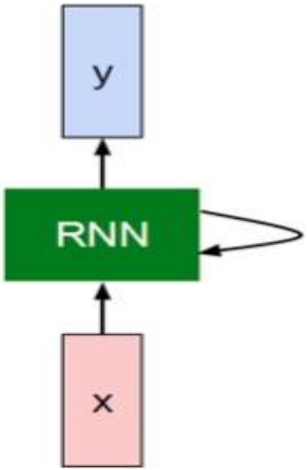
new state

some function with parameters W

old state

input vector at some time step

Notice: the same function and the same set of parameters are used at every time step.



Unfolding Computational Graphs

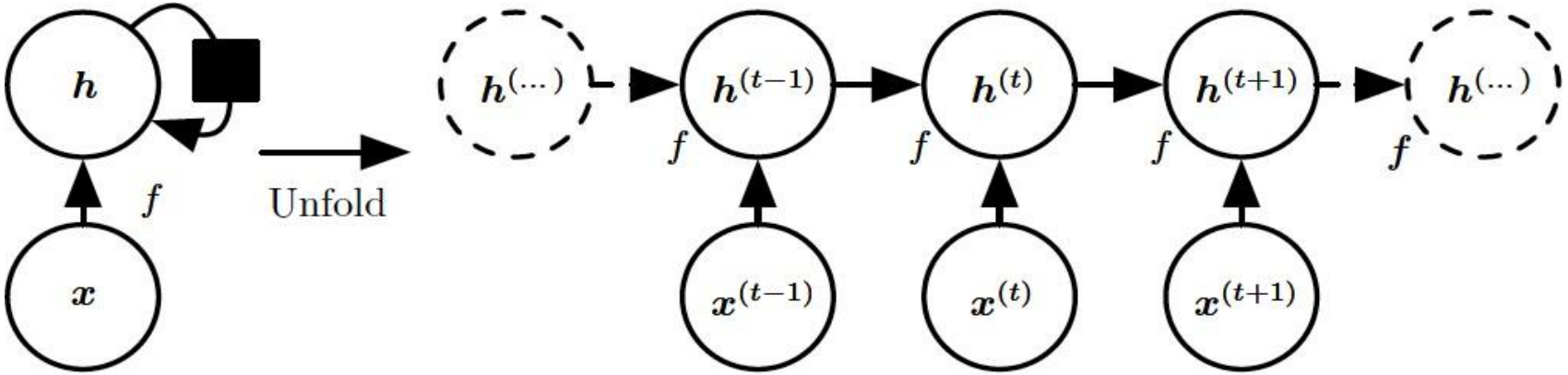


Figure 10.2

Recurrent Neural Networks

- 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

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

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 to learn a separate model $g^{(t)}$ for all possible time steps.

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

Recurrent Hidden Units and Training Loss

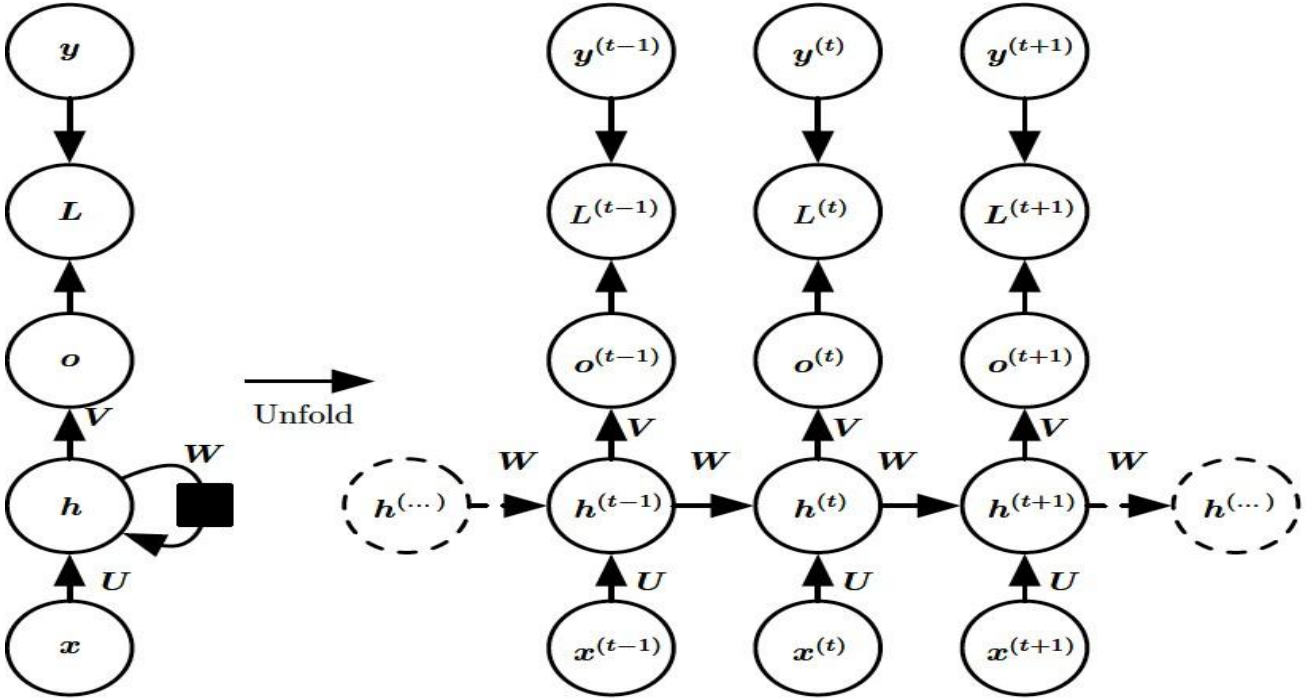


Figure 10.3

Design patterns

Lot of flexibility

RNNs have many design patterns, we will see some of those now

Recurrent connections between hidden units

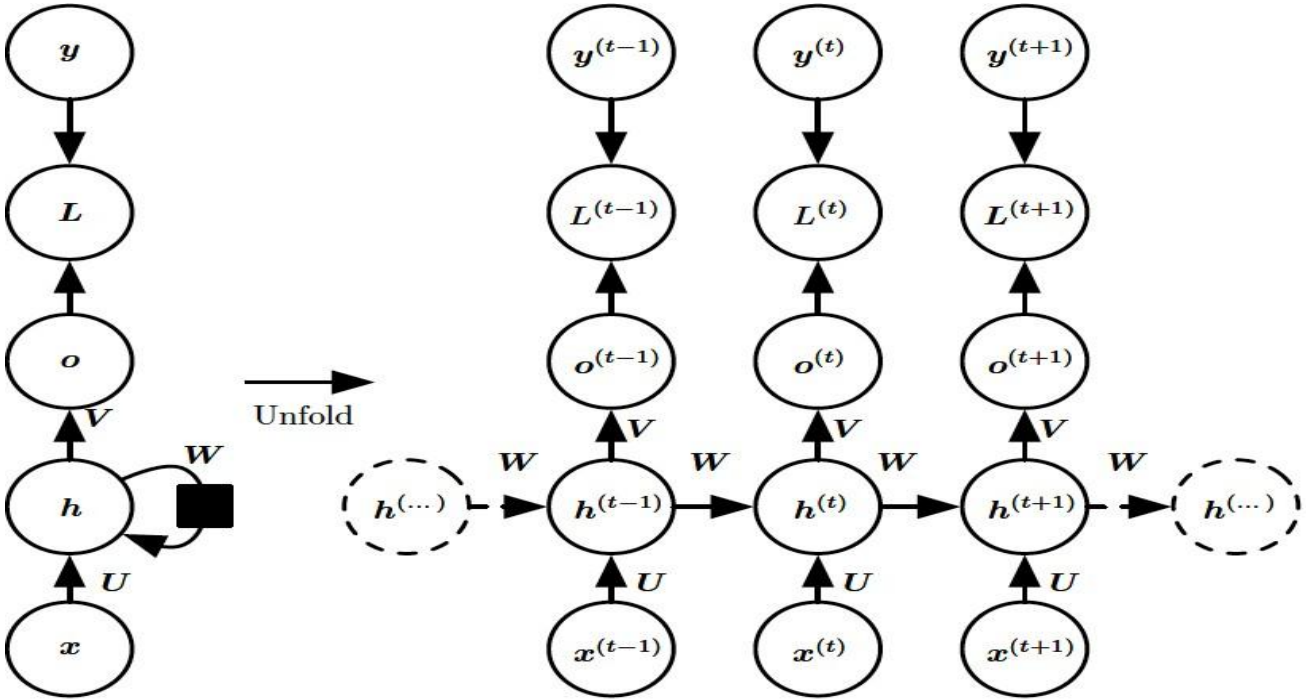


Figure 10.3

Recurrence through the Output Only

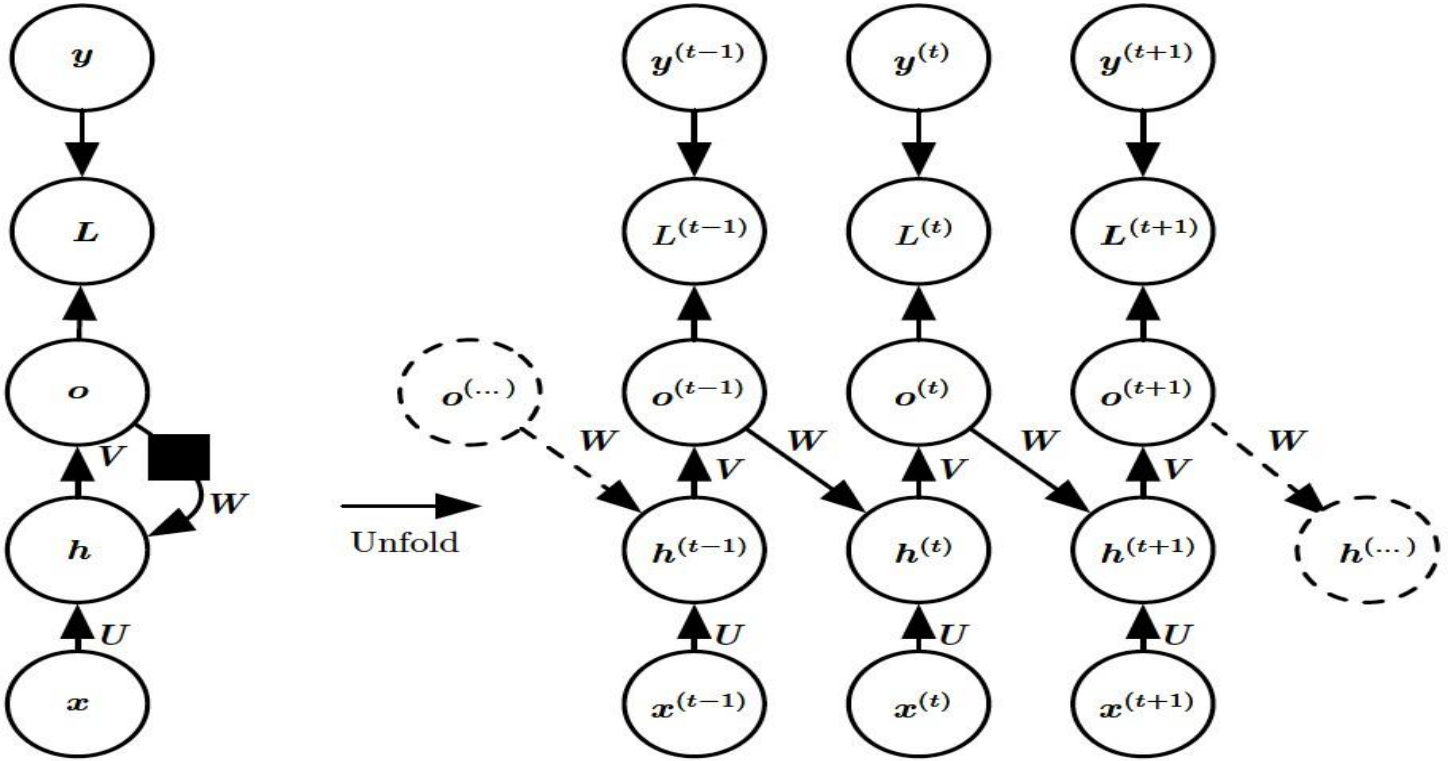


Figure 10.4

Sequence Input, Single Output

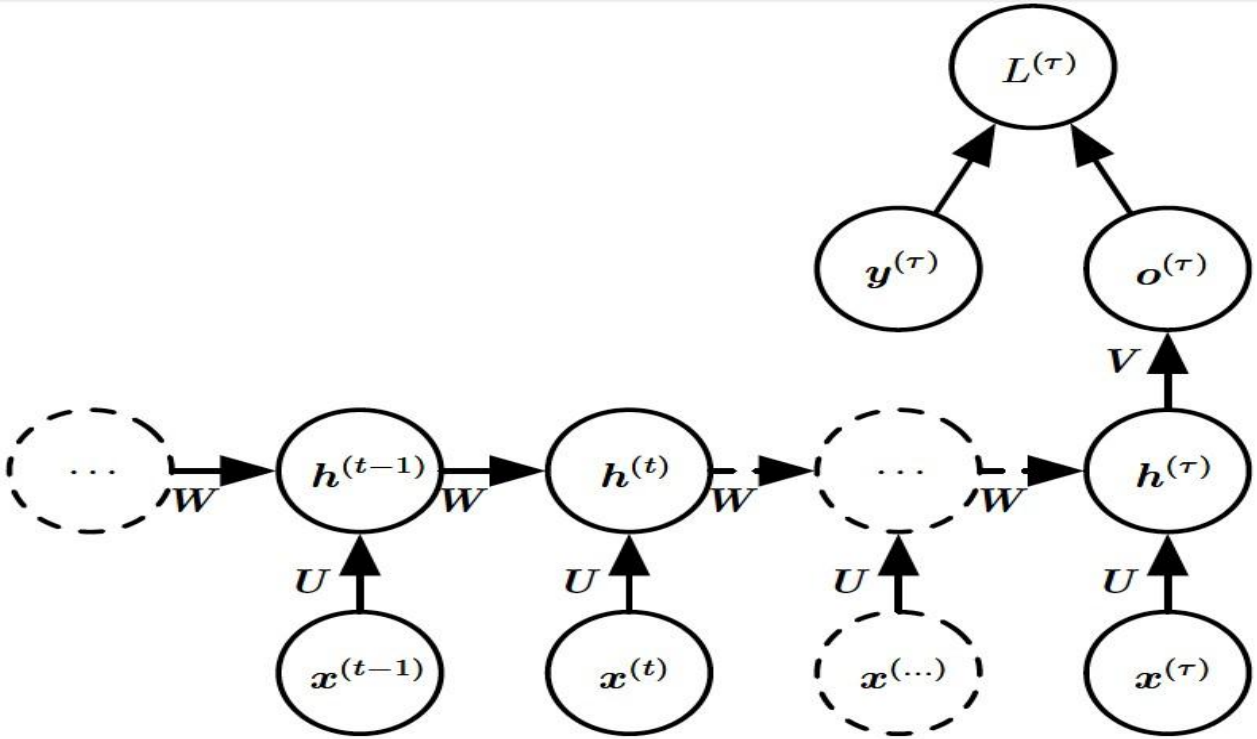


Figure 10.5

Forward propagation for the RNN: first model

Activation function for the hidden units

Assume the hyperbolic tangent activation function

Forward propagation for the RNN: first model

Activation function for the hidden units

Assume the hyperbolic tangent activation function

Form of output and loss function

Assume output is discrete - predicting words or characters

We can obtain a vector normalized probabilities over the output - $\hat{y}$.

Forward propagation for the RNN: first model

Activation function for the hidden units

Assume the hyperbolic tangent activation function

Form of output and loss function

Assume output is discrete - predicting words or characters

We can obtain a vector normalized probabilities over the output - $\hat{y}$.

Update Equations

Initial state - $h^{(0)}$

Forward propagation for the RNN: first model

Activation function for the hidden units

Assume the hyperbolic tangent activation function

Form of output and loss function

Assume output is discrete - predicting words or characters

We can obtain a vector normalized probabilities over the output - $\hat{y}$.

Update Equations

Initial state - $h^{(0)}$

From $t = 1$ to $t = \tau$, the following update equation is applied:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

Forward Propagation

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

$$o^{(t)} = c + Vh^{(t)}$$

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

Forward Propagation

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

$$o^{(t)} = c + Vh^{(t)}$$

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

This maps an input sequence to an output sequence of the same length

Loss Function

Total loss is sum of the losses over all the time steps.
So, if $L^{(t)}$ is the negative log likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(\tau)}$, then

Loss Function

Total loss is sum of the losses over all the time steps

So, if $L^{(t)}$ is the negative log likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(\tau)}$, then

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

Loss Function

Total loss is sum of the losses over all the time steps
So, if $L^{(t)}$ is the negative log likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(\tau)}$, then

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

where $p_{model}(y^{(t)} | \{x^{(1)}, \ldots, x^{(\tau)}\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{y}^{(t)}$

Loss Function

Total loss is sum of the losses over all the time steps.

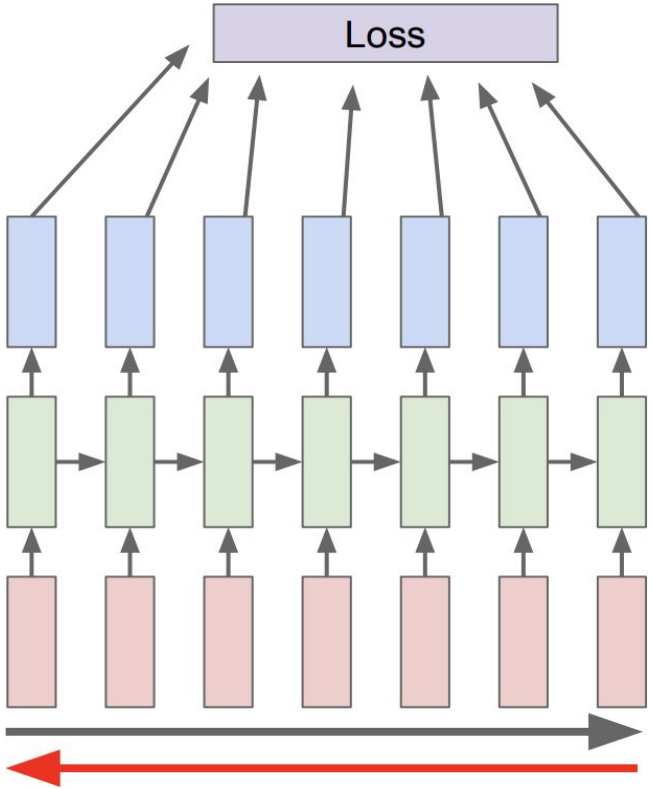
So, if $L^{(t)}$ is the negative log likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(\tau)}$, then

$$L(\{x^{(1)}, \ldots, x^{(\tau)}\}, \{y^{(1)}, \ldots, y^{(\tau)}\}) = \sum_t L^{(t)} = -\sum_t \log p_{model}(y^{(t)} | \{x^{(1)}, \ldots, x^{(\tau)}\})$$

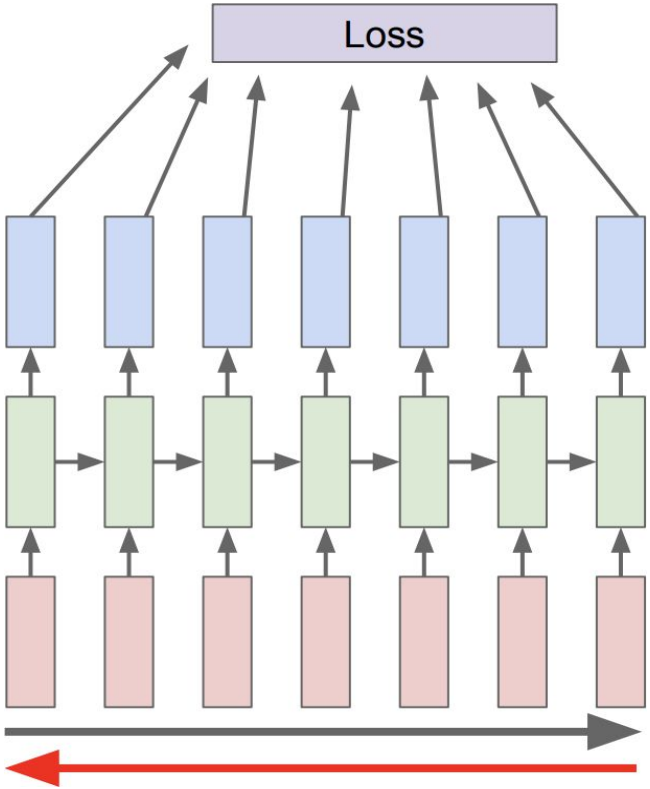
where $p_{model}(y^{(t)} | \{x^{(1)}, \ldots, x^{(\tau)}\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{y}^{(t)}$

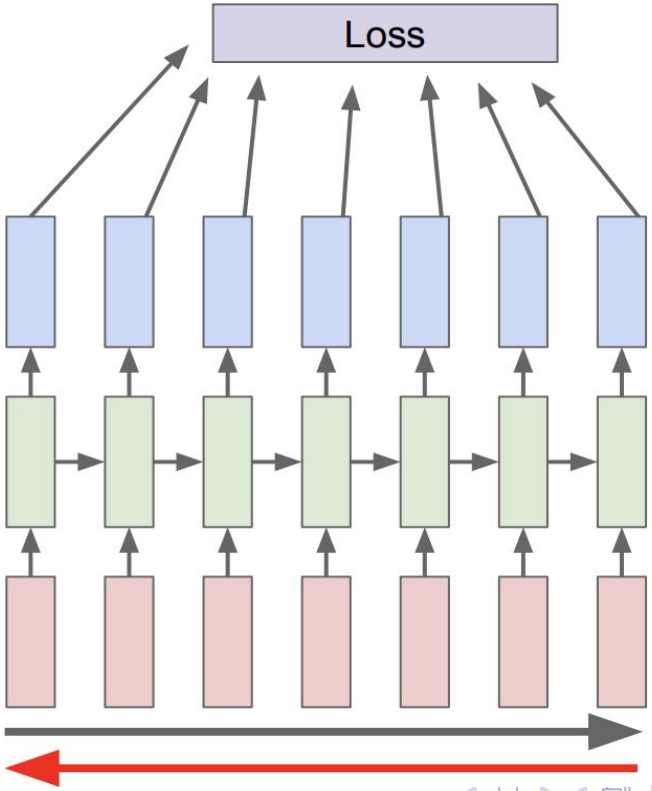
Back propagation - right to left - backpropagation through time (BPTT)

Backpropagation through time (BPTT)



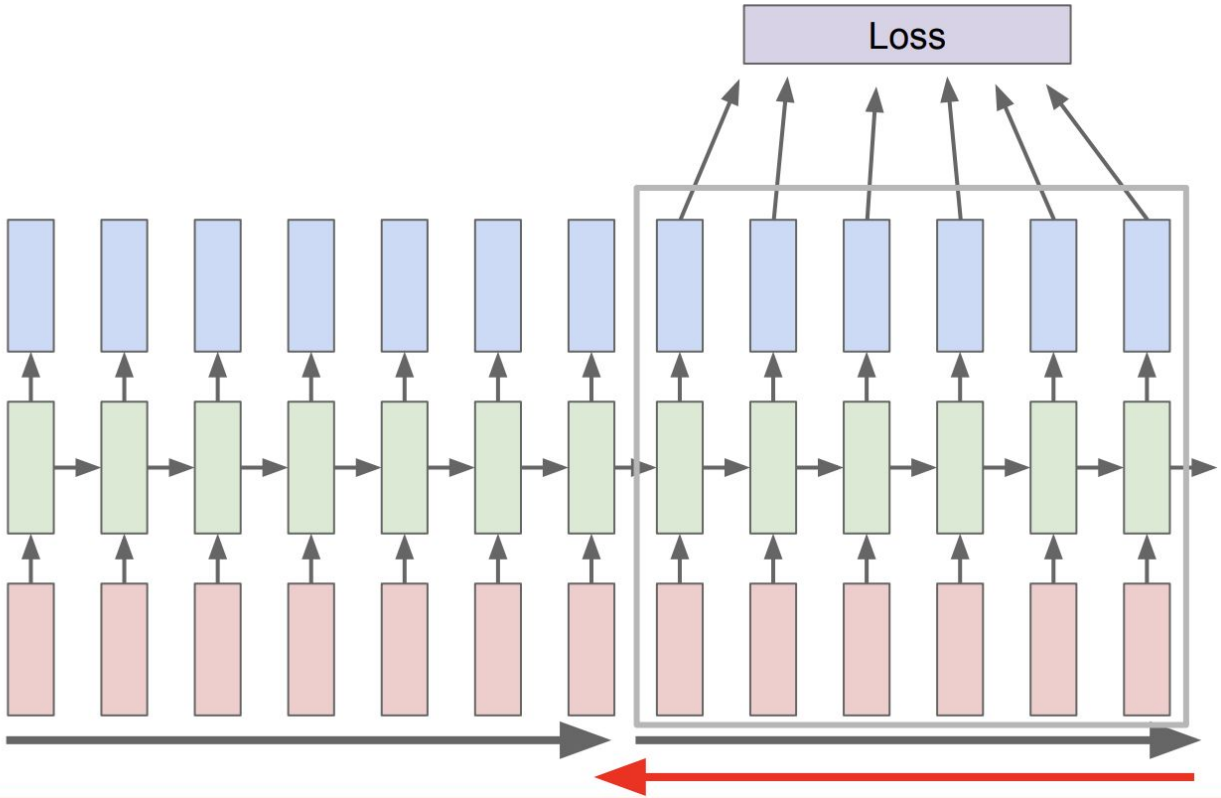
Truncated Backpropagation through time (BPTT)





Process only chunk of sequence and backprop to update W

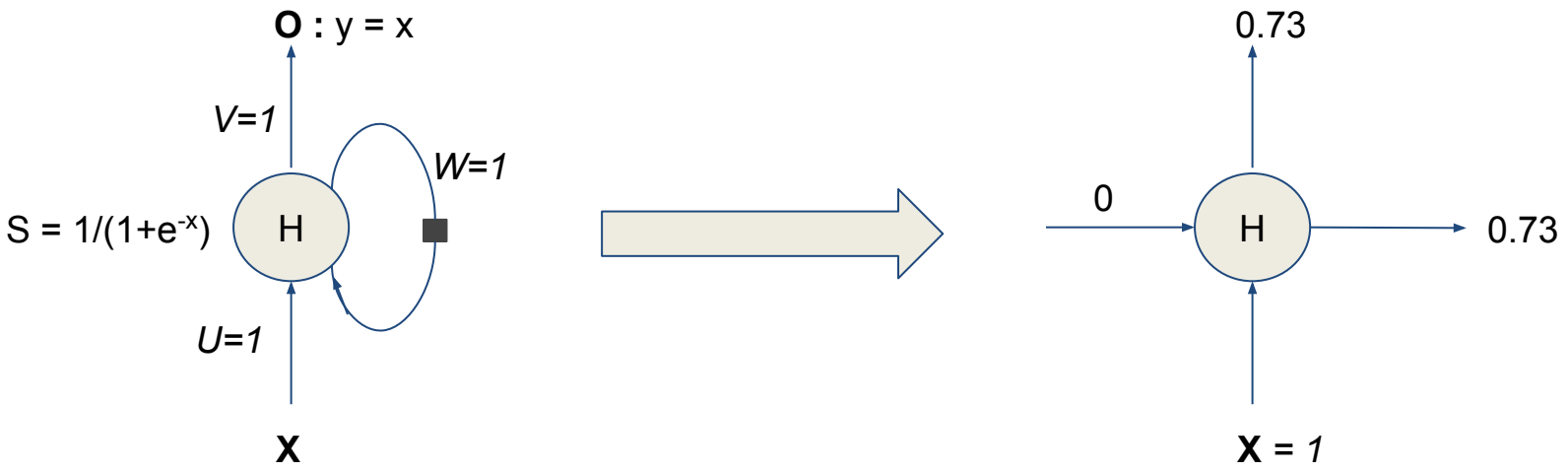
Truncated Backpropagation through time (BPTT)



Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

RNN Sequence Processing Example

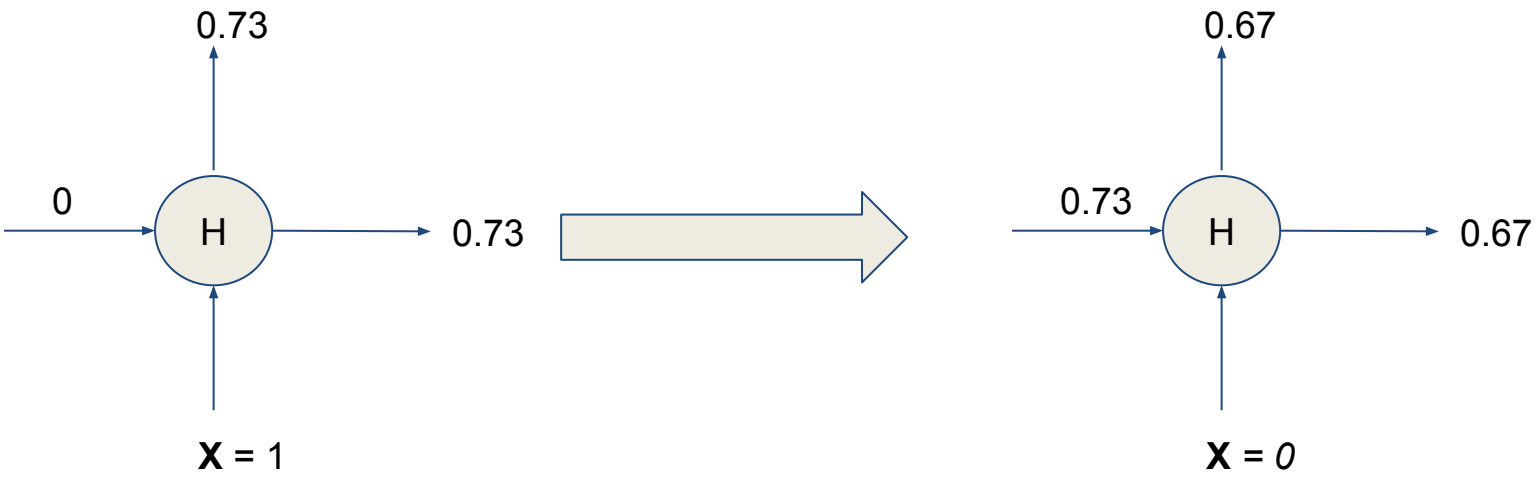
Input Sequence: 1 0 0 0 1 0



T = 1

RNN Sequence Processing Example

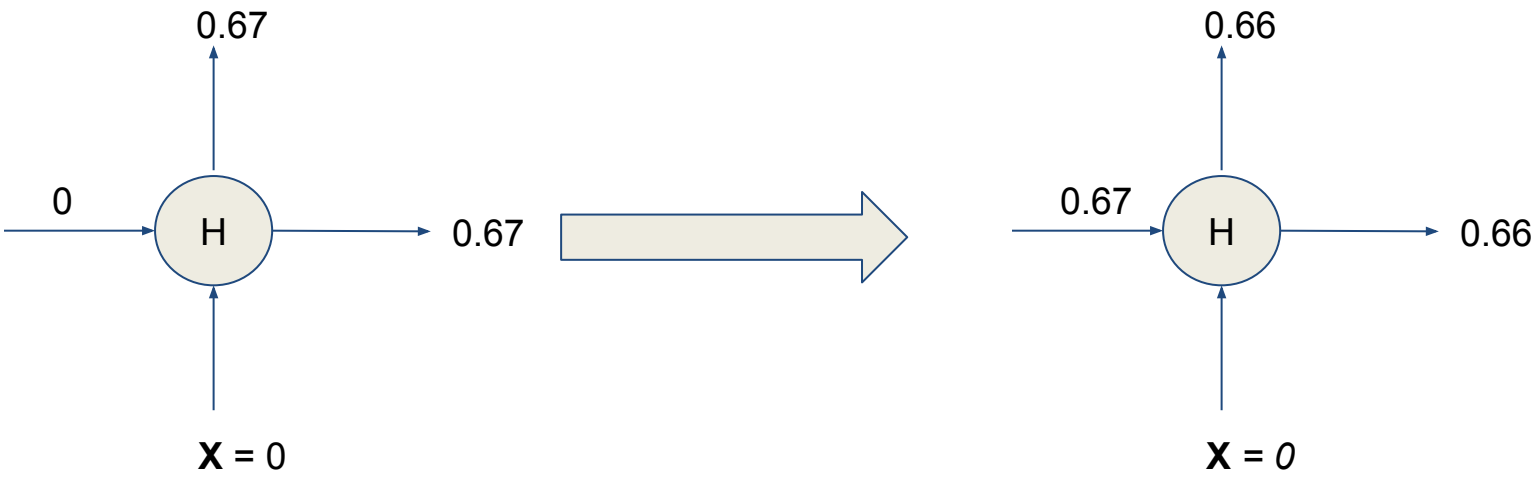
Input Sequence: 1 **0** 0 0 1 0



T = 2

RNN Sequence Processing Example

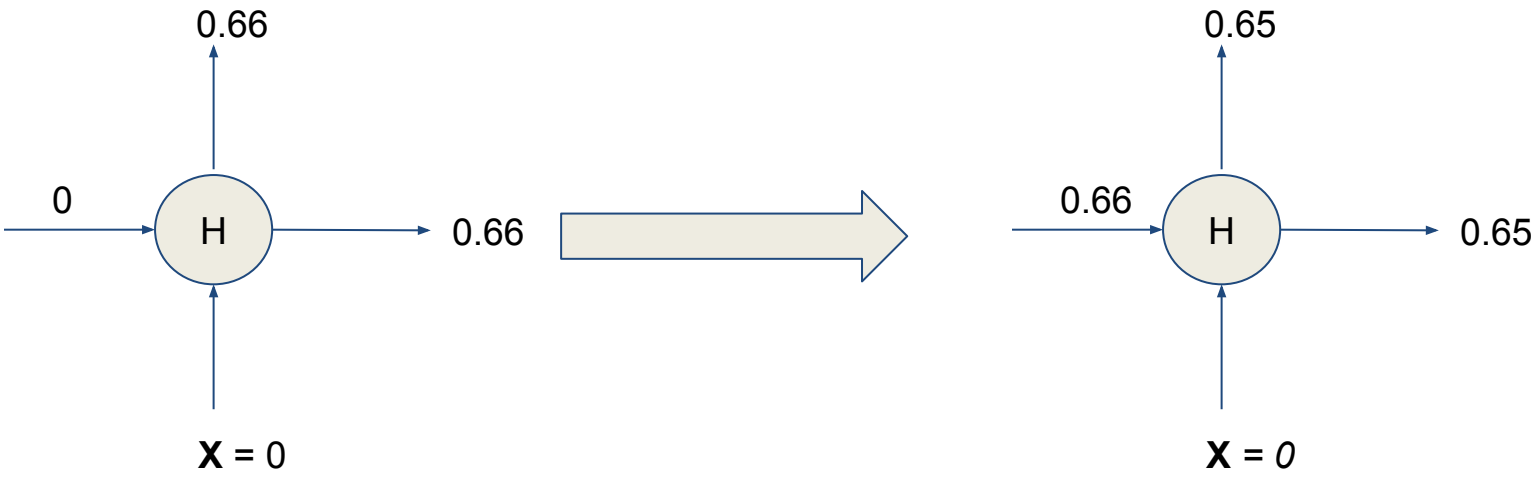
Input Sequence: 1 0 **0** 0 1 0



T = 3

RNN Sequence Processing Example

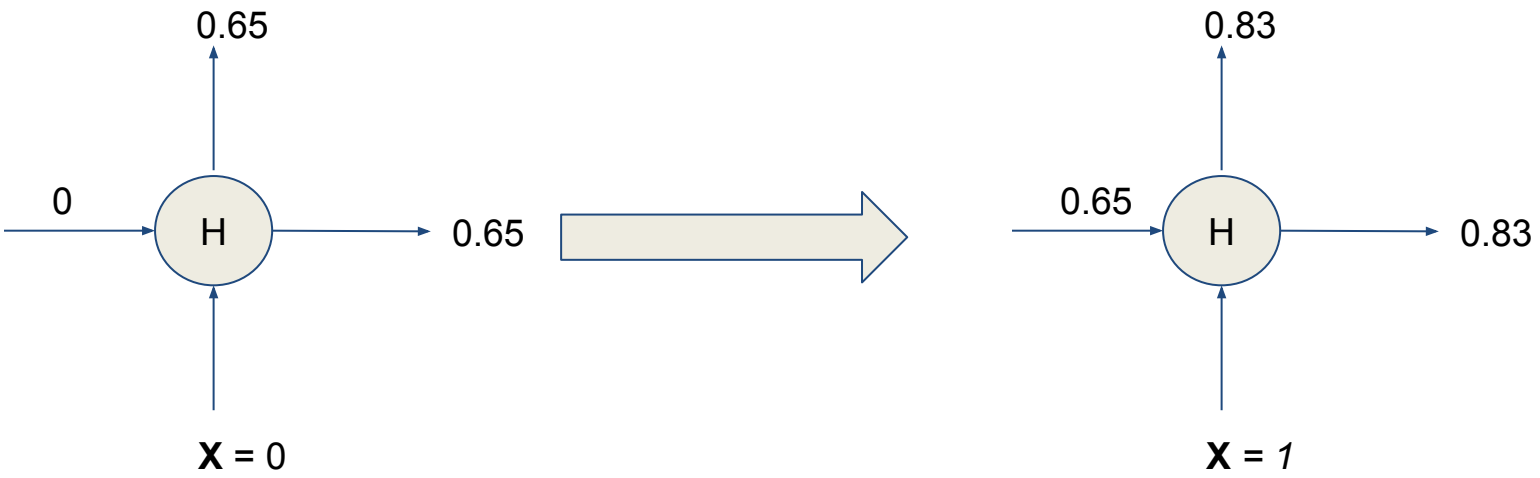
Input Sequence: 1 0 0 **0** 1 0



T = 4

RNN Sequence Processing Example

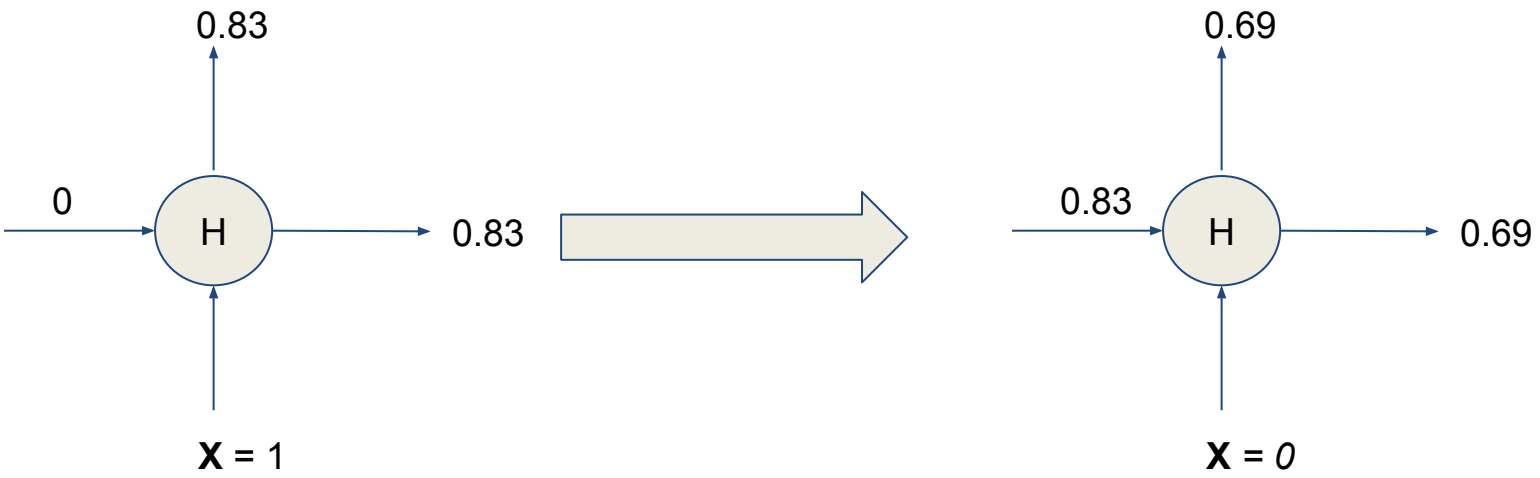
Input Sequence: 1 0 0 0 **1** 0



T = 5

RNN Sequence Processing Example

Input Sequence: 1 0 0 0 1 **0**



T = 6

Networks with Output Recurrence

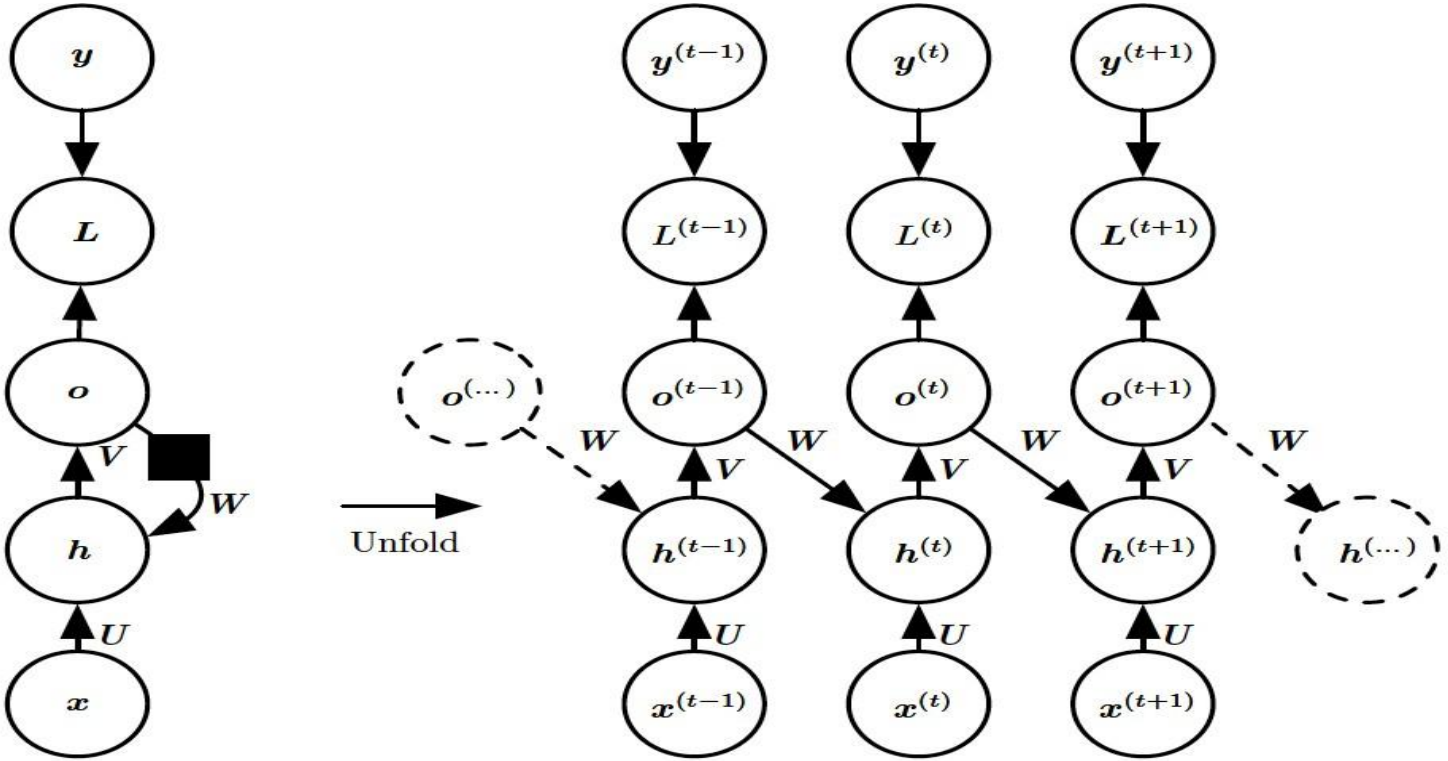


Figure 10.4

Networks with Output Recurrence

Less powerful?

- Lacks hidden-to-hidden recurrence,
- thus requires that the output units capture all the information about the past that the network will use to predict the future

Networks with Output Recurrence

Less powerful?

- Lacks hidden-to-hidden recurrence, thus requires that the output units capture all the information about the past that the network will use to predict the future
- However, the output units are explicitly trained to match the training set targets, and is unlikely to capture the necessary information about the past history of input, unless explicitly provided as training set target

Networks with Output Recurrence

What is the advantage?

- All the time steps are decoupled, and training can be parallelized.
- **Teacher Forcing:**There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

Teacher Forcing

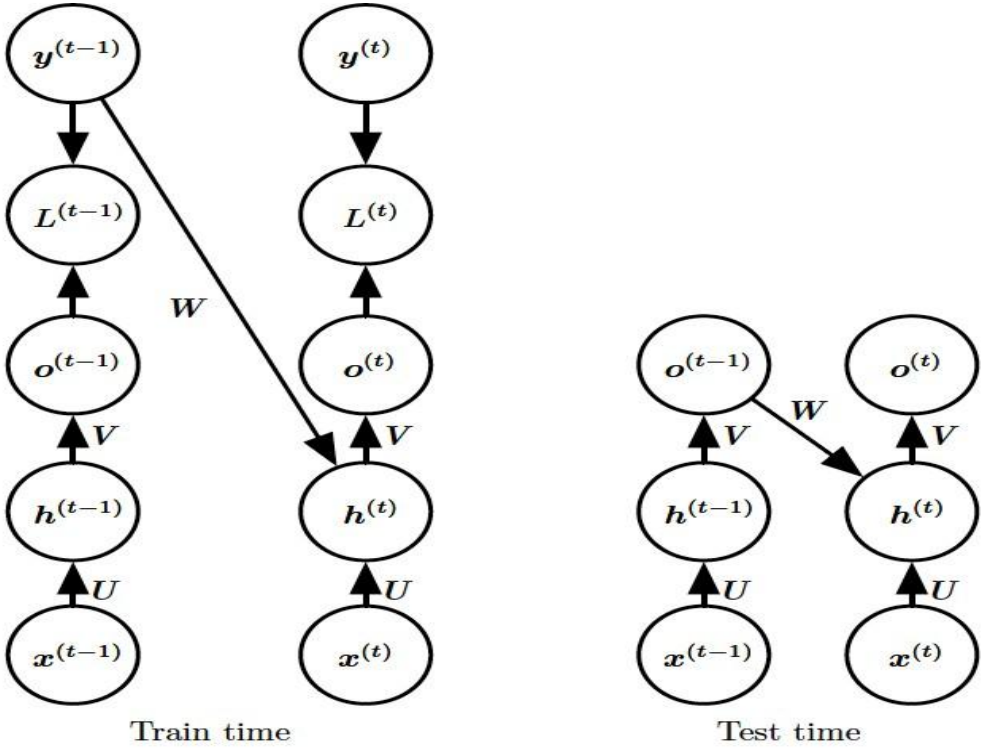
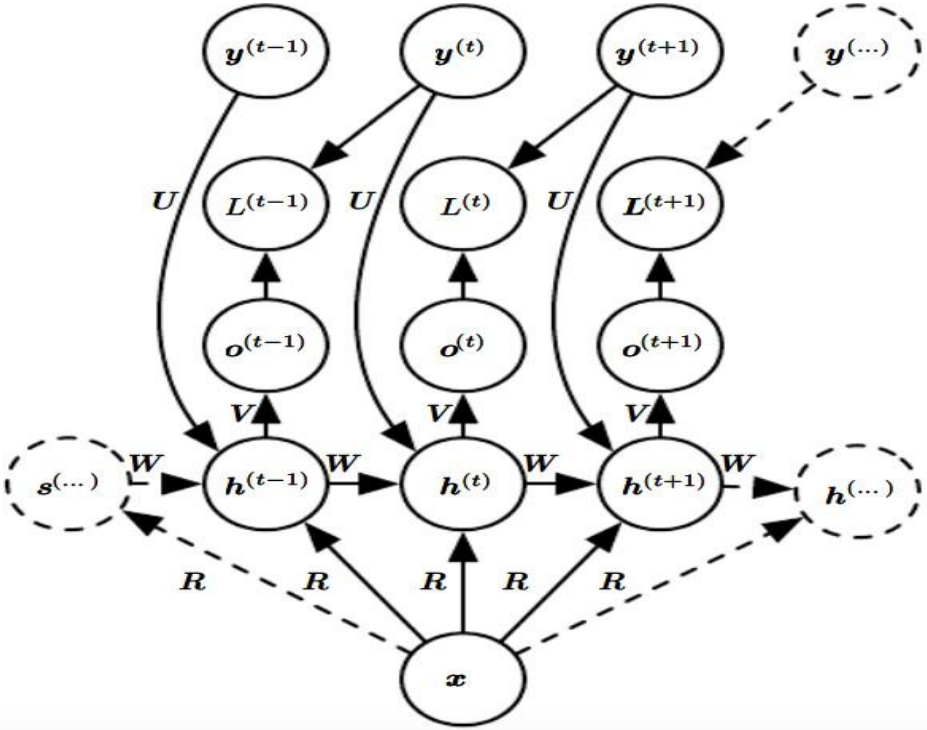


Figure 10.6

Teacher Forcing

- During training, the model receives the ground truth output $y^{(t)}$ as input at time $t + 1$ (and not the model output)
- At the time of deployment, the true output is not known and the correct output $y^{(t)}$ is approximated by the model's output $o^{(t)}$.

Taking only a single vector x as input



Taking only a single vector x as input

- The same product $x^T R$ is added as an extra input to the hidden unit at each time step

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

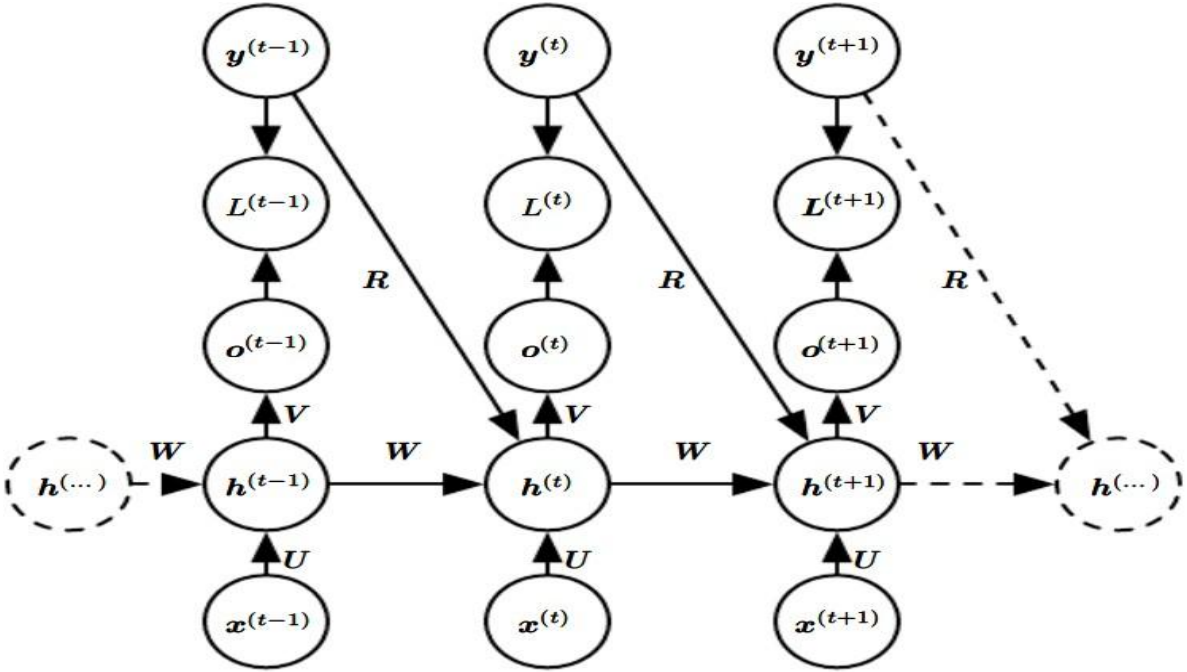
Removing the independence assumption

Conditional distribution $P(y^{(1)},..., y^{(\tau)}|x^{(1)}, \ldots, x^{(\tau)})$ makes a conditional independence assumption factorizing it as:

$$\prod_t P(y^{(t)}|x^{(1)}, \ldots, x^{(\tau)})$$

To remove the conditional independence assumption, we can add connections from the output at time t to the hidden unit at time $t + 1$.

Removing the independence assumption



Bidirectional RNNs

What we have seen till now?

- The state at time t only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$, and the present input $x^{(t)}$
- Some models also allow information from past y values to affect the current state when the y values are available

However, in many applications we want to output a prediction of $y^{(t)}$ which may depend on the whole input sequence. e.g., speech recognition, handwriting recognition

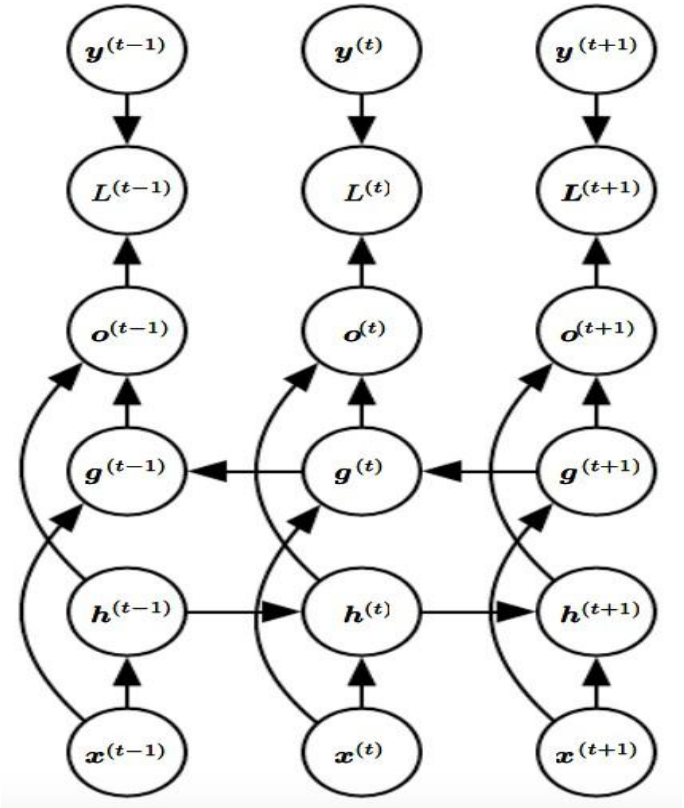
Bidirectional RNNs

What we have seen till now?

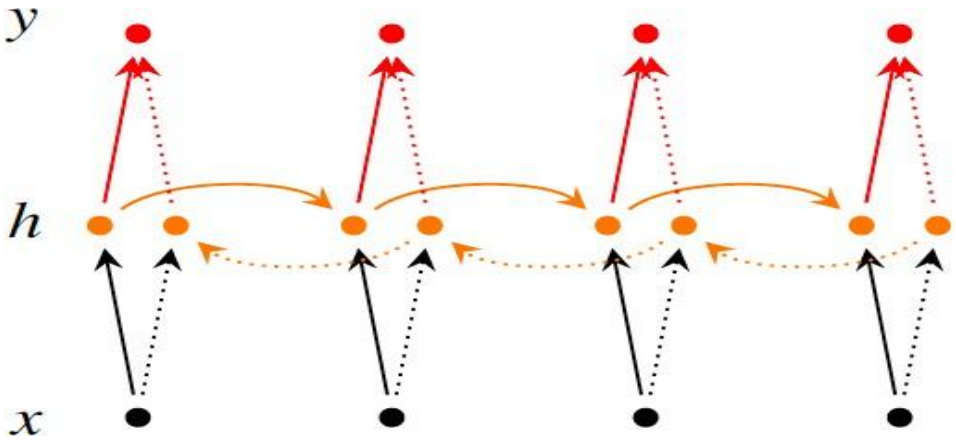
- The state at time t only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$, and the present input $x^{(t)}$
- Some models also allow information from past y values to affect the current state when the y values are available

However, in many applications we want to output a prediction of $y^{(t)}$ which may depend on the whole input sequence. e.g., speech recognition, handwriting recognition.

Bidirectional RNNs



Bidirectional RNNs: simplified representation



$$\vec{h}_t = f(\vec{W}x_t + \vec{V}\vec{h}_{t-1} + \vec{b})$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b})$$

$$y_t = g(U[\vec{h}_t; \overleftarrow{h}_t] + c)$$

Any more constraints?

What have we seen?

- Input sequence to a fixed-size vector
- Fixed-sized vector to sequence

- Input sequence to an output sequence of the same length

Any more constraints?

What have we seen?

- Input sequence to a fixed-size vector
- Fixed-sized vector to sequence
- Input sequence to an output sequence of the same length

Any other constraint?

Mapping input sequence to an output sequence, not necessarily of the same length (e.g. *Speech recognition, machine translation, question answering*)

Sequence-to-sequence architecture

Also known as encoder-decoder architecture

- Input sequence $X = (x^{(1)}, \ldots, x^{(nx)})$
- Output sequence $Y = (y^{(1)}, \ldots, y^{(ny)})$

Sequence-to-sequence architecture

Also known as encoder-decoder architecture

- Input sequence $X = (x^{(1)}, \ldots, x^{(nx)})$
- Output sequence $Y = (y^{(1)}, \ldots, y^{(ny)})$

Encoder (reader/input) RNN: Emits the context C, a vector summarizing the input sequence, usually as a simple function of its final hidden state

Sequence-to-sequence architecture

Also known as encoder-decoder architecture

- Input sequence $X = (x^{(1)}, \ldots, x^{(nx)})$
- Output sequence $Y = (y^{(1)}, \ldots, y^{(ny)})$

Encoder (reader/input) RNN: Emits the context C, a vector summarizing the input sequence, usually as a simple function of its final hidden state

Also known as encoder-decoder

(Writer/output) RNN: Is conditioned on the context C to generate the output sequence.

Sequence-to-sequence architecture

Also known as encoder-decoder architecture

- Input sequence $X = (x^{(1)}, \ldots, x^{(nx)})$
- Output sequence $Y = (y^{(1)}, \ldots, y^{(ny)})$

Encoder (reader/input) RNN: Emits the context C, a vector summarizing the input sequence, usually as a simple function of its final hidden state

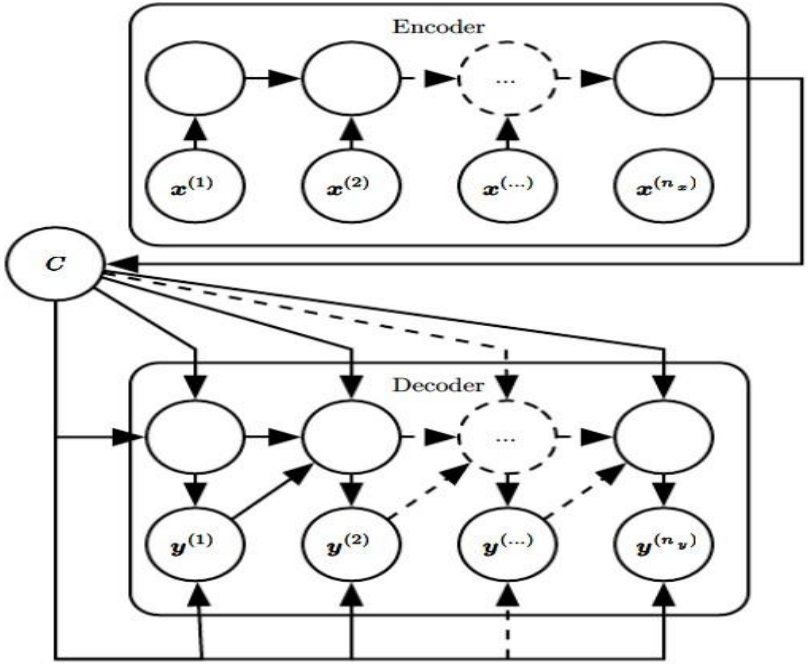
Also known as encoder-decoder

(Writer/output) RNN: Is conditioned on the context C to generate the output sequence.

What is the innovation?

The lengths n_x and n_y can vary from each other

Encoder-Decoder Architecture



Encoder and Decoder RNNs

- Last state of the encoder RNN is used as a representation C of the input sequence, provided as input to the decoder RNN

- Decoder RNN is a vector-to-sequence RNN, described earlier

- **THREE ways to receive input:**
  - as the initial state of the RNN, or
  - can be connected to the hidden units at each time step
  - combination of these two

Deep Recurrent Networks

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

from the input to the hidden state

from the previous hidden state to the next hidden state, and from the

hidden state to the output

Deep Recurrent Networks

The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

from the input to the hidden state
from the previous hidden state to the next hidden state, and from the hidden state to the output

In the basic model, each of these blocks is associated with a single weight matrix.

Deep Recurrent Networks

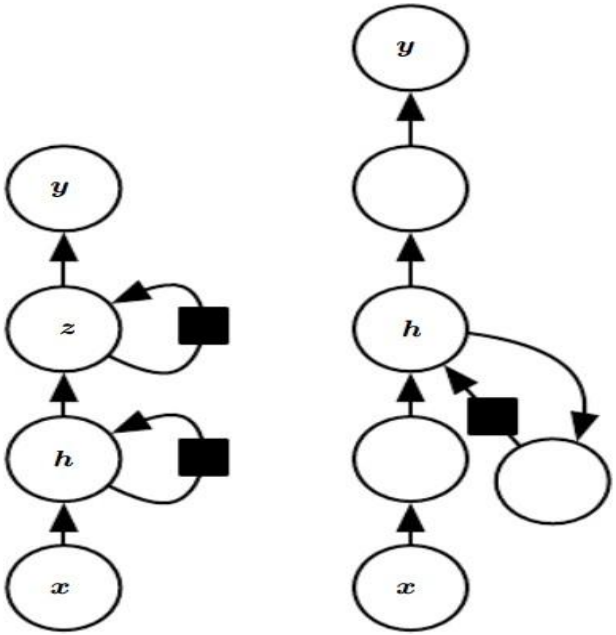
The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

- from the input to the hidden state
- from the previous hidden state to the next hidden state, and from
- the hidden state to the output

In the basic model, each of these blocks is associated with a single weight matrix.

Can one introduce depth in each of these operations?

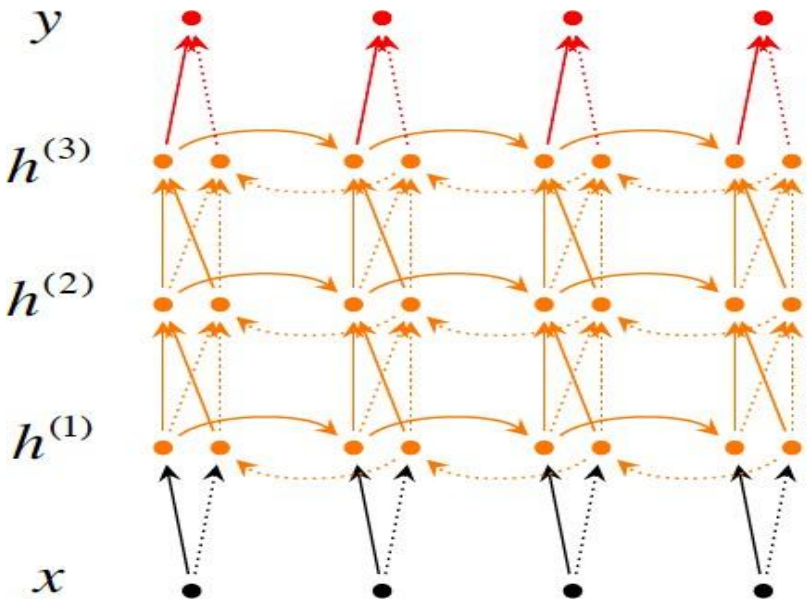
Deep RNNs



Deep RNNs

- The hidden recurrent state can be broken into groups organized hierarchically
- Deeper computations can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts

Deep Bidirectional RNNs



$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

$$y_t = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

Different configurations of RNNs

one to many

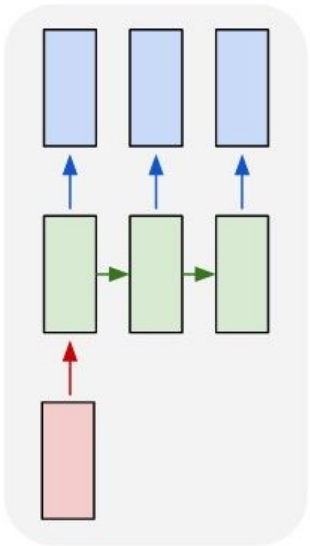
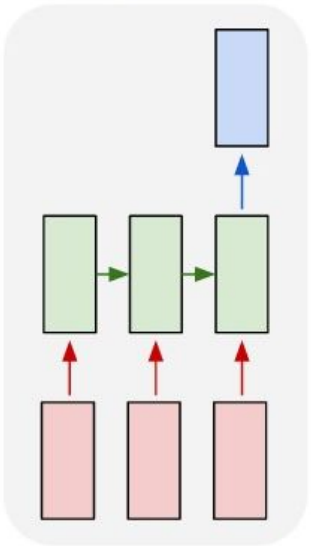


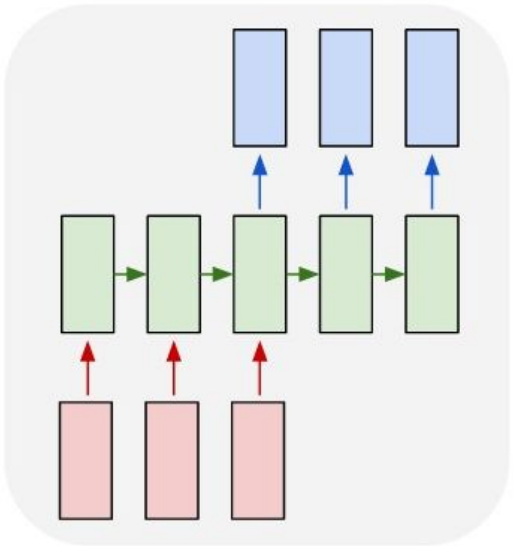
Image Captioning

many to one



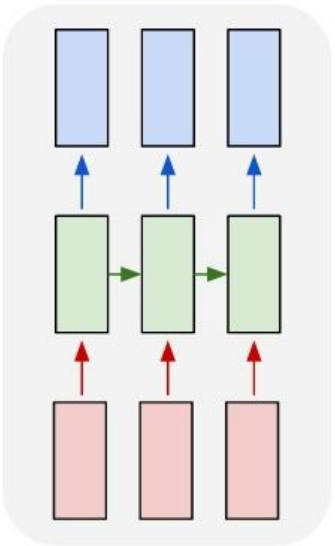
Sentiment Analysis

many to many



Machine Translation

many to many

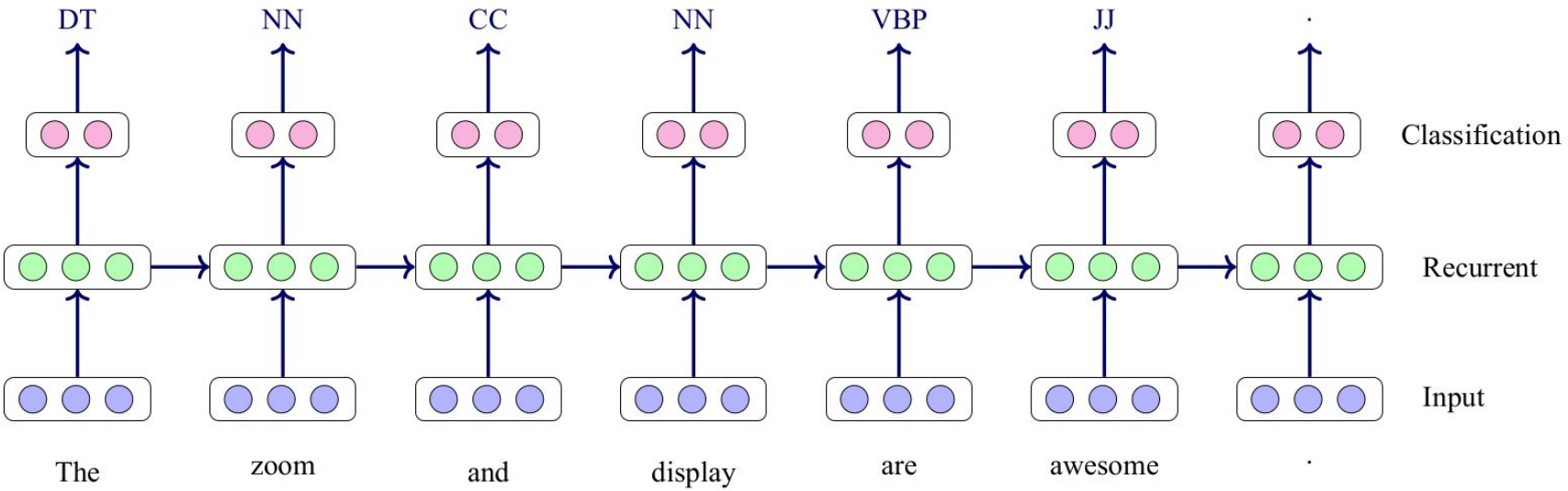


POS / NER / LM

RNN - Example 1

Part-of-speech tagging:

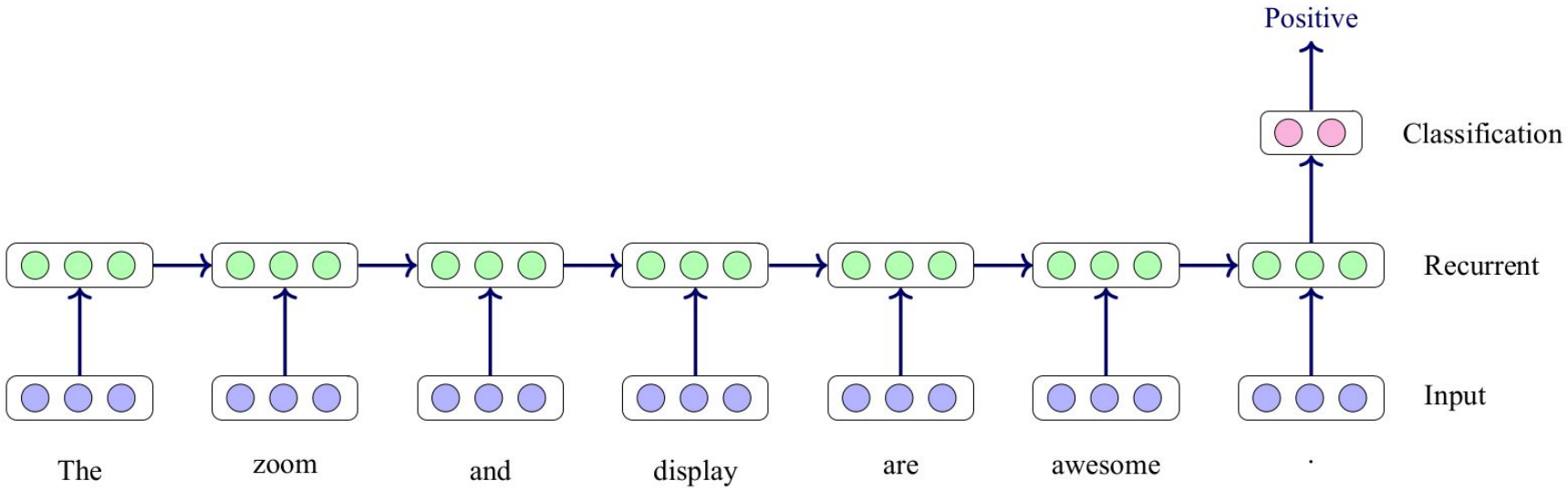
Given a sentence X, tag each word with its corresponding grammatical class



RNN - Example 2

Sentiment Classification:

Given a sentence X, find the associated sentiment polarity of the sentence.



Problems with RNNs

Better Units

Motivation

Better Units

Motivation

- RNNs not very good at capturing long-term dependencies – vanishing gradients

Better Units

Motivation

- RNNs not very good at capturing long-term dependencies – vanishing gradients
- Also, huge change to the hidden states

$$h_t = f(Wh_{t-1} + Ux_t)$$

Better Units

Motivation

- RNNs not very good at capturing long-term dependencies – vanishing gradients
- Also, huge change to the hidden states

$$h_t = f(Wh_{t-1} + Ux_t)$$

Basic Idea

- You should be able to keep some elements of 'h_t' somewhat constant so that they can carry over long term information

Better Units

Motivation

- RNNs not very good at capturing long-term dependencies – vanishing gradients
- Also, huge change to the hidden states

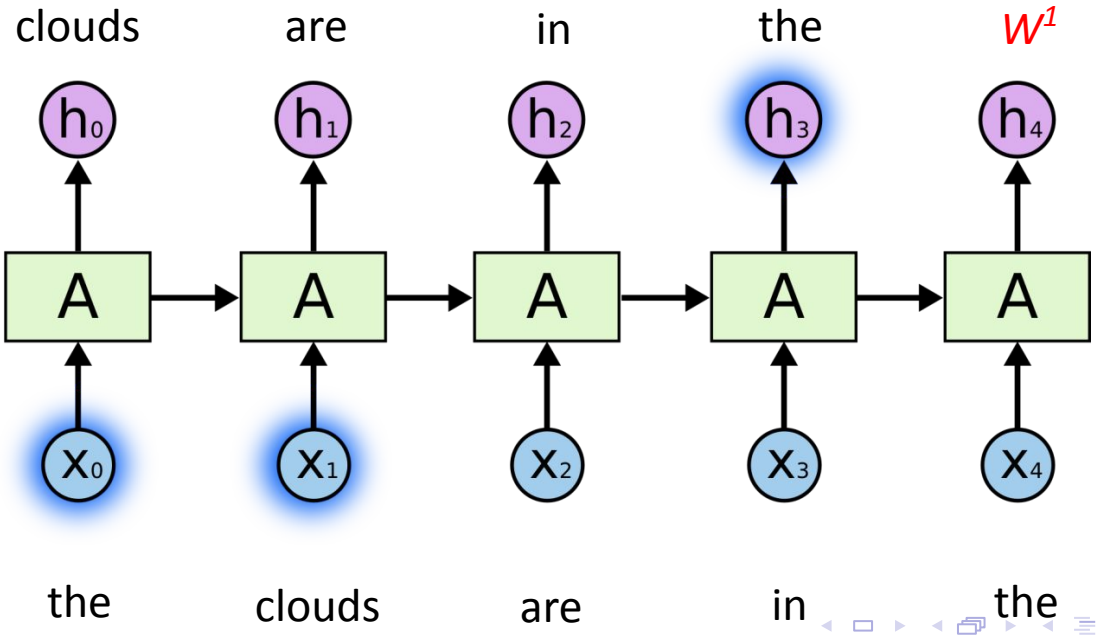
$$h_t = f(Wh_{t-1} + Ux_t)$$

Basic Idea

- You should be able to keep some elements of 'h_t' somewhat constant so that they can carry over long term information
- Gated Recurrent Units (GRU) and Long Short Term Memory (LSTM) networks let you control this further

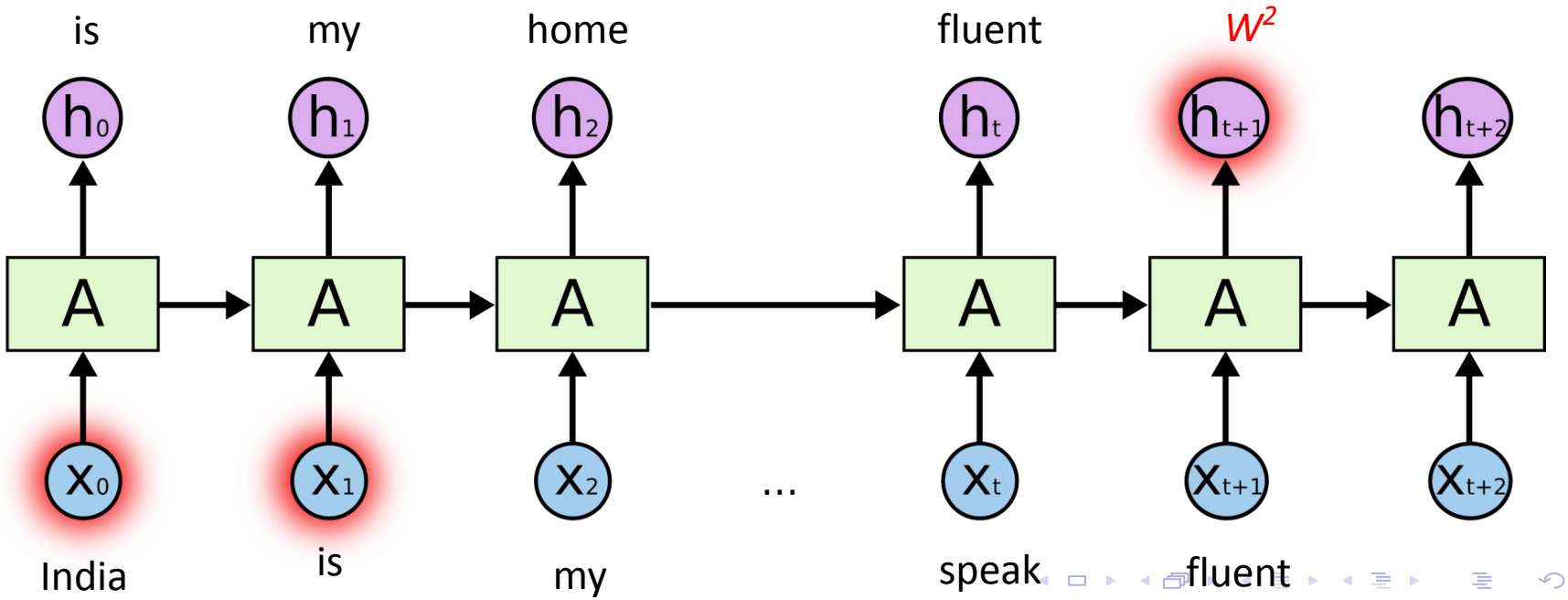
Language modelling: Example - 1

- "the clouds are in the *sky*"



Language modelling: Example - 2

- "India is my home country. I can speak fluent *Hindi*."

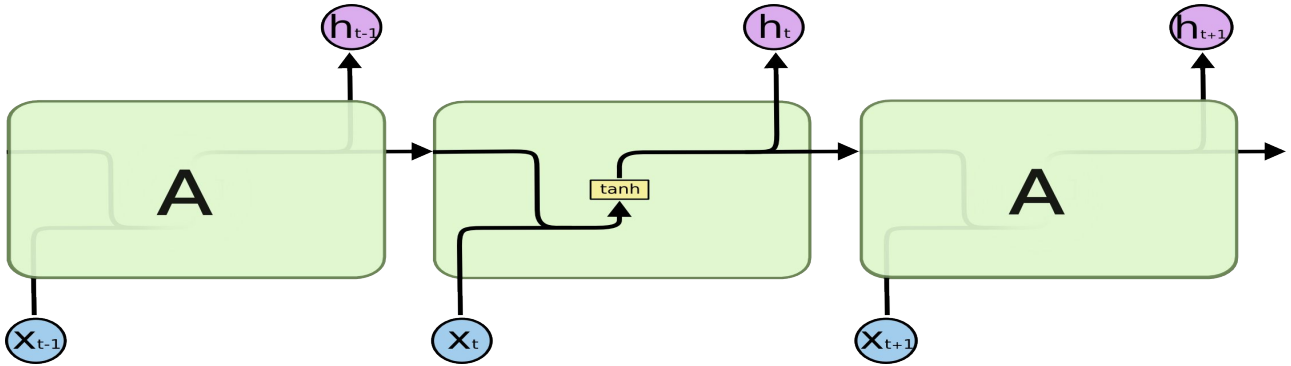


Vanishing/Exploding gradients

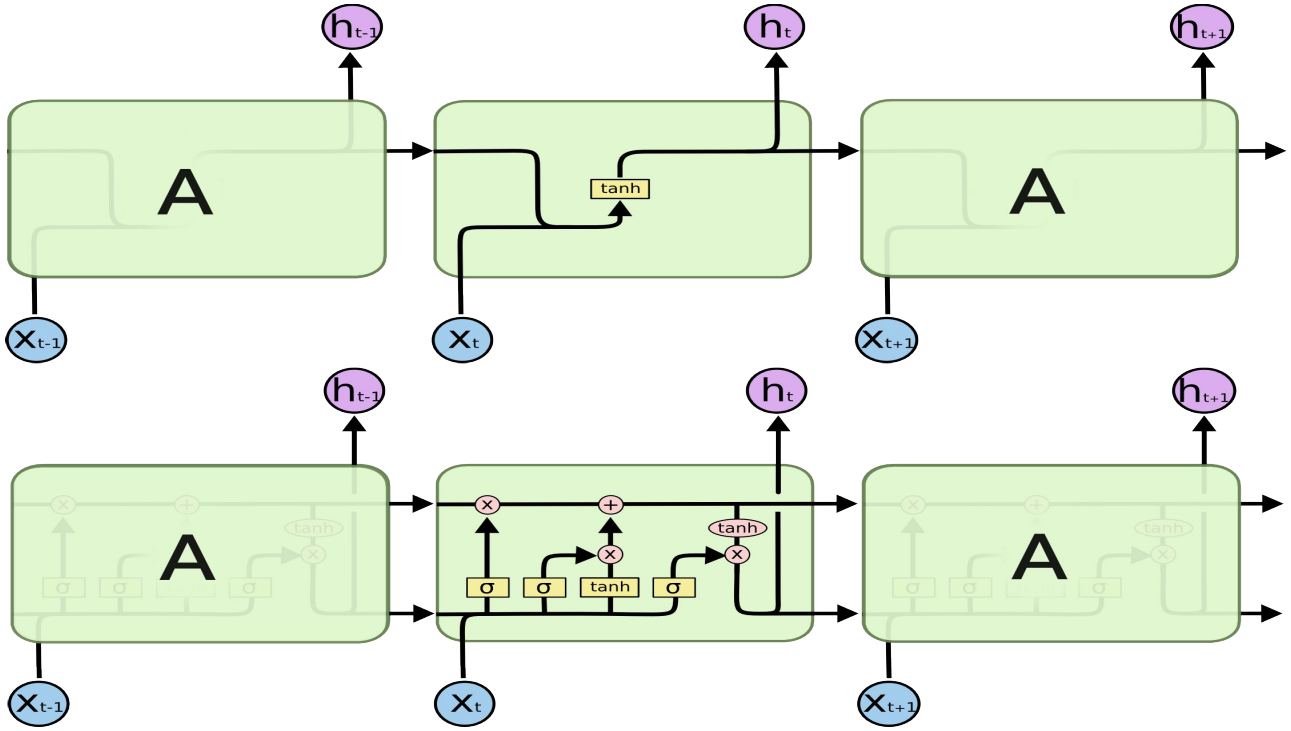
- Cue word for the prediction
  Example 1: ***sky*** → ***clouds*** [3 units apart]
  Example 2: ***Hindi*** → ***India*** [9 units apart]

- As the sequence length increases, it becomes hard for RNNs to learn ***long-term dependencies***
  Vanishing gradients: If weights are small, gradient shrinks exponentially. Network stops learning
  Exploding gradients: If weights are large, gradient grows exponentially. Weights fluctuate and become unstable

Long Short Term Memory (LSTM) Networks



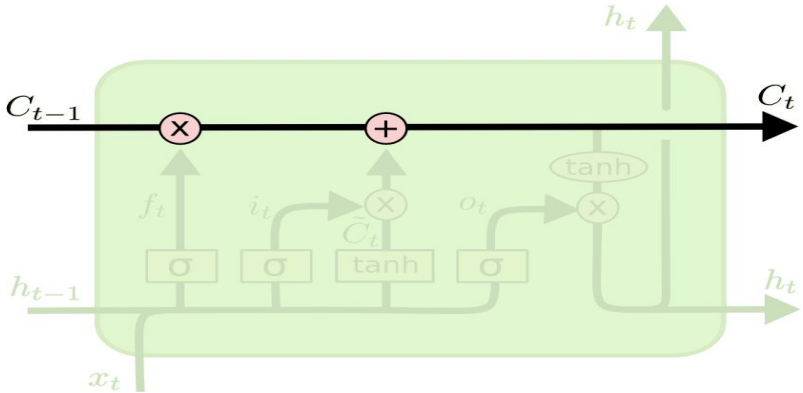
Long Short Term Memory (LSTM) Networks



The repeating module in LSTM contains four interacting layers

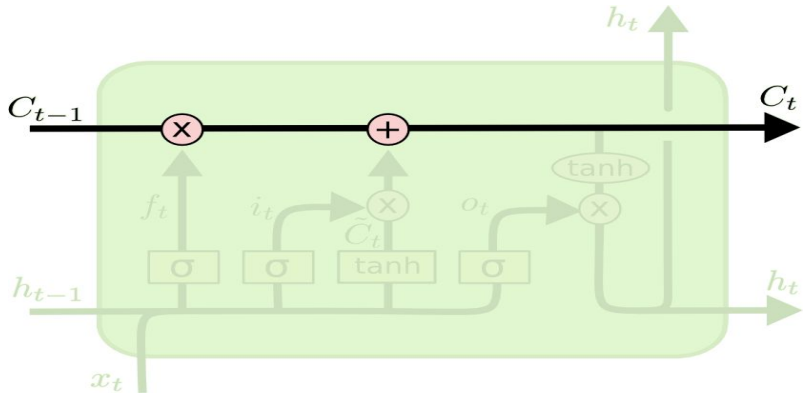
Core idea behind LSTMs

The key to LSTMs is the cell state, horizontal line running through the top of the diagram. It is easy for information to flow along it unchanged



Core idea behind LSTMs

The key to LSTMs is the cell state, horizontal line running through the top of the diagram. It is easy for information to flow along it unchanged.



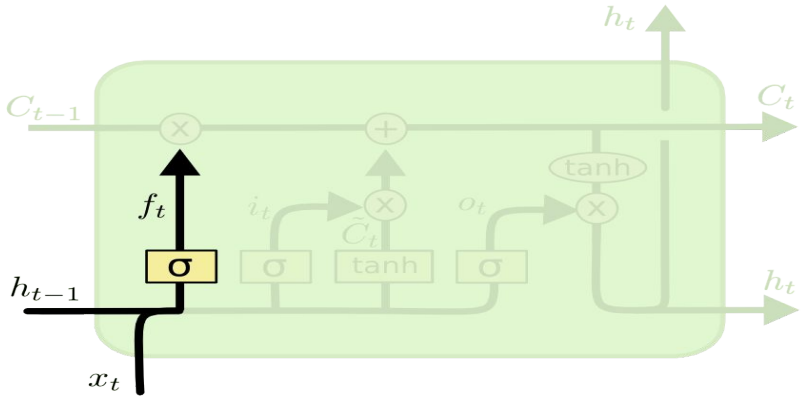
- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by the structures called gates
- Gates are a way to optionally let information through. The sigmoid layer in the gate outputs numbers between zero and one, describing how much of each component should be let through

What information to throw away?

Forget gate decides what information we are going to throw away from the cell state.

What information to throw away?

Forget gate decides what information we are going to throw away from the cell state



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

A 1 represents “completely keep this” while a 0 represents “completely get rid of this”

What new information to keep in the cell state?

Two parts

- Input gate decides which values we will update
- A *tanh* layer creates a vector of new candidate values $\tilde{C}$, to be added to the state

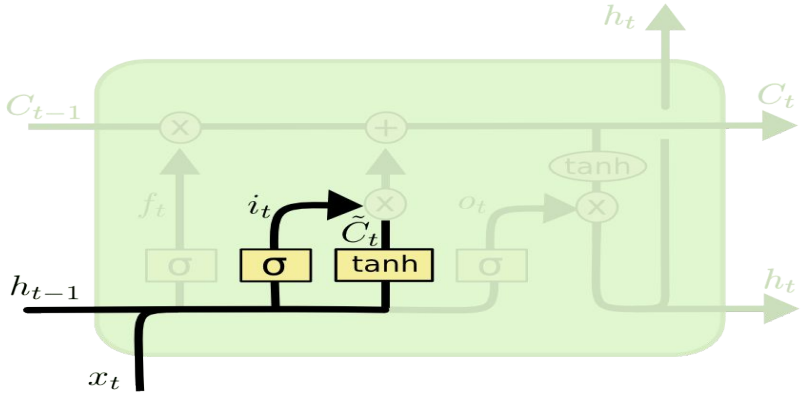
These two are then combined to create an update to the state

What new information to keep in the cell state?

Two parts

- Input gate decides which values we will update
- A *tanh* layer creates a vector of new candidate values $\tilde{C}$, to be added to the state

These two are then combined to create an update to the state



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

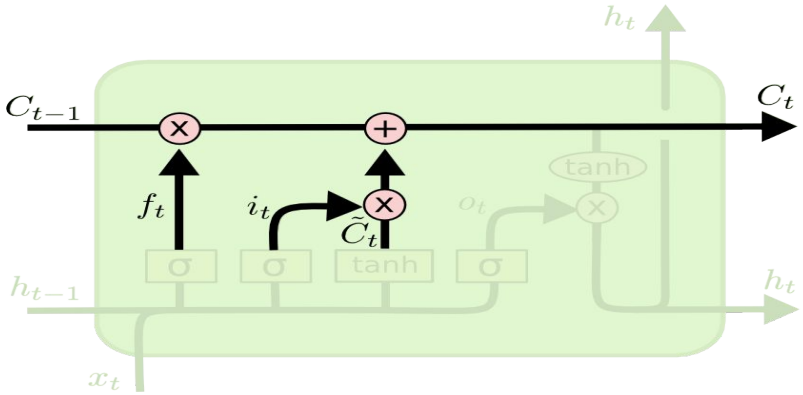
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Updating the old Cell state

- We multiply the old state by f_t, forgetting the things we decided to forget earlier
- Then we add $i_t \circ \tilde{C}_t$ – new candidate values, scaled by how much we decided to update each state value

Updating the old cell state

- We multiply the old state by f_t, forgetting the things we decided to forget earlier
- Then we add $i_t \circ \tilde{C}_t$ – new candidate values, scaled by how much we decided to update each state value



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

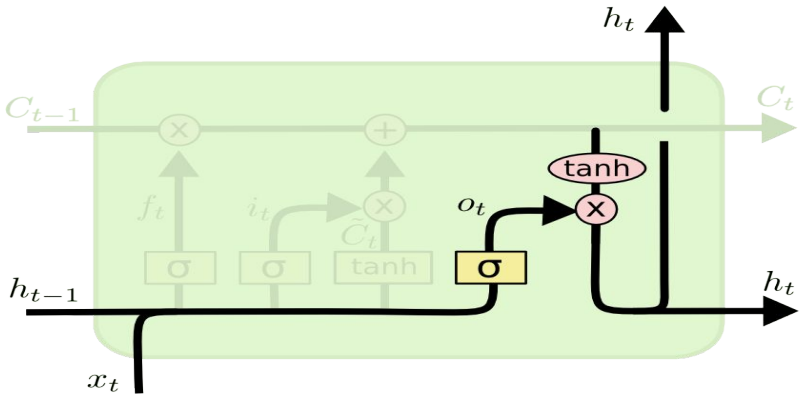
Deciding the output

- Output will be based on the cell state, but a filtered version
- A sigmoid layer decides what parts of the cell state we are going to output

*We put the cell state through **tanh** and multiply it by the output of the **sigmoid gate**, so that we only output the parts we decided to.*

Deciding the output

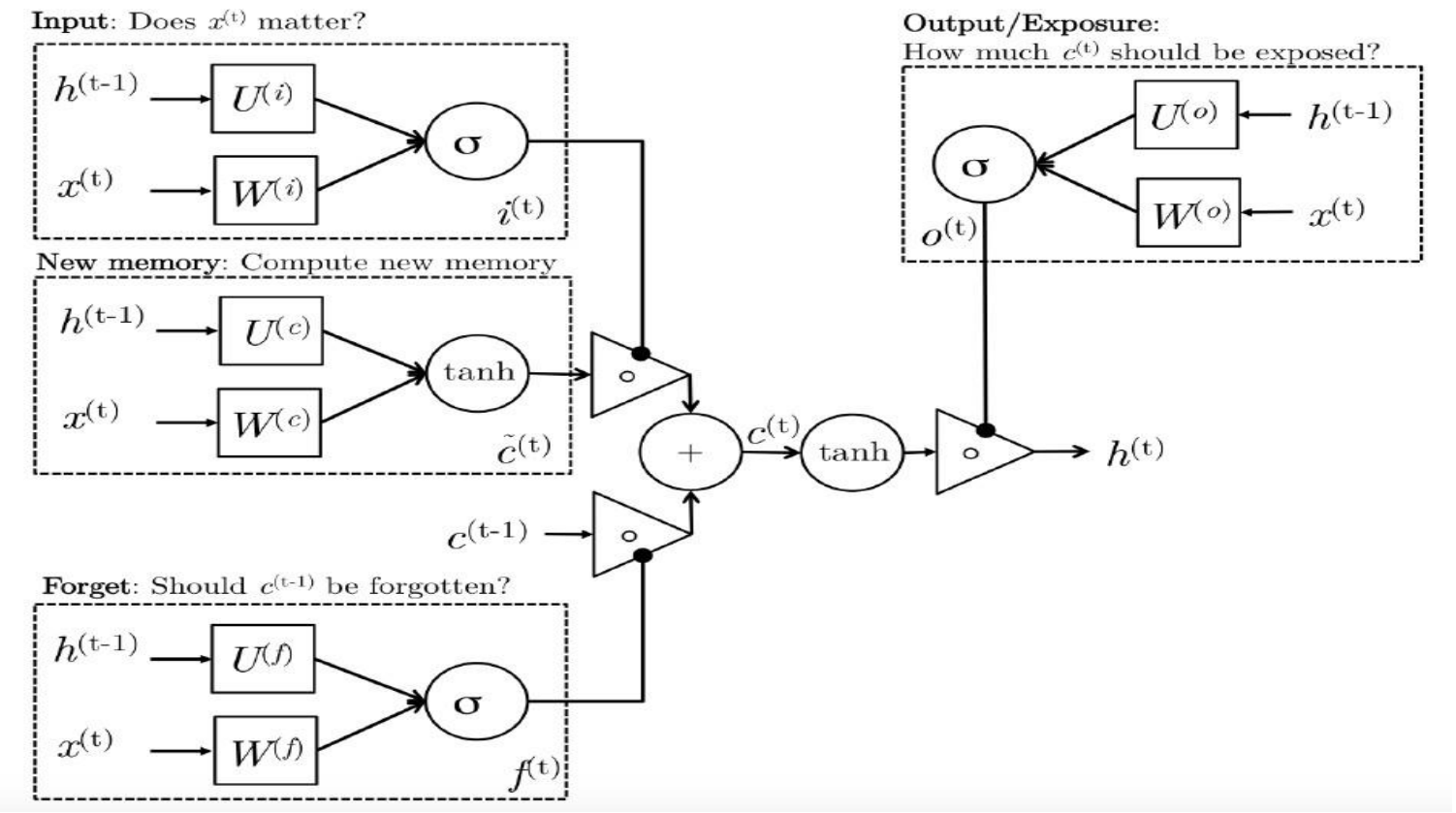
- Output will be based on the cell state, but a filtered version
- A sigmoid layer decides what parts of the cell state we are going to output
- We put the cell state through *tanh* and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to



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

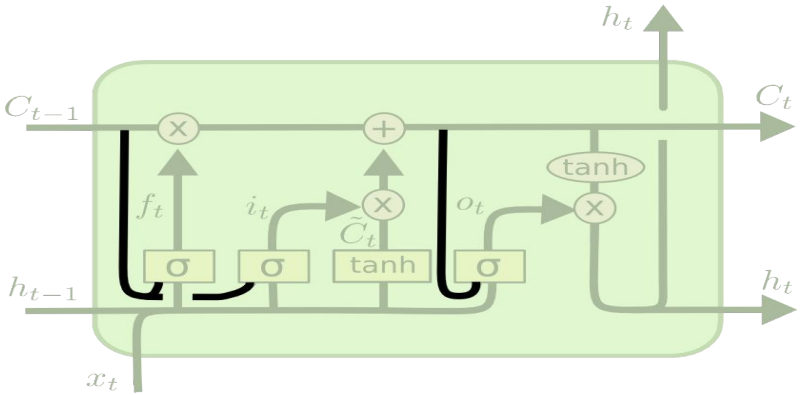
$$h_t = o_t * \tanh\left(C_t\right)$$

Internals of LSTM



LSTM Variation: Peephole Connections

We let the gate layers look at the cell state.

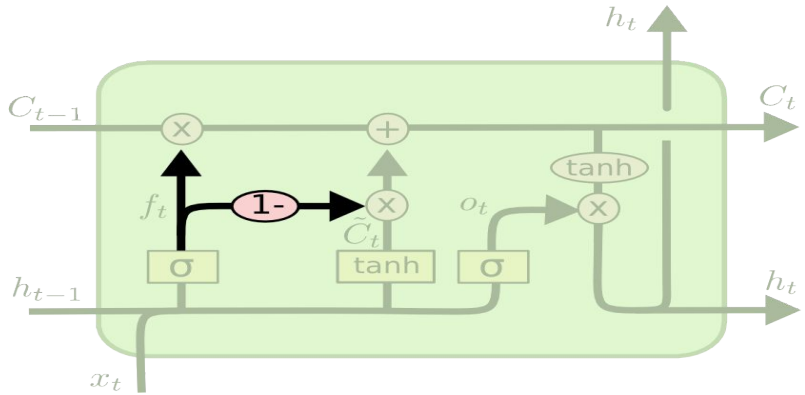


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

Some variants may add only some of the peepholes.

LSTM Variation:Coupled forget and input gates

Instead of separately deciding what to forget and what we should add new information to, we make those decisions together.



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

From LSTM to GRU

- GRU is the newer generation of RNN and is pretty similar to an LSTM
- GRU *got rid of the cell state and used the hidden state to transfer information*
- Has only two gates: a ***reset gate*** and ***update gate***
- **Update Gate**
  - acts similar to the forget and input gate of an LSTM
  - decides what information to throw away and what new information to add
- **Reset Gate**
  - used to decide how much past information to forget

GRU's has fewer operations; therefore, they are a little speedier to train than LSTM's

BUT, there isn't any clear winner which one is better

GRUs

Note

Let's forget about the output, only focus on computation of hidden states

GRUs

Note

Let's forget about the output, only focus on computation of hidden states

See it as a black box first

You are still computing new state h_t based on current input x_t and previous state h_{t-1}

GRUs

Note

Let's forget about the output, only focus on computation of hidden states

See it as a black box first

- We are still computing new state h_t based on current input x_t and previous state h_{t-1}
- *You use some additional gates – **update gate and reset gate,** that let you control how much previous memory (state) you want to carry forward, whether you want to ignore the current input etc.*

GRUs

Update gate

$$z_t = \sigma(W^{(z)} h_{t-1} + U^{(z)} x_t)$$

GRUs

Update gate

$$z_t = \sigma(W^{(z)} h_{t-1} + U^{(z)} x_t)$$

Reset gate

$$r_t = \sigma(W^{(r)} h_{t-1} + U^{(r)} x_t)$$

GRUs

Update gate

$$z_t = \sigma(W^{(z)} h_{t-1} + U^{(z)} x_t)$$

Reset gate

$$r_t = \sigma(W^{(r)} h_{t-1} + U^{(r)} x_t)$$

Note: both these are vectors

GRUs

Update gate

$$z_t = \sigma(W^{(z)} h_{t-1} + U^{(z)} x_t)$$

Reset gate

$$r_t = \sigma(W^{(r)} h_{t-1} + U^{(r)} x_t)$$

Note: both these are vectors

New memory content

$$\tilde{h}_t = tanh(U x_t + r_t \circ W h_{t-1})$$

GRUs

Update gate

$$z_t = \sigma(W^{(z)} h_{t-1} + U^{(z)} x_t)$$

Reset gate

$$r_t = \sigma(W^{(r)} h_{t-1} + U^{(r)} x_t)$$

Note: both these are vectors

New memory content

$$\tilde{h}_t = tanh(U x_t + r_t \circ W h_{t-1})$$

If reset gate is 0, it ignores previous memory and only stores the current word information

GRUs

Update gate

$$z_t = \sigma(W^{(z)} h_{t-1} + U^{(z)} x_t)$$

Reset gate

$$r_t = \sigma(W^{(r)} h_{t-1} + U^{(r)} x_t)$$

Note: both these are vectors

New memory content

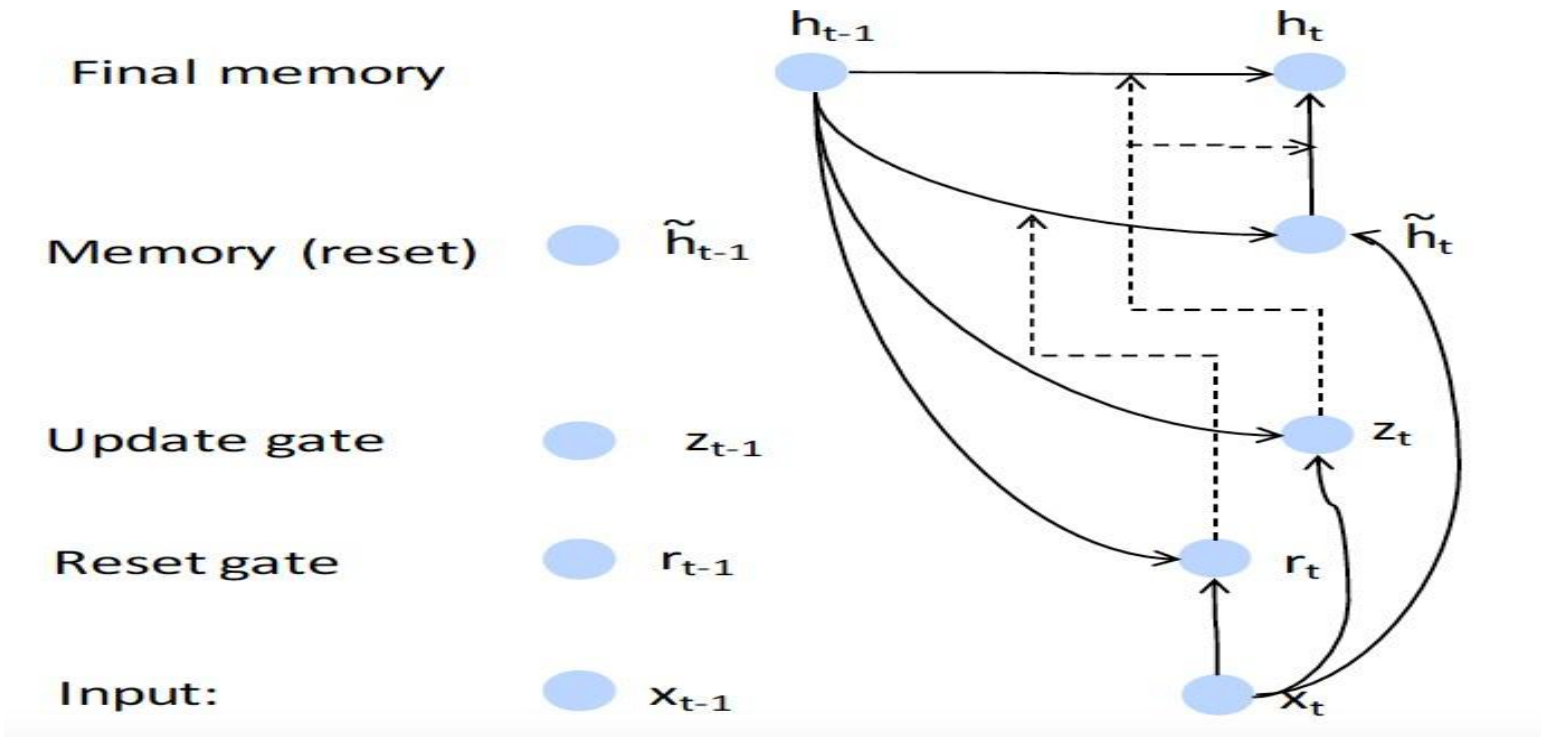
$$\tilde{h}_t = tanh(U x_t + r_t \circ W h_{t-1})$$

If reset gate is 0, it ignores previous memory and only stores the current word information

Final memory (current state)

$$\tilde{}$$

GRU: Another Illustration



GRU Intuition

If reset is close to 0

Ignore previous hidden state

GRU Intuition

If reset is close to 0

Ignore previous hidden state

→ allows model to drop the information that is irrelevant in future

GRU Intuition

If reset is close to 0

Ignore previous hidden state

→ allows model to drop the information that is irrelevant in future

Update gate z

controls how much of the past state should matter now

GRU Intuition

If reset is close to 0

Ignore previous hidden state

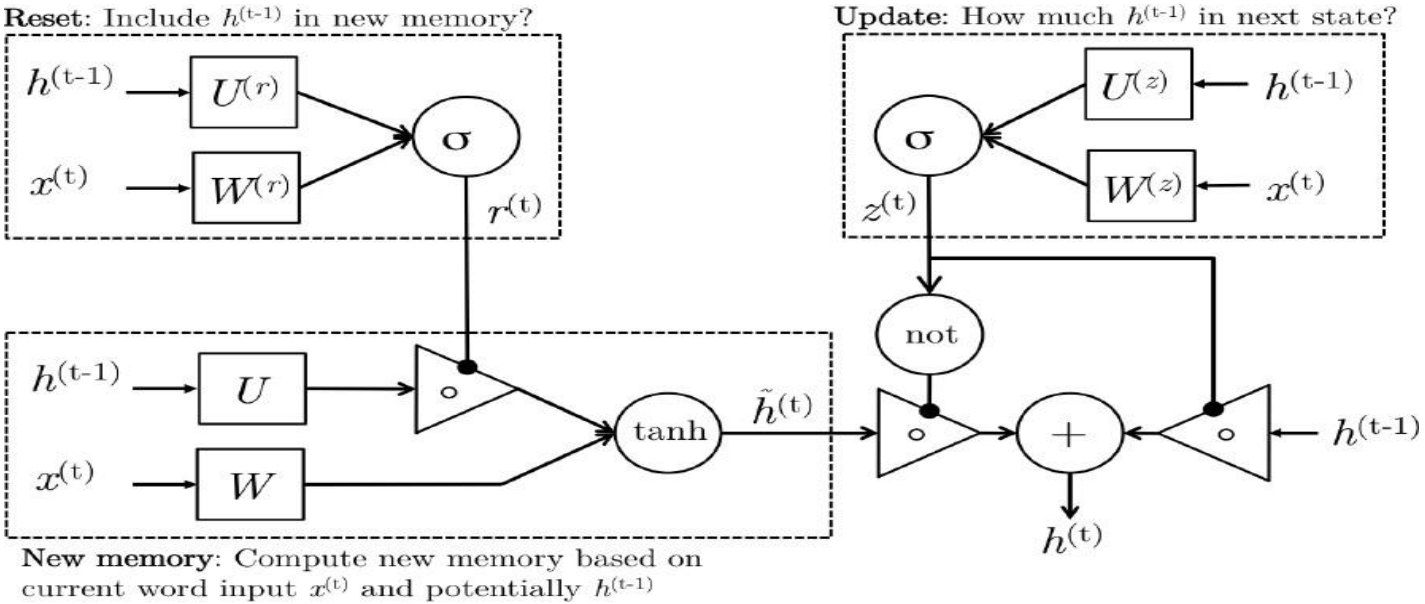
→ allows model to drop the information that is irrelevant in future

Update gate z

controls how much of the past state should matter now

→ If z is close to 1, then we can copy information in that unit through many time steps. Less vanishing gradient!!

Internals of GRU



Thank you for your Attention!